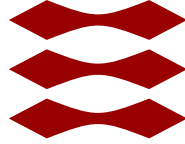

DTU



02247 Compiler Construction

Spring 2023

Alceste Scalas

<alcs@dtu.dk>

May 08, 2023



**This work is licensed under a
Creative Commons Attribution-NonCommercial-NoDerivatives
4.0 International License**

Contents

0	Module 0: Overview of the Course and Assessment	3
0.1	What is a Compiler?	3
0.2	Course Objectives and Rationale	5
0.3	A Taste of Hygge	6
0.4	Tools and Programming Languages Used During the Course	8
0.5	Assessment: Group Project and Oral Group Examination	9
0.6	Group Work Organisation: Some Suggestions	11
1	Module 1: Introduction to RISC-V	13
1.1	What is RISC-V? And Why Is It Relevant?	13
1.2	Base and Floating-Point Registers	14
1.3	A Few RISC-V Assembly Instructions	15
1.4	RISC-V Assembly Program Structure	19
1.5	RARS – RISC-V Assembler and Runtime Simulator	22
1.6	References and Further Readings	24
1.7	Lab Exercises	25
2	Module 2: The Hygge0 Language Specification	29
2.1	Formal Syntax of Hygge0	30
2.2	Formal Semantics of Hygge0	36
2.3	Type-Checking Hygge0 Programs	43
2.4	References and Further Readings	52
2.5	Lab Activities	53
3	Module 3: Hands-On with <code>hyggec</code>	55
3.1	Quick Start	55
3.2	The Compiler Phases of <code>hyggec</code>	55
3.3	Overview of the <code>hyggec</code> Source Tree	57
3.4	The Abstract Syntax Tree	58
3.5	The Lexer and Parser	62
3.6	The Built-In Interpreter	71
3.7	Types and Type Checking	74
3.8	Code Generation	79

3.9	The Test Suite of <code>hygdec</code>	85
3.10	Example: Extending <code>Hygge0</code> and <code>hygdec</code> with a Subtraction Operator	86
3.11	Project Ideas	93
4	Module 4: Lab Day	99
5	Module 5: Mutability and Loops	101
5.1	Overall Objective	101
5.2	Mutable Variables	102
5.3	“While” Loop	116
5.4	Project Ideas	124
6	Module 6: Functions and the RISC-V Calling Convention	129
6.1	Overall Objective	129
6.2	Syntax	131
6.3	Operational Semantics	132
6.4	Typing Rules	135
6.5	The RISC-V Memory Layout, Stack, and Calling Convention	138
6.6	Implementation	161
6.7	Limitations of the Current Specification and Code Generation	168
6.8	References and Further Readings	168
6.9	Project Ideas	169
7	Module 7: Structured Data Types and the Heap	177
7.1	Overall Objective	177
7.2	Syntax	179
7.3	Operational Semantics	180
7.4	Typing Rules	185
7.5	Implementation	189
7.6	References and Further Readings	194
7.7	Project Ideas	194
8	Module 8: Lab Day	201
9	Module 9: Closures	203
9.1	Overall Objective	203
9.2	What is a Closure?	205
9.3	Closures that Capture Immutable Variables	209
9.4	Closures that Capture Mutable Variables	213
9.5	Closures that Capture Top-Level Variables	216
9.6	Implementation	218
9.7	References and Further Readings	219
9.8	Project Ideas	220
10	Module 10: Discriminated Unions and Recursive Types	227
10.1	Discriminated Union Types and Pattern Matching	227
10.2	Recursive Types	239
10.3	References and Further Readings	244
10.4	Project Ideas	244

11	Module 11: Intermediate Representations and Register Allocation	249
11.1	Overall Objective	249
11.2	What is an Intermediate Representation (IR)?	250
11.3	Administrative Normal Form (ANF)	251
11.4	Transformation of a Hygge Expression into ANF	254
11.5	ANF-Based Linear Register Allocation	261
11.6	Implementation: ANF Transformation and Register Allocation in hygge	266
11.7	References and Further Readings	275
11.8	Project Ideas	276
12	Module 12: Optimisation	279
12.1	Overall Objective	279
12.2	Partial Evaluation	281
12.3	Copy Propagation and Common Subexpression Elimination (CSE)	288
12.4	Peephole Optimisation	292
12.5	References and Further Readings	295
12.6	Project Ideas	296
A	ChangeLog	299

These are the lecture notes of the course 02247 Compiler Construction at DTU Compute.

These lecture notes are also available in [HTML format, readable on a browser](#)¹.

Important: The **home page for this course** (with course plan, calendar, groups, project submission...) is on DTU Learn: <https://learn.inside.dtu.dk/d21/home/145305>

These lecture notes will be updated throughout the course. The latest updates are listed in the *ChangeLog*.

¹ <http://courses.compute.dtu.dk/02247/f23>

Module 0: Overview of the Course and Assessment

Here is an overview of what you will learn in this course, which tools will be used, and how you will be assessed. This module discusses, in particular, the *Tools and Programming Languages Used During the Course* and the *Assessment: Group Project and Oral Group Examination*.

0.1 What is a Compiler?

A **compiler** is a translator from code written in a **source language**, into code written in a **target language**. Typically, the source language operates at a higher level of abstraction, whereas the target language operates at a lower level of abstraction. For example:

- compiling C++ code into C code (a common approach of the early C++ compilers);
- compiling C code into x86 assembly;
- compiling TypeScript into WebAssembly;
- compiling Java code into Java Virtual Machine bytecode;
- compiling F# into .NET Common Language Runtime bytecode;
- ...

When the source and target languages have similar levels of abstraction, we typically use the term **transpiler** instead of compiler.

A compiler typically performs the source-to-target language translation in a series of **phases**: each phase takes as input a representation of the code being compiled, and outputs another representation. This way, an overall complex source-to-target translation is broken down into simpler steps, each one having a distinct purpose.

Fig.1 outlines the basic phases of a compiler, what they receive as input, and what they produce as output.

Note: The number of phases and nomenclature may vary between compilers; moreover, a phase can be sometimes split into sub-phases, and two or more phases may be merged

into one.

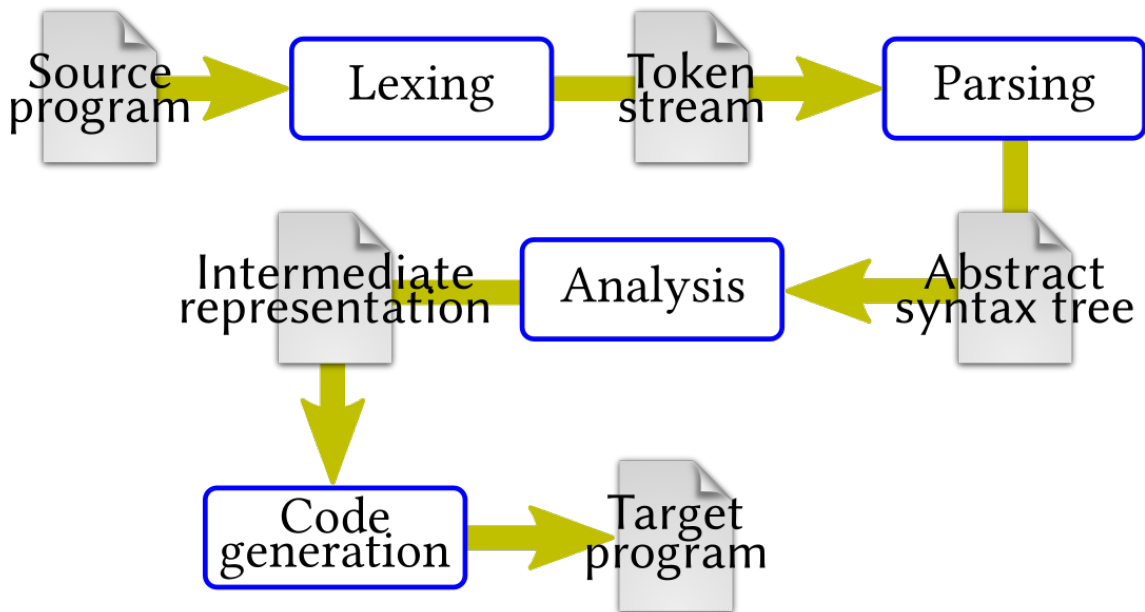


Fig. 1: Basic phases of a compiler.

Let us explore the purpose of the phases in Fig.1 with an example. Consider the following simple **source program**:

```
println(2 + 3) // Should print a number between 4 and 37
```

The **lexing** phase (a.k.a. **tokenization**) classifies groups of characters in the source program, by recognising e.g. keywords, parentheses, literal integers, operators. Recognised groups of characters are transformed into **tokens**, and irrelevant information in the source program (e.g. white spaces, comments) can be discarded. For example, a tokenization of the program above may look like:

```
PRINTLN; LPAR; LIT_INT 2; PLUS; LIT_INT 3; RPAR
```

The **parsing** phase reads a stream of tokens and applies a series of **grammar rules** to reconstruct the syntactic structure of the source code, creating an **Abstract Syntax Tree (AST)**. An analogy with human languages is: each token represents a valid word in a given language, and the parser checks whether a sequence of words forms a grammatically-valid sentence. For example, the sequence of tokens above forms the following AST, which means: there is a top-level `PrintLn` expression with an argument sub-expression, which in turn is an `Addition` having two sub-expressions: an `lhs` (left-hand side) which is an `IntValue 2`, and a `rhs` (right-hand side) which is an `IntValue 3`.

```
PrintLn
├─arg: Add
│   ├──lhs: IntVal 2
│   └──rhs: IntVal 3
```

The **analysis** phase checks whether a given AST is “correct”, and produces an **intermediate representation (IR)** useful for further compilation. The kind of analysis being performed, and the details of the IR, may vary between compilers. A common scenario is: the analysis is based on **type checking**, and the produced IR may be a **typed AST** similar to the input AST, but augmented with type information. For example, the type-checking of the AST above may produce the following typed AST:

```
PrintLn; type: unit
└─arg: Add; type: int
    └─lhs: IntVal 2; type: int
        └─rhs: IntVal 3; type: int
```

Finally, the **code generation** phase translates the intermediate representation into a target program. For example, if the compiler produces **assembly code**, then the program generated from the typed AST above may look like:

```
.text:
    li t0, 2
    li t1, 3
    add t0, t0, t1
    addi sp, sp, -8
    sw a7, 0(sp)
    sw a0, 4(sp)
    mv a0, t0
    li a7, 1
    ecall
    lw a7, 0(sp)
    lw a0, 4(sp)
    addi sp, sp, 8
    li a7, 10
    ecall
```

0.2 Course Objectives and Rationale

This is a **project-based course**: you’ll learn how to design and extend a compiler by **working on a compiler**, combining theory and hands-on experience. To this purpose, we study how to compile a **high-level programming language** (with features found in many “mainstream” programming languages) into **low-level assembly code** for a **real-world CPU architecture**: RISC-V.

The main learning tool for this course is a small high-level programming language called **Hygge** (/ˈhygə/), designed for teaching compiler construction.

The Hygge language is designed to be easy to use and to parse, so we can focus on the more interesting aspects of its compilation process. You can think of Hygge as a micro-F# with a syntax reminiscent of C (see *A Taste of Hygge*). We will study how to translate an Hygge program into an executable RISC-V assembly program.

Due to time constraints, this course does not attempt to build a Hygge compiler com-

pletely from scratch. Instead, we start from a **skeleton (and incomplete) compiler called `hygdec`**. You will work on a **group project** to improve both the Hygge language and the `hygdec` skeleton compiler, by designing and implementing various new features (either suggested by the teacher, or proposed by you).

The aim of `hygdec` is to provide an approachable starting point for the course project: its source code is compact and contains many comments, and it is designed to be easy to debug and extend. Moreover, `hygdec` provides some handy facilities to speed up compiler development: e.g. a pretty-printer, an internal API to create and combine fragments of target RISC-V code, and an automatic testing framework.

0.3 A Taste of Hygge

At the beginning of the course, we will focus on a minimalistic fragment of the Hygge programming language (called **Hygge0**) and explore how to compile it into RISC-V assembly. Hygge0 is little more than a calculator: it only supports variables, a few arithmetic operations, if-then-else, and reading/writing data from/to the terminal. An Hygge0 program looks like the following:

```
1 let x: int = 1;
2 let y: int = 2;
3
4 if x < y then println("x is smaller than y")
5     else println("x is not smaller than y");
6
7 print("The result of x + y is: ");
8 println(x + y)
```

You will learn how to improve Hygge0 and its compiler by adding new simple programming language constructs — in particular, new operators.

After this experience, we will explore the more complete Hygge programming language (which builds upon Hygge0) and you will learn how to design and implement more advanced features to fix its limitations and expand its capabilities. Towards the end of the course, we will be able to compile into RISC-V and run some rather complex Hygge programs, like the following:

```
1 // Define a list type as a labelled union type: either an empty list ('Nil'),
2 // or a list 'Elem'ent followed by the rest of the list.
3 type List = union {
4     Nil: unit;
5     Elem: struct {
6         value: int;
7         rest: List
8     }
9 };
10
11 // Is the given list empty?
```

(continues on next page)

(continued from previous page)

```

12 fun isEmpty(l: List): bool = {
13     match l with {
14         Nil{ _ } -> true;
15         Elem{ _ } -> false
16     }
17 };
18
19 // Create a list of ints starting with 'start' and ending with 'end' (included).
20 fun range(start: int, end: int): List = {
21     if (start < end or start = end)
22         then Elem{struct{value = start; rest = range(start+1, end)}}
23         else Nil{()}
24 };
25
26 // Compute and return the length of the given list.
27 fun length(l: List): int = {
28     match l with {
29         Nil{ _ } -> 0;
30         Elem{e} -> 1 + length(e.rest)
31     }
32 };
33
34 // Display the elements of the given list: show the elements between square
35 // brackets, with a semicolon between consecutive elements, and a final newline.
36 fun display(l: List): unit = {
37     // Internal helper function to display each list element
38     fun displayRec(l: List): unit = {
39         match l with {
40             Nil{ _ } -> ();
41             Elem{e} -> {
42                 print(e.value);
43                 if not isEmpty(e.rest) then print("; ")
44                     else ();
45                 displayRec(e.rest)
46             }
47         }
48     };
49     print("[");
50     displayRec(l);
51     println("]");
52 };
53
54 // Map the given function over the given list
55 fun map(f: (int) -> int, l: List): List = {
56     match l with {
57         Nil{ _ } -> Nil{()};
58         Elem{e} -> Elem{struct{ value = f(e.value);
59                                 rest = map(f, e.rest) }}
60     }

```

(continues on next page)

(continued from previous page)

```
61 };
62
63 let l: List = range(1, 42);
64 print("The length of the list 'l' is: ");
65 println(length(l));
66
67 print("The elements of the list 'l' are: ");
68 display(l);
69
70 // Create a list 'l2' by adding 1 to each element of 'l'
71 let l2: List = map(fun(x: int) -> x + 1, l);
72 print("The elements of the list 'l2' are: ");
73 display(l2)
```

0.4 Tools and Programming Languages Used During the Course

This section summarises what *Software Requirements* you should install to follow the course, and provides some pointers about the *F# Programming Language*.

0.4.1 Software Requirements

To run and modify the Hygge compiler, and to use the tools recommended for this course, you will need to install on your computer:

- .NET 6.0 - <https://dotnet.microsoft.com/en-us/download/dotnet/6.0>
- Java (version 17 recommended) - <https://www.oracle.com/java/technologies/downloads/#java17>

Also highly recommended:

- Git - <https://git-scm.com>

0.4.2 F# Programming Language

The hygge compiler is written in F#, and the lectures will often discuss how something can be implemented in F#. Therefore, it is highly recommended to have (or acquire) some experience with F#. Here are some useful resources.

- F# home page: <https://dotnet.microsoft.com/en-us/languages/fsharp>
- First steps with F#: <https://learn.microsoft.com/en-gb/training/paths/fsharp-first-steps>

Some specific F# features will be used very often:

- **Discriminated union types:** <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions>
- **Records:** <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/records>
- **Pattern matching:** <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching>
- **Mapping over lists:** <https://bradcollins.com/2015/04/17/f-friday-the-map-function>
- **Folding over lists:** [https://riptutorial.com/fsharp/example/7373/intro-to-folds-with-a-handful-of-examples²](https://riptutorial.com/fsharp/example/7373/intro-to-folds-with-a-handful-of-examples<sup>2</sup)

0.5 Assessment: Group Project and Oral Group Examination

This is a **project-based course with work in groups (normally 5 people)**.

Your goal is to design and implement features that are not initially supported by the Hygge language and compiler provided with the course materials. For the assessment, your group will need to submit:

1. the **source code of the improved compiler** where you implemented such features – including the test cases, and
2. a *Project Report* describing how you designed and implemented each feature.

The course ends with an *Oral Group Examination*.

You can choose between two possible project routes: the *Standard Project Route* or the *Custom Project Route*.

0.5.1 Standard Project Route

You select some of the “**Project Ideas**” presented throughout the course, and realise them by extending the Hygge language and hygge compiler provided as part of the course.

You don’t need to implement all the Project Ideas presented during the course: whenever a module presents a list of “Project Ideas” it will also specify how many of them you should select for your project – and if you want to do more than that, you are welcome! You can select the Project Ideas you like, without need to request the teacher’s approval.

² <https://riptutorial.com/fsharp/example/7373/intro-to-folds--with-a-handful-of-examples>

0.5.2 Custom Project Route

Besides the standard Project Ideas discussed above, your group can propose **variations**, and **new and alternative** project ideas. Your proposals are welcome! You'll need to **talk about your proposal with the teacher** and find an agreement before proceeding.

As an extreme, your group may even propose a radically different project — e.g. writing a new compiler from scratch, or start from another language and/or compiler, instead of the ones provided for the course. Note, however, that this project path is **more risky**: it may require a much higher implementation effort, and it may clash with the course timeline; moreover, the teacher and TAs may not be able to help you in case of difficulties. If you *really* want to pursue this possibility, **talk about it with the teacher** and find an agreement before proceeding.

0.5.3 Project Report

To write the project report, you should use the LaTeX template available on DTU Learn, which contains some examples and guidelines. The **individual contributions of each group member** must be explained in the report, because **the course grades are individual**.

A few recommendations:

- when you show formal rules (for the syntax, semantics, type checking...) then you may not need a long explanation (the rules may be enough);
- when showing code snippets, try to reduce them to the bare minimum that helps you explain what you did. Instead of including long code snippets, you can refer to the relevant files and functions in your project source code.

0.5.4 Oral Group Examination

The course ends with an oral group examination, with the following structure.

1. The oral exam will open with a **5-minutes group presentation** of your project: you should provide a brief overview of your work, and highlight the main features, challenges, and anything you deem noteworthy. When preparing the presentation, consider that the teacher and external examiner will have already read your project report: besides the overview and highlights, you can refer to it for the technical details.
2. Then, the exam will continue with **questions to each group member**. The main objective is to assess your contribution to the project work: e.g. you may be asked to clarify some aspect of your work, or discuss possible alternative approaches to what you did. You will *not* be asked detailed technical questions about parts of the project that were implemented by other team members. You may be asked general questions about compiler concepts (e.g. what are the main phases of a compiler, what is subtyping...).

0.6 Group Work Organisation: Some Suggestions

To succeed in your project, you will need to organise your group work. When working on a compiler, there are two common (and somewhat opposite) approaches.

1. **Phase-oriented:** each group member specialises in a compiler phase (parsing, type checking, ...). When a new feature needs to be added to the compiler, the group members contribute by taking care of the phase where they are specialised.
 - **Pros:** easier organisation, because every group member tends to work on different files in the compiler source tree.
 - **Cons:** individual group members may not learn much outside the compiler phase where they are specialised.
2. **Feature-oriented:** when a new feature needs to be added to the compiler, a subset of the group (one or a few members) takes charge of it, and implements it across all compiler phases (from lexing to code generation).
 - **Pros:** every group member tends to work through all the compiler phases, and learns something about all of them.
 - **Cons:** requires more organisation, because multiple group members (working on different features) may need to modify at the same time the same files in the compiler source tree.

As a group, you can adopt the approach you prefer: one of the above, or some hybrid between the two, or some variation, or something else entirely... It's your choice — and you don't need the teacher's approval to decide. In any case, **the individual contributions of each group member must be explained in the project report.**

Note: If you choose approach 2 (feature-oriented), you may adopt something like the **Git Feature Branch Workflow** to make your group coordination smoother:

- <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>

The link above mentions Bitbucket, but you can realise the workflow in the same way when using the [DTU Compute GitLab service](https://lab.compute.dtu.dk)³, since it supports [merge requests](https://docs.gitlab.com/ee/user/project/merge_requests)⁴.

Warning: Be aware that, if you choose approach 1 (phase-oriented), **the lexing and parsing phases have less weight in the course assessment** with respect to the other phases: therefore, as a group member, you may not want to only work on those phases. The lexing/parsing assessment weight may increase if you propose some related Project Idea (to be agreed with the teacher, see [Custom Project Route](#)).

³ <https://lab.compute.dtu.dk>

⁴ https://docs.gitlab.com/ee/user/project/merge_requests

Module 1: Introduction to RISC-V

In this module, you will learn the basics of assembly programming in RISC-V. This will be necessary to understand the code generated by `hyggec`, and to implement code generation for new Hygge programming language constructs.

1.1 What is RISC-V? And Why Is It Relevant?

RISC-V is a modern **Open Source Instruction Set Architecture (ISA)** with an explosive growth in popularity and adoption across all areas — hobbyists, academia, industry.

The first letters of “RISC-V” stand for **Reduced Instruction-Set Computer**: a CPU design philosophy where processors have few instructions, but run them very efficiently. As a comparison: the basic RISC-V ISA consists of 47 different instructions, whereas the x86 ISA consists of many hundreds of instructions.

RISC-V has a modular design: its base ISA has a rather limited set of registers and instructions, but various extensions expand the architecture capabilities by adding more registers and instructions. Consequently, the RISC-V base ISA only supports very simple integer arithmetic — and there is an extension that adds support for integer division and multiplication, another extension for floating-point arithmetic, another for vector arithmetic... With this design, the RISC-V architecture can scale from very small, low-cost and low-power microcontrollers, to powerful multicore CPUs with built-in hardware acceleration for numerical processing.

In this course we will use a combination of RISC-V extensions denoted **RV32IMF**:

- **RV32I** is the base instruction set with 32-bit registers and operations;
- **M** is the extension that adds integer division and multiplication operations;
- **F** is the extension that adds single-precision, 32-bit floating-point registers and operations.

1.2 Base and Floating-Point Registers

Table 1.1 and Table 1.2 list, respectively, the 32 integer registers available in the base 32-bit RISC-V ISA, and the additional 32 registers introduced by the single-precision floating-point extension. Each register has a size of 32 bits.

Table 1.1: 32-bit RISC-V base integer registers.

Base name	register	Symbolic name	Description	Saved by
x0		zero	Hard-wired zero	—
x1		ra	Return address	Caller
x2		sp	Stack pointer	Callee
x3		gp	Global pointer	—
x4		tp	Thread pointer	—
x5		t0	Temporary / alternate link register	Caller
x6 – x7		t1 – t2	Temporaries	Caller
x8		s0 / fp	Saved register / frame pointer	Callee
x9		s1	Saved register	Callee
x10 – x11		a0 – a1	Function arguments / return values	Caller
x12 – x17		a2 – a7	Function arguments	Caller
x18 – x27		s2 – s11	Saved registers	Callee
x28 – x31		t3 – t6	Temporaries	Caller

Table 1.2: RISC-V single-precision floating-point registers.

Floating-point name	register	Symbolic name	Description	Saved by
f0 – f7		ft0 – ft7	Floating-point temporaries	Caller
f8 – f9		fs0 – fs1	Floating-point saved registers	Callee
f10 – f11		fa0 – fa1	Floating-point arguments/return values	Caller
f12 – f17		fa2 – fa7	Floating-point arguments	Caller
f18 – f27		fs2 – fs11	Floating-point saved registers	Callee
f28 – f31		ft8 – ft11	Floating-point temporaries	Caller

The register names are x0...x31 (for integer registers) and f0...f31 (for floating-point registers). Their use is unrestricted: a program can write and read registers for any purpose – with one exception: register x0 is immutable and always contains the value 0.

Each register also has a **symbolic name** that reflects its *conventional* use. For example:

- register x0 is also called zero;
- register x6 is also called t1, because it can be used to hold a temporary value that may be later discarded;

- register `x10` is also called `a0`, because it should be used to pass an argument when calling a function; it should also be used to hold the function's return value.

The conventional use of RISC-V registers is usually observed by compiler developers, to ensure that the RISC-V code generated by one compiler can interoperate with code generated by other compilers.

There is also another important register: the *program counter* `pc`, which always contains the memory address of the instruction being executed by the CPU. Unlike the registers listed above, the content of `pc` cannot be read or written directly: it is only retrieved or updated by instructions that control the program execution — such as jumps (see *A Few RISC-V Assembly Instructions* below).

Note: For now, we ignore the “Saved by” column in [Table 1.1](#) and [Table 1.2](#). We will reprise it later in the course.

1.3 A Few RISC-V Assembly Instructions

We will not address all RISC-V assembly instructions in detail. Instead, we practice with a few of them (listed in the following subsections) to write some RISC-V assembly programs. This experience will be helpful to explore the rest of the RISC-V ISA and learn how other instructions work.

A few remarks:

- a **word** in RISC-V is **32 bits** (4 bytes) in size;
- all memory accesses must be **32-bit aligned** (i.e. any memory address used to read/write data or execute code must be a multiple of 32);
- a **label** represents a memory address in RISC-V assembly (this will be clearer when we will discuss the *RISC-V Assembly Program Structure*).

Attention: Some of the assembly instructions below are marked as **pseudo instructions**: this means that they are not implemented in hardware. Instead, they are made available (as a convenience) by most **assemblers** — which are programs that transform RISC-V assembly code into actual RISC-V binary machine code. Therefore, a RISC-V pseudo instruction may be translated by the assembler into multiple RISC-V machine instructions.

The distinction between RISC-V machine instructions and pseudo instructions will not be very relevant for this course — but you may notice that:

- when reading RISC-V documentation, some pseudo instructions may be different or absent; and
- when running or debugging RISC-V assembly code using *RARS — RISC-V Assembler and Runtime Simulator* (or other similar tools), the pseudo instructions

are expanded into the corresponding machine instructions.

1.3.1 Load and Store Instructions

These instructions load data from memory into a register, copy data between registers, or store data from a register into memory.

Syntax	Name	Description
<code>li rd, val</code>	Load immediate	Load into register <code>rd</code> the 32-bit value <code>val</code> . (Pseudo instruction)
<code>lw rd, label</code>	Load word	Load into register <code>rd</code> the word stored at memory address <code>label</code> . (Pseudo instruction)
<code>la rd, label</code>	Load absolute	Load into register <code>rd</code> the memory address <code>label</code> . (Pseudo instruction)
<code>mv rd, rs</code>	Move	Move (i.e. copy) the content of register <code>rs</code> into register <code>rd</code> .
<code>sw rs2, offset(rs1)</code>	Store word	Store the 32-bit value contained in the register <code>rs2</code> into memory. The destination memory address is computed adding the value <code>offset</code> to the content of register <code>rs1</code> .

1.3.2 Integer Arithmetic Instructions

These instructions operate on base integer registers.

Syntax	Name	Description
<code>add rd, rs1, rs2</code>	Addition	Add the contents of registers <code>rs1</code> and <code>rs2</code> and store the result in register <code>rd</code> .
<code>sub rd, rs1, rs2</code>	Subtraction	Subtract the contents of register <code>rs2</code> from <code>rs1</code> and store the result in register <code>rd</code> .
<code>mul rd, rs1, rs2</code>	Multiplication	Multiply the contents of registers <code>rs2</code> and <code>rs1</code> and store the result in register <code>rd</code> .
<code>div rd, rs1, rs2</code>	Division	Divide the content of register <code>rs1</code> by <code>rs2</code> and store the result in register <code>rd</code> .

1.3.3 Control Transfer Instructions

These instructions perform jumps, with or without conditions.

Syntax	Name	Description
<code>j label</code>	Jump	Jump to memory address <code>label</code> and execute the code stored there. (Pseudo instruction)
<code>beq rs1, rs2, label</code>	Branch if equal	Compare the contents of registers <code>rs1</code> and <code>rs2</code> , and jump to <code>label</code> if they are equal.
<code>bne rs1, rs2, label</code>	Branch if not equal	Compare the contents of registers <code>rs1</code> and <code>rs2</code> , and jump to <code>label</code> if they are not equal.
<code>blt rs1, rs2, label</code>	Branch if less than	Compare the contents of registers <code>rs1</code> and <code>rs2</code> , and jump to <code>label</code> if the content of <code>rs1</code> is smaller than the content of <code>rs2</code> .

1.3.4 Single-Precision Floating-Point Instructions

These instructions include numerical operations between floating-point registers. There are also instructions to transfer data between registers (`fmv.w.x`, `fmv.s`), and to compare the contents of floating-point registers (`feq.s`, `flt.s`, `fle.s`).

Syntax	Name	Description
<code>fmv.w.x rd, rs</code>	Integer to floating-point register move	Move (copy) the content of <i>integer</i> register <code>rs</code> into floating-point register <code>rd</code> . The content of <code>rs</code> is expected to be a single-precision floating-point value in IEEE 754-2008 ⁵ encoding.
<code>fmv.s rd, rs</code>	Floating-point to floating-point register move	Move (copy) the content of floating-point register <code>rs</code> into floating-point register <code>rd</code> .
<code>fadd.s rd, rs1, rs2</code>	Floating-point addition	Add the contents of floating-point registers <code>rs1</code> and <code>rs2</code> and write the result in floating-point register <code>rd</code> .

continues on next page

Table 1.6 – continued from previous page

Syntax	Name	Description
<code>fsub.s rd, rs1, rs2</code>	Floating-point subtraction	Subtract the contents of floating-point register <code>rs2</code> from floating-point register <code>rs1</code> and write the result in floating-point register <code>rd</code> .
<code>fmul.s rd, rs1, rs2</code>	Floating-point multiplication	Multiply the contents of floating-point registers <code>rs2</code> and <code>rs1</code> and write the result in floating-point register <code>rd</code> .
<code>fdiv.s rd, rs1, rs2</code>	Floating-point division	Divide the content of floating-point register <code>rs1</code> by floating-point register <code>rs2</code> and write the result in floating-point register <code>rd</code> .
<code>feq.s rd, rs1, rs2</code>	Floating-point equality comparison	Check whether the contents of floating-point registers <code>rs1</code> and <code>rs2</code> are equal, and write the result of the check into the <i>integer</i> register <code>rd</code> : write 1 if the check is true, and 0 otherwise.
<code>flt.s rd, rs1, rs2</code>	Floating-point less-than comparison	Check whether the content of floating-point register <code>rs1</code> is less than the content of floating-point register <code>rs2</code> , and write the result of the check into the <i>integer</i> register <code>rd</code> : write 1 if the check is true, and 0 otherwise.
<code>fle.s rd, rs1, rs2</code>	Floating-point less-or-equal comparison	Check whether the content of floating-point register <code>rs1</code> is less than or equal to the content of floating-point register <code>rs2</code> , and write the result of the check into the <i>integer</i> register <code>rd</code> : write 1 if the check is true, and 0 otherwise.

⁵ <https://doi.org/10.1109/IEEESTD.2008.4610935>

1.3.5 System Instructions

These instructions allow a RISC-V assembly program to interact with the surrounding operating system.

Syntax	Name	Description
<code>ebreak</code>	Environment break	Stop the execution. This instruction acts as a breakpoint, and is used e.g. to let debuggers take control of a running program.
<code>ecall</code>	Environment call	Perform a system call. This will become clearer in when we will discuss the <i>RISC-V Assembly Program Structure</i> and <i>RARS – RISC-V Assembler and Runtime Simulator</i> .

1.4 RISC-V Assembly Program Structure

Example 1 shows simple RISC-V assembly program.

Example 1 (A simple RISC-V assembly program)

```

1  # A simple program that adds two integers (one stored in memory, the other
2  # immediate), stores the result in memory, and exits.
3
4  .data          # The next items are stored in the Data memory segment
5  value:        # Label for the memory address of the value below
6      .word 3    # Allocate a word (size: 4 bytes) and initialise it to value 3
7  result:       # Label for the memory address of the value below
8      .word 0    # Allocate a word (size: 4 bytes) and initialise it to value 0
9
10 .text         # The next items are stored in the Text memory segment
11  lw t0, value  # Load word at the memory address 'value' in register t0
12  li t1, 42     # Load the immediate value 42 in register t1
13  add t2, t0, t1 # Add contents of t0 and t1, store result in t2
14  la t3, result # Load the memory address of label 'result' in t3
15  sw t2, 0(t3)  # Store word in t2 in memory address in t3 (offset 0)
16
17  li a7, 10     # Load the immediate value 10 in register a7
18  ecall        # Perform syscall. In RARS, if a7 is 10, this means: "Exit"

```

When a program runs on a RISC-V architecture, its memory is divided into **segments**. The two principal segments are:

- **Text segment**, which contains the program's machine code executed by the CPU (therefore, the pc register should always contain a memory address within this segment);
- **Data segment**, which contains data used by the program.

The RISC-V assembly program in *Example 1* uses the `.data` and `.text` directives (respectively on lines 4 and 10) to place contents in the corresponding segments: such contents can be either values (lines 5–8) or machine code generated from assembly instructions (lines 11–18).

Moreover, the RISC-V assembly program in *Example 1* uses **labels** to represent memory addresses. For example, line 5 says that the label called `value` is an alias for the memory address of the content defined on line 6. Then, the program uses the label `value` to access that content and load it into a register (line 11). When the assembly program is given to an **assembler** to generate the corresponding RISC-V machine code, each label is replaced by an actual, 32-bit-aligned memory address.

The last two lines of the RISC-V assembly program in *Example 1* perform a **system call** that exits from the running program: we will see the precise meaning shortly, when discussing *RARS – RISC-V Assembler and Runtime Simulator*.

Example 2 shows another RISC-V assembly program, featuring a loop (based on conditional jumps) and the use of floating-point values and operations.

Example 2 (A RISC-V assembly program with floats and a loop)

```
1 # A program that increments a single-precision floating-point value
2 # (starting from 1.0) by adding 10 times the value 0.1. Before each
3 # increment, the program prints on the console a message reporting the
4 # current value. Then, the program exits.
5
6 .data          # The next items are stored in the Data memory segment
7 msg:          # Label for the mem addr of the first char of the string below
8   .string "The current value is: " # Allocate a string, in C-style: a
9                                     # sequence of characters in adjacent
10                                    # memory addresses, terminated with 0
11
12 .text         # The next items are stored in the Text memory segment
13   li t0, 0x3f800000 # Load this immediate value into register t0
14                       # The value above is the 32-bit hexadecimal representation of
15                       # the single-precision floating-point number 1.0.
16                       # To convert values between floating-point and hex, see e.g.:
17                       # https://www.h-schmidt.net/FloatConverter/IEEE754.html
18   fmv.w.x ft0, t0 # Move the content of t0 into floating-point reg ft0
19                       # Register ft0 contains the value we will increment
20
21   li t0, 0x3dcccccd # Load this immediate value into register t0
22                       # The value above is the 32-bit hexadecimal representation of
23                       # the single-precision floating-point number 0.1
24   fmv.w.x ft1, t0 # Move the content of t0 into floating-point reg ft1
```

(continues on next page)

(continued from previous page)

```

25         # Register ft1 contains the increment we will add to ft0
26
27     li t0, 0 # Load value 0 into register t0 (used as counter)
28     li t1, 1 # Load value 1 into register t1 (used as counter increment)
29     li t2, 10 # Load value 10 into register t2 (number of increments)
30
31 loop_begin: # Label for memory location of the beginning of the loop
32     la a0, msg # Load address of label 'msg' into a0, for printing below
33     li a7, 4 # Load immediate value 4 into register a7
34     ecall # Syscall. In RARS, if a7=4, this means: "PrintString"
35
36     li a7, 2 # Load value 2 into register a7
37     fmv.s fa0, ft0 # Copy float value in ft0 into fa0, for printing below
38     ecall # Syscall. In RARS, if a7=2, this means: "PrintFloat"
39
40     li a0, '\n' # Load value of char '\n' into a0, for printing below
41     li a7, 11 # Load immediate value 11 into register a7
42     ecall # Syscall. In RARS, if a7=11, this means: "PrintChar"
43
44     beq t0, t2, loop_end # If t0 and t2 are equal, jump to loop_end
45
46     fadd.s ft0, ft0, ft1 # Increment the floating-point value: add the
47                          # contents of floating point registers ft0 and
48                          # ft1, write the result in ft0
49
50     add t0, t0, t1 # Increment loop counter: add t0 and t1, result in t0
51
52     j loop_begin # Jump to the beginning of the loop
53
54 loop_end: # Label for memory location of the end of the loop
55     li a7, 10 # Load the immediate value 10 in register a7
56     ecall # Perform syscall. In RARS, if a7 is 10, this means: "Exit"

```

Example 2 highlights some more characteristics of RISC-V assembly programming:

- we can store strings in memory (lines 7 and 8) and then access them (line 32);
- to load an immediate floating-point value into a floating-point register, we first load its “raw” byte representation into a base register (lines 13, 21) and then copy the value into a floating-point register (lines 18, 24);
- we use system calls for printing various types of data (lines 34, 38, 42): they are made available by *RARS – RISC-V Assembler and Runtime Simulator*.

1.5 RARS — RISC-V Assembler and Runtime Simulator

To run a RISC-V assembly program, we need to:

1. process the assembly program with an **assembler** that translates it into the corresponding RISC-V binary machine code; and
2. execute the RISC-V binary machine code using either **real RISC-V hardware**, or a **RISC-V emulator**.

In this course we will use the RISC-V assembler and emulator **RARS**, which implements both functionalities above, and includes very useful features for debugging RISC-V assembly programs.

1.5.1 Downloading and Running RARS

RARS is available at:

- <https://github.com/TheThirdOne/rars>

The instructions below are based on the RARS “continuous” build released on 28 June 2022:

- <https://github.com/TheThirdOne/rars/releases/tag/continuous>

To see RARS in action, you can follow these steps.

1. Download the file `rars_27a7c1f.jar` from the link above.
2. Launch RARS from a terminal: `java -jar rars_27a7c1f.jar`
3. On the main RARS program window, click on the menu “File” → “New”.
4. Copy & paste the code of *Example 1* in the “Edit” area.
5. Save the code being edited (this step is necessary to proceed): “File” → “Save as...”.
6. Assemble the RISC-V assembly code, generating RISC-V machine code: click on the menu “Run” → “Assemble”

The main area of the RARS program window will now switch from the “Edit” to the “Execute” view. You should now see:

- the **Text memory segment** of the running program:
 - the “Source” column shows each instruction in your RISC-V assembly code
 - the “Basic” column shows the corresponding RISC-V machine instructions (you may see how pseudo instructions are expanded)
 - the “Code” column shows the corresponding binary machine code
 - the “Address” column shows the memory address of each machine instruction
 - the “Bkpt” column can be used to place a breakpoint (e.g. for debugging)

- the **Data memory segment** of the running program
- the **RISC-V registers** (base and floating-point)
- a **“Run I/O” console** with program execution information, and the program input/output

Tip: Before proceeding, you can make the register contents easier to read: click on the menu “Settings” and *deselect* the option “Values displayed as hexadecimal”.

You can now execute your program: if you hover with your mouse cursor on the icons in the toolbar, a pop-up will show their functionality. Note, in particular, that you can:

- **run the current program** until it terminates, and **pause** or **stop** its execution. You can use the “Run I/O” console to:
 - see the running program output, and its termination status;
 - provide inputs to the running program;
- perform a **single execution step**: RARS will highlight the current instruction, and which register or memory location have been modified;
- **undo the program execution, one step at a time** (very useful for debugging);
- **reset the memory and registers**, thus restarting the program execution.

Tip: The RARS documentation is available on its Wiki:

- <https://github.com/TheThirdOne/rars/wiki>

You can also get a very handy quick reference help by clicking on the RARS menu: “Help” → “Help”.

1.5.2 RARS System Calls

Besides emulating a RISC-V CPU, RARS also simulates some bits of an operating system — and RISC-V assembly programs can interact with this mini-OS by performing **system calls** (a.k.a. **syscalls**) using the instruction `ecall`. This allows the running program to access various services — e.g. read inputs from the “Run I/O” console, produce outputs, read or write files, terminate execution, even play MIDI music (!)...

To perform a system call, a RISC-V assembly program needs to:

- load the desired syscall number into register `a7`;
- load the syscall arguments (if any) into other registers (depending on which syscall is selected in `a7`);
- perform the syscall, with the instruction `ecall`;

- after the syscall returns, some registers (depending on which syscall is selected in a7) may be updated with its result.

The RARS syscalls are documented here:

- <https://github.com/TheThirdOne/rars/wiki/Environment-Calls>

They are also listed in the quick reference help, available by clicking on the RARS menu: “Help” → “Help”.

1.6 References and Further Readings

The official RISC-V specification documents all the details of the ISA. It also documents ISA extensions, including floating-point.

- RISC-V ISA Specification (ratified), Volume 1 (Unprivileged spec). Available at: <https://riscv.org/technical/specifications>

Note: The main intended audience of the RISC-V ISA specification are CPU designers and implementers; many of its details (e.g. how RISC-V instructions are encoded in bits) are not crucial for RISC-V assembly programming.

A very useful reference for RISC-V assembly programming is provided by the Shakti initiative at IIT-Madras (India), which develops RISC-V CPUs and products.

- Shakti ASM manual. Available at: <http://shakti.org.in/documentation.html>
 - See, in particular, chapters 1, 2, 4, 5.

Note: As of January 2023, the Shakti ASM manual does not cover floating-point instructions (but this might change in later editions).

There are also several quick reference cards for RISC-V assembly, which summarise the ISA in a few A4 pages. For example:

- James Zhu’s RISC-V Reference Card. Available at: <https://github.com/jameslzhou/riscv-card>

There are also other RISC-V emulators, besides RARS. Some of them run in a browser, and do not require any software installation: they can be quite handy for experiments, but their features can be quite limited and sometimes incompatible with RARS. For example:

- Keyhan Vakil’s Venus RISC-V simulator. Available at: <https://venus.kvakil.me/>

Important: Besides the links above, you can find more RISC-V documentation, tutorials, and tools on the Web — and new materials are published very frequently. If you find any other good resource that you would recommend, you are welcome to share it on Piazza!

1.7 Lab Exercises

Note: The following exercises are *not* assessed: their purpose is to practice with RISC-V assembly programming, and become familiar with RARS. This experience will be useful when dealing with code generation in the rest of the course.

You are welcome (and encouraged!) to discuss the exercises and your solutions with your fellow students – either in person, or on Piazza.

You should be able to solve all exercises by only using the RISC-V instructions listed in *A Few RISC-V Assembly Instructions* – but feel free to browse the *References and Further Readings* and experiment with other instructions.

Exercise 1 (Minimising register usage)

Adjust the RISC-V assembly code in *Example 1* so that it computes the same result by only using registers t0 and t1 (plus register a7 for the final syscall).

Hint: The source and destination registers of RISC-V instructions can overlap. For instance, see the code in *Example 2*, line 50...

Exercise 2 (Adding integers read from the console)

Write a RISC-V assembly program that reads two integer values from the console, computes their sum, and prints it on screen.

Hint: You will need to use the ReadInt and PrintInt RARS syscalls: see the [documentation](#)⁶.

Consider that, after the ReadInt syscall, the integer value read from the console is available in register a0. If you call ReadInt again, a0 is overwritten with the new console input...

Exercise 3 (Adding floats read from the console)

Write a RISC-V assembly program that reads two float values from the console, computes their sum, and prints it on screen.

Hint: The solution is similar to *Exercise 2* – except that you will need to use the

⁶ <https://github.com/TheThirdOne/rars/wiki/Environment-Calls>

ReadFloat and PrintFloat RARS syscall: see the [documentation](#)⁷.

Exercise 4 (Comparing integers)

Write a RISC-V assembly program that reads two integer values from the console, and prints on the console a message saying whether the two values are equal, or the first is greater than the second, or *vice versa*.

Hint: See the hints of [Exercise 2](#). Moreover:

- To print a message on the console, you will need to use the PrintString RARS syscall, as in [Example 2](#).
 - To print different messages depending on which value is greater, you will need to use conditional branch instructions.
-
-

Exercise 5 (Comparing floats)

Write a RISC-V assembly program that reads two single-precision floating-point values from the console, and prints on the console a message saying whether the two values are equal, or the first is greater than the second, or *vice versa*.

Hint: You can use the solution of [Exercise 2](#) as a starting point. However:

- To print a float on the console, you will need to use the PrintFloat RARS syscall, as in [Example 2](#).
 - To print different messages depending on which value is greater, you will need to use *both* floating-point comparison instructions *and* conditional branch instructions. Therefore, the resulting code can be quite different from the solution of [Exercise 2](#)...
-
-

Exercise 6 (Factorial)

Write a RISC-V assembly program that reads an integer n from the console, and checks whether it is positive. If n is positive, the program computes and prints on the console the factorial $n!$, defined as:

$$n! = n \times (n - 1) \times \dots \times 2 \times 1$$

⁷ <https://github.com/TheThirdOne/rars/wiki/Environment-Calls>

Exercise 7 (Sum of n floats read from the console)

Write a RISC-V assembly program that reads an integer n from the console, and checks whether it is positive. If n is positive, the program reads n single-precision floating-point values from the console, and prints their sum on the console.

Exercise 8 (Approximation of Pi)

Write a RISC-V assembly program that reads an integer n from the console, and checks whether it is positive. If n is positive, the program computes and prints on the console the approximation of π calculated using the Taylor expansion up to the n th term:

$$\pi \approx 4 \left(\sum_{i=0}^{n-1} \frac{(-1)^i}{2i+1} \right) = 4 \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + \frac{(-1)^n}{2n+1} \right)$$

Module 2: The Hygge0 Language Specification

This module explains the formal syntax, semantics, and typing system of the Hygge0 programming language. Besides the focus on Hygge0, the broader objective of this module is to learn (or revise) some key concepts of programming language theory: grammars, syntax trees, structural semantics, type checking. These concepts provide us with the foundations for implementing a compiler – and we will see that they are very recognisable in the `hygdec` compiler internals. Moreover, programming languages theory gives us the tools to reason on how a program is expected to run, and whether a programming language is correctly designed.

Example 3 (A Sample of Hygge0)

Here is an example of Hygge0 program: our goal in this module is to specify how a program like this can be checked for syntactic correctness, how it should behave when executed, and how it can be type-checked to avoid many forms of runtime error.

```
1 let x: int = 1; // Variable declaration
2
3 type MyInt = int; // Type declaration
4
5 let y: MyInt = {
6     println("Initialising y");
7     2: int // Type ascription
8 };
9
10 if x < y then println("x is smaller than y")
11     else println("x is not smaller than y");
12
13 print("The result of x + y is: ");
14 println(x + y);
15 assert(x + y < 42) // Assertion
```

Note: Unlike Hygge0, most programming languages do not have a formal specification: their intended behaviour is only explained using (many pages of) English prose.

A noteworthy exception is WebAssembly: it includes an [extensive formal specification](#)⁸, and this serves as an inspiration for the specification of Hygge0 (although the specification of WebAssembly is *much* more complex!).

2.1 Formal Syntax of Hygge0

Definition 1 below specifies the syntax the Hygge0 programming language, as a **context-free grammar**. If we take a sequence of symbols (e.g. characters read from a text file), the grammar below determines whether that sequence represents a syntactically-valid Hygge0 *expression*. A Hygge0 program consists of just one expression, possibly containing many sub-expressions.

Definition 1 (Formal Grammar of Hygge0)

We define an **identifier** as any sequence of characters that:

1. is non-empty,
2. contains letters, numbers, or the character `_`, and
3. does *not* begin with a number.

The grammar for Hygge0 expressions e and values v below uses some identifiers to denote specific operations and values (e.g. “and”, “print”, “assert”, “true”...): those identi-

⁸ <https://webassembly.github.io/spec/core/>

fiers are considered **reserved**.

Expression	$e ::=$	type $x = t; e$	(Declare x as alias of t in scope e)
		let $x : t = e_1; e_2$	(Declare x as e_1 in scope e_2)
		$e_1; e_2$	(Sequencing)
		$\{ e \}$	(Expression in curly brackets)
		if e_1 then e_2 else e_3	(Conditional)
		e_1 or e_2	(Logical "or")
		e_1 and e_2	(Logical "and")
		$e_1 = e_2$	(Relation: equality)
		$e_1 < e_2$	(Relation: less than)
		$e_1 + e_2$	(Addition)
		$e_1 * e_2$	(Multiplication)
		not e	(Logical negation)
		readInt()	(Read integer from console)
		readFloat()	(Read float from console)
		print(e)	(Print on console)
		println(e)	(Print on console with newline)
		assert(e)	(Assertion)
		$e : t$	(Type ascription)
		(e)	(Expression in parentheses)
		x	(Variable)
		v	(Value)
Value	$v ::=$	1 2 3 ...	(Integers)
		true false	(Booleans)
		"Hello" "Hej" ...	(Strings)
		3.14f 42.0f ...	(Single-precision float values)
		()	(Unit value)
Variable	$x ::=$	z foo a123 ...	(Any non-reserved identifier)
Pretype	$t ::=$	int unit foo ...	(Any non-reserved identifier)

The notation in [Definition 1](#) is a grammar in **Backus-Naur form**. The items on the left of “ $::=$ ” are **syntactic categories** (expression e , value v ...) while on the right of “ $::=$ ” are **grammar rules** (also called **production rules**, or simply **productions**). The notation means: given a sequence of input symbols (e.g. characters), the grammar classifies the input in one of the categories on the left of $::=$ *if and only if* that sequence matches one of the rules on the right.

The syntax of Hygge0 includes many familiar expressions and operations (addition, multiplication, comparisons...).

Notice that the grammar rules in [Definition 1](#) are recursive: for example, in order to classify a sequence of symbols like $2 + (3 * y)$ as a valid Hygge0 expression, we can only use the rule for addition – and that rule, in turn, requires us to check that both sub-sequences 2 and $(3 * y)$ are valid expressions, by recursively checking them against the same set of rules. This recursive checking can terminate in two possible ways:

1. we reach a sub-sequence of input symbols on which no rule can be applied, hence the input sequence is rejected; or
2. we reach and accept **terminal** symbols that do not have any further sub-component to check (such as variable y , or value 3).

2.1.1 Syntax Trees

When the grammar rules in *Definition 1* classify a sequence of symbols as an expression e , we can construct a **syntax tree** based on which rules have been applied, thus capturing the syntactic structure e : see *Example 4*.

Example 4 (A Simple Hygge0 Expression)

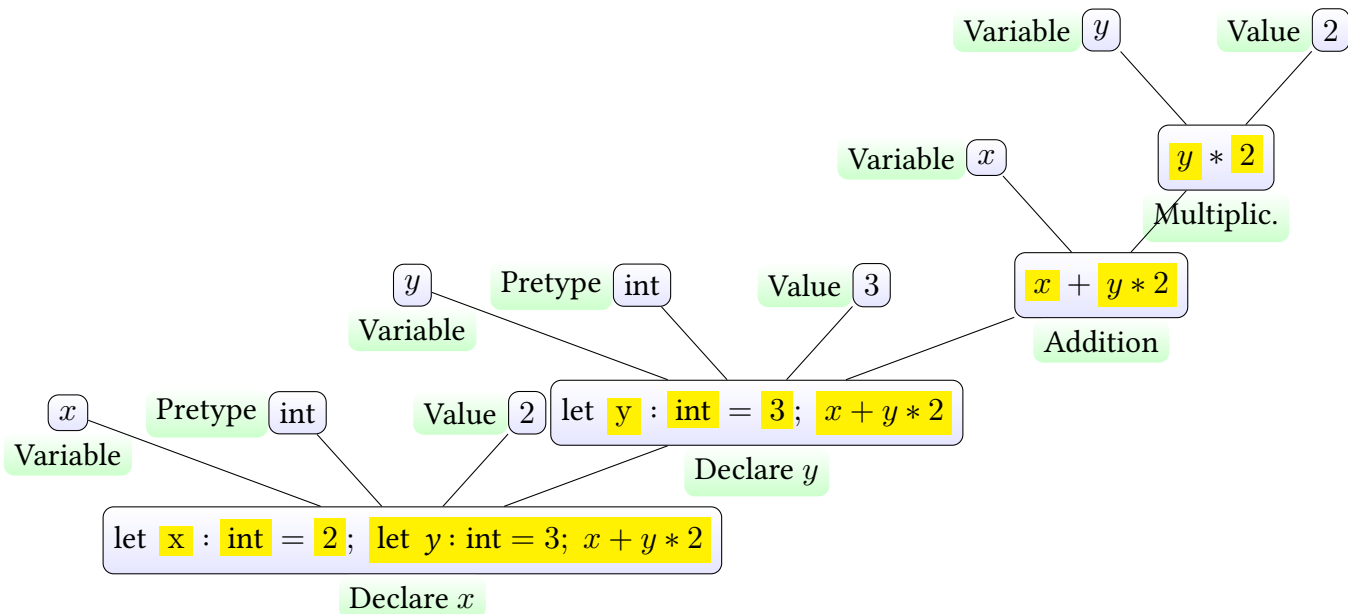
The grammar in *Definition 1* accepts the following sequence of symbols, and classifies it as a syntactically-valid Hygge0 expression:

```
let  $x$  : int = 2;  
let  $y$  : int = 3;  
 $x + y * 2$ 
```

The grammar can accept the expression above by applying its rules in the order depicted in the syntax tree below, where:

- each node contains a Hygge0 expression, value, variable, or pretype;
- the edges (going up) connect an expression to its immediate syntactic sub-components (if any);
- the labels describe which grammar rule in *Definition 1* accepts the nearby node;
- the root of the tree is the whole expression being accepted, and
- the leaves of the trees are terminal symbols in the grammar, which don't have any further sub-component to check.

For clarity, the depiction below highlights all immediate syntactic sub-components of each node.



Example 5 (Something That is Not a Syntactically-Valid Hygge0 Expression)

The Hygge0 grammar in *Definition 1* does *not* accept the following sequence of symbols:

```
let x : int = 2;
let y : int = 3;
x + y *
```

The reason is that there is no rule in *Definition 1* which can accept a `*` symbol that does not have a sub-expression on the right. Therefore, the sequence of symbols above does not constitute a syntactically-valid Hygge0 expression, and we cannot build a syntax tree for it.

The Hygge0 grammar in *Definition 1* is simple but flexible: its main syntactic category are expressions, which can be freely composed and nested to create larger expressions — and a Hygge0 program is just one (possibly large and complex) expression. This design is inspired by functional languages like F#, Scala, or Haskell.

Example 6 (Hygge0 Grammar vs. C or Java)

Consider the following Hygge0 expression, that initialises variable `x` with a value computed by nesting other expressions within curly brackets:

```
let x : int = {
  let y : int = if (2 < 42) then 0 else 42;
  y + 1
};
println(x)
```

This cannot be written in languages like C or Java: their grammar distinguishes expressions from *statements*, and their grammar rules do not allow statements (which include

e.g. if ... then ... else ... and variable declarations) to appear inside expressions.

Exercise 9 (Drawing a Syntax Tree)

Draw the syntax tree of the Hygge0 expression in *Example 6*.

Example 7 (Another Simple Hygge0 Expression)

The *Definition 1* also accepts the following expression:

```
let x : foo = 2;  
y + x * "Hello"
```

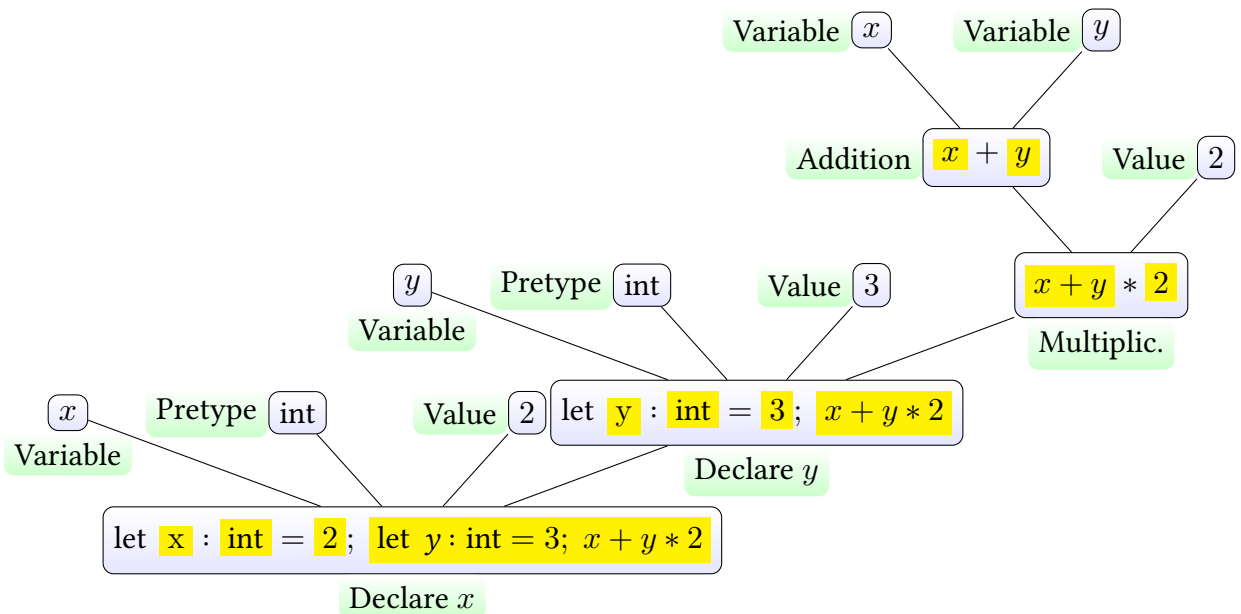
The expression is syntactically correct, but meaningless: `foo` is not a valid type, `y` is an undefined variable, and (as we will see later) the result of multiplying something by the string `"Hello"` is undefined. We will address these issues later, by *Type-Checking Hygge0 Programs*.

2.1.2 Grammar Ambiguities

According to *Definition 1*, some sequences of symbols may be classified as valid Hygge0 expressions by applying the grammar rules in different ways – which means that some sequences of symbols may have different syntax trees. Therefore, the Hygge0 grammar is **ambiguous**: see *Example 8*.

Example 8 (Grammar Ambiguity)

Consider again the simple Hygge0 expression in *Example 4*. The grammar in *Definition 1* also allows us to accept that same sequence of symbols by applying the rules in a different order and classifying the addition as a sub-expression of the multiplication. This leads to the following syntax tree (observe the nodes at the top-right corner, and contrast them with the syntax tree in *Example 4*):



These two ways to form syntax trees (and group sub-expressions) lead to different expression executions and results, as we will see later in [Example 12](#).

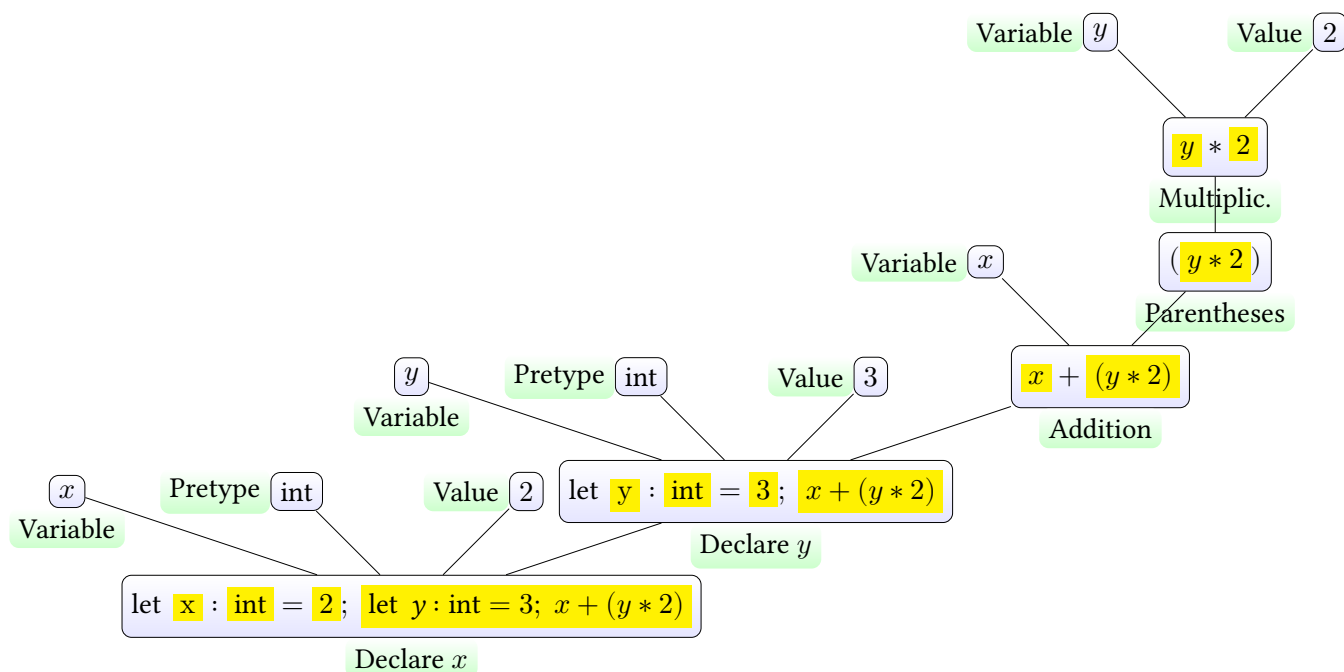
To enforce a specific way to apply the grammar rules and form a syntax tree for a given sequence of symbols, we need to adopt one or more of the following approaches:

1. explicitly use parentheses to group sub-expressions and resolve ambiguities; or
2. revise [Definition 1](#) to make it non-ambiguous, e.g. by arranging the rules for multiplications and additions in a way that enforces one predetermined application order; or
3. define a **precedence** between the rules of [Definition 1](#), e.g. “when checking a sequence of symbols, the grammar rules in [Definition 1](#) must be tried in order, top-to-bottom”.

For now we will intuitively follow the third approach, without delving into details. We will address the second approach later in the course. The first approach (explicitly using parentheses) is conceptually the simplest, but it also makes programs more verbose: see [Example 9](#).

Example 9 (Enforcing Precedence with Parentheses)

To enforce that a sequence of symbols like the one in [Example 4](#) has a unique syntax tree, we can e.g. rewrite “ $x + y * 2$ ” as “ $x + (y * 2)$ ”: by explicitly placing parentheses around the multiplication, we ensure that the sub-expression y is not “captured” by the addition. The result is the following syntax tree:



Exercise 10 (More Grammar Ambiguities)

The *Definition 1* has many more ambiguities besides the one discussed in *Example 4*. Write some examples of Hygge0 expressions (using sequencing, logical operators, relations...) that are accepted by the grammar by applying its rules in different ways – hence forming different syntax trees that group their sub-expressions in different ways.

For example: according to the grammar, in which different ways could we group the sub-expressions of $2 * 3 < 2 + 5$?

2.2 Formal Semantics of Hygge0

We now define how Hygge0 programs are expected to behave when running. To this end, we define a **structural operational semantics** for Hygge0 expressions (see *Definition 4* below). The semantics provides a high-level view of the behaviour of a Hygge0 expression: it can be directly used to write an **interpreter** of the language, and it also helps in writing a compiler by unambiguously stating how expressions should be evaluated, and what results they produce.

In order to define the operational semantics Hygge0, we first need to introduce some technical definitions.

2.2.1 Preliminaries: Inference Rules and Substitutions

First, we need to define how to substitute a variable inside an Hygge0 expression, by replacing it with another Hygge0 expression.

Definition 2 (Substitution of a Variable in an Hygge0 Expression)

The substitution of a variable x with expression e' in expression e is written $e[x \mapsto e']$, and is defined as follows:

$$\begin{aligned}
 x[x \mapsto e'] &= e' \\
 y[x \mapsto e'] &= y \quad (\text{when } y \neq x) \\
 v[x \mapsto e'] &= v \\
 (e_1 + e_2)[x \mapsto e'] &= e_1[x \mapsto e'] + e_2[x \mapsto e'] \\
 (e_1 * e_2)[x \mapsto e'] &= e_1[x \mapsto e'] * e_2[x \mapsto e'] \\
 &\vdots \\
 (\text{let } x : t = e_1; e_2)[x \mapsto e'] &= \text{let } x : t = e_1[x \mapsto e']; e_2 \\
 (\text{let } y : t = e_1; e_2)[x \mapsto e'] &= \text{let } y : t = e_1[x \mapsto e']; e_2[x \mapsto e'] \quad (\text{when } y \neq x) \\
 (\text{type } y = t; e)[x \mapsto e'] &= \text{type } y = t; e[x \mapsto e']
 \end{aligned}$$

Definition 2 says that the substitution of variable x with e' in e proceeds along each sub-expression of e , until it reaches a terminal symbol; and if that terminal symbol is the variable x , it is substituted with e' . Moreover, the substitution encounters a “let $x : t = e_1; e_2$ ” that redefines the variable x , then the substitution is propagated through e_1 , but not in the scope e_2

Example 10 (Simple Substitution)

Consider the expression $x+3$. To obtain the expression derived from $x+3$ by substituting the variable x with the value 2, we write:

$$(x + 3)[x \mapsto 2]$$

which, according to *Definition 2*, this is equivalent to:

$$(x + 3)[x \mapsto 2] = x[x \mapsto 2] + 3[x \mapsto 2] = 2 + 3$$

Example 11 (Substitution Under “Let”)

Consider the expression let $x : t = z + 1; x + z$. Since the “let” is declaring (i.e. **binding**) variable x , the substitution of x has no effect and produces the same expression:

$$(\text{let } x : t = z + 1; x + z)[x \mapsto 42] = \text{let } x : t = z + 1; x + z$$

Instead, the variable z is not bound (i.e. it is *free* in the given expression), so its substitution *does* have an effect and produces an updated expression:

$$\begin{aligned} & (\text{let } x : t = z + 1; x + z) [z \mapsto 42] \\ = & \text{let } x : t = (z + 1) [z \mapsto 42]; (x + z) [z \mapsto 42] \\ = & \text{let } x : t = (z [z \mapsto 42] + 1 [z \mapsto 42]); (x [z \mapsto 42] + z [z \mapsto 42]) \\ = & \text{let } x : t = 42 + 1; x + 42 \end{aligned}$$

Exercise 11 (Defining Substitutions)

Definition 2 is incomplete. Provide a definition of the missing substitution cases: you should define one new case for each form of expression e that appears in the *Definition 1*, but is omitted in *Definition 2*, such as $<$, $=$, $\text{if } \dots \text{ then } \dots \text{ else } \dots$, $\text{print}(\dots)$. Write some examples showing how the new substitution cases work.

Then, we need a way to formalise how to derive a conclusion using a set of inference rules.

Definition 3 (Inference Rules)

An inference rule has the following shape:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{C} \text{ [Name]}$$

which reads: according to the rule called “Name”, the *conclusion* C is true when all the *premises* P_1, P_2, \dots, P_n are true.

The number of premises above the line can be 0: in this case, we leave the space above the line empty, and we say that the inference rule is an **axiom**.

Given a set of inference rules, the application of such rules to prove a conclusion creates a tree structure, called a **derivation**.

2.2.2 Structural Operational Semantics of Hygge0

We can now define the semantics of Hygge0 as a set of inference rules describing when an expression can **reduce** (i.e. perform an execution step) and become another expression.

More specifically, given an Hygge0 expression e , we define its behaviour under a **runtime environment** R to model how e can interact with the world around it. Since Hygge0 is a simple language that only performs input/output on the system console, R is a record with two fields:

- R .Printer provides a way to send an output to the console. It can be left undefined, which means that no console output is possible.
-

- R .Reader provides a way to read an input from the console. It can be left undefined, which means that no console input is possible.

Definition 4 (Structural Operational Semantics of Hygge0)

We define the **structural operational semantics of Hygge0** as a reduction of the following form:

$$\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle$$

which reads: the composition of runtime environment R and expression e reduces into an updated environment R' and an updated expression e' . The reduction is defined by the following rules.

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e + e_2 \rangle \rightarrow \langle R' \bullet e' + e_2 \rangle} \text{ [R-Add-L]} \quad \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v + e \rangle \rightarrow \langle R' \bullet v + e' \rangle} \text{ [R-Add-R]}$$

$$\frac{v_1 + v_2 = v_3}{\langle R \bullet v_1 + v_2 \rangle \rightarrow \langle R \bullet v_3 \rangle} \text{ [R-Add-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e * e_2 \rangle \rightarrow \langle R' \bullet e' * e_2 \rangle} \text{ [R-Mul-L]} \quad \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v * e \rangle \rightarrow \langle R' \bullet v * e' \rangle} \text{ [R-Mul-R]}$$

$$\frac{v_1 \times v_2 = v_3}{\langle R \bullet v_1 * v_2 \rangle \rightarrow \langle R \bullet v_3 \rangle} \text{ [R-Mul-Res]}$$

⋮

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet (e) \rangle \rightarrow \langle R \bullet (e') \rangle} \text{ [R-Par-Eval]} \quad \frac{}{\langle R \bullet (v) \rangle \rightarrow \langle R \bullet v \rangle} \text{ [R-Par-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \{e\} \rangle \rightarrow \langle R \bullet \{e'\} \rangle} \text{ [R-Curly-Eval]} \quad \frac{}{\langle R \bullet \{v\} \rangle \rightarrow \langle R \bullet v \rangle} \text{ [R-Curly-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e; e_2 \rangle \rightarrow \langle R \bullet e'; e_2 \rangle} \text{ [R-Seq-Eval]} \quad \frac{}{\langle R \bullet v; e \rangle \rightarrow \langle R \bullet e \rangle} \text{ [R-Seq-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{let } x : t = e; e_2 \rangle \rightarrow \langle R' \bullet \text{let } x : t = e'; e_2 \rangle} \text{ [R-Let-Eval-Init]}$$

$$\frac{}{\langle R \bullet \text{let } x : t = v; e \rangle \rightarrow \langle R \bullet e[x \mapsto v] \rangle} \text{ [R-Let-Subst]}$$

$$\frac{}{\langle R \bullet \text{type } x = t; e \rangle \rightarrow \langle R \bullet e \rangle} \text{ [R-Type-Res]}$$

$$\frac{}{\langle R \bullet e : t \rangle \rightarrow \langle R \bullet e \rangle} \text{ [R-Ascr-Res]}$$

(Continues on the next page...)

(...Continued from the previous page)

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{assert}(e) \rangle \rightarrow \langle R' \bullet \text{assert}(e') \rangle} \text{ [R-Assert-Eval-Arg]}$$

$$\frac{}{\langle R \bullet \text{assert}(\text{true}) \rangle \rightarrow \langle R \bullet () \rangle} \text{ [R-Assert-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{print}(e) \rangle \rightarrow \langle R' \bullet \text{print}(e') \rangle} \text{ [R-Print-Eval-Arg]}$$

$$\frac{R.\text{Printer is defined}}{\langle R \bullet \text{print}(v) \rangle \rightarrow \langle R \bullet () \rangle} \text{ [R-Print-Res]}$$

$$\frac{R.\text{Reader is defined} \quad R.\text{Reader yields } v}{\langle R \bullet \text{readInt}() \rangle \rightarrow \langle R \bullet v \rangle} \text{ [R-Read-Int]}$$

$$\frac{R.\text{Reader is defined} \quad R.\text{Reader yields } v}{\langle R \bullet \text{readFloat}() \rangle \rightarrow \langle R \bullet v \rangle} \text{ [R-Read-Float]}$$

If the runtime environment and expression $\langle R \bullet e \rangle$ cannot reduce by any of the rules above, and e is not a value, then we say that $\langle R \bullet e \rangle$ is **stuck**.

The [Definition 4](#) formalises a **left-to-right** evaluation order for expressions. The style of this semantics is called **small-step**, because it describes each Hygge0 program computation in terms of reductions — and this includes mathematical expressions, whose operations reduce until they reach a value.

Consider, for example, how we can reduce an addition $e_1 + e_2$. Intuitively, ignoring the runtime environment R , we have:

- by rule [R-Add-L], if the left operand of the addition is an expression e which can reduce to e' (according to the premise of the rule), then the whole addition $e + e_2$ reduces to the addition $e' + e_2$;
- by rule [R-Add-R], if the left operand of the addition is a value v , and the right operand is an expression e which can reduce to e' (according to the premise of the rule), then the whole addition $v + e$ reduces to the addition $v + e'$;
- by rule [R-Add-Res], if both operands of the addition are values (v_1 and v_2), then the expression reduces to a value v_3 (which, by the premise of the rule, is the result of $v_1 + v_2$).

Summing up: the rules in [Definition 4](#) require us to reduce an addition by first reducing the left operand into a value, then by reducing the right operand into a value, and finally by reducing the whole addition into a result value.

Example 12 (Reductions of a Simple Hygge0 Expression)

Consider the simple Hygge0 expression in [Example 4](#). To see how it reduces in a runtime environment R , we apply the rules of the [Definition 4](#).

The first reduction step is allowed by rule [R-Let-Subst], which reduces the outermost “let” by substituting x with the value 2:

$$\frac{}{\langle R \bullet \begin{array}{l} \text{let } x : \text{int} = 2; \\ \text{let } y : \text{int} = 3; \\ x + y * 2 \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{l} \text{let } y : \text{int} = 3; \\ 2 + y * 2 \end{array} \rangle} \text{[R-Let-Subst]}$$

The second reduction step is also allowed by rule [R-Let-Subst], which now substitutes y with 3:

$$\frac{}{\langle R \bullet \begin{array}{l} \text{let } y : \text{int} = 3; \\ 2 + y * 2 \end{array} \rangle \rightarrow \langle R \bullet 2 + 3 * 2 \rangle} \text{[R-Let-Subst]}$$

The next reduction steps depend on the syntax tree initially chosen for the sub-expression $x + y * 2$, which is reflected in the current expression $2 + 3 * 2$.

- If the syntax tree of $2 + 3 * 2$ follows [Example 4](#) (where the multiplication is a sub-expression of the addition) we can only perform a reduction by rule [R-Add-R] in [Definition 4](#), which in turn has a premise requiring us to show that the right operand of the addition (i.e. the multiplication $3 * 2$) can reduce. To satisfy this premise, we can use rule [R-Mul-Res] to perform the multiplication between values 3 and 2, without further premises. Therefore, we have the following reduction:

$$\frac{\frac{}{\langle R \bullet 3 * 2 \rangle \rightarrow \langle R \bullet 6 \rangle} \text{[R-Mul-Res]}}{\langle R \bullet 2 + 3 * 2 \rangle \rightarrow \langle R \bullet 2 + 6 \rangle} \text{[R-Add-R]}$$

The reduction above is followed by another reduction by rule [R-Add-Res], giving us the result 8:

$$\frac{}{\langle R \bullet 2 + 6 \rangle \rightarrow \langle R \bullet 8 \rangle} \text{[R-Add-Res]}$$

- Instead, if the syntax tree of $2 + 3 * 2$ follows [Example 8](#) (where the addition is a sub-expression of the multiplication) we can only perform a reduction by rule [R-Mul-L], using in its premise the reduction of the left operand of the multiplication (using rule [R-Add-Res]):

$$\frac{\frac{}{\langle R \bullet 2 + 3 \rangle \rightarrow \langle R \bullet 5 \rangle} \text{[R-Add-Res]}}{\langle R \bullet 2 + 3 * 2 \rangle \rightarrow \langle R \bullet 5 * 2 \rangle} \text{[R-Mul-L]}$$

The reduction above is followed by another reduction by rule [R-Mul-Res], giving us the result 10:

$$\frac{}{\langle R \bullet 5 * 2 \rangle \rightarrow \langle R \bullet 10 \rangle} \text{[R-Mul-Res]}$$

A few more remarks about the semantic rules in [Definition 4](#):

- the rules ignore the pretypes appearing in Hygge0 expressions:
 - by rule [R-Let-Subst], a let $x : t = v; e'$ reduces by substituting x with v in e' , ignoring t ;
 - by rule [R-Type-Res], a type declaration type $x = t; e$ reduces to e (ignoring x and t);
 - by rule [R-Ascr-Res], a type ascription like $e : t$ reduces to e (ignoring t);
- by rules [R-Seq-Eval] and [R-Seq-Res], sequencing of expressions “ $e_1; e_2$ ” is reduced by first reducing e_1 into a value v , and then discarding v , whole expression becomes e_2 and the reductions can continue from there;
- to reduce an assertion $\text{assert}(e)$, we first need to reduce its argument e into a value (by applying rule [R-Assert-Eval-Arg] 0 or more times); then, when we reach the expression $\text{assert}(v)$, we can only reduce it by applying rule [R-Assert-Res], which in turn requires the argument v to be true, producing the unit value $()$. As a consequence, $\text{assert}(\text{false})$ does not reduce, hence it is stuck: this models a program that stops running due to a failed assertion.

If no rule can be applied (e.g. for an addition like $2 + \text{true}$, or a failed assertion like $\text{assert}(\text{false})$) then we say that the expression is *stuck*.

Exercise 12 (Expression Reductions)

Using the rules in [Definition 4](#), show the reductions of the following expressions, in a runtime environment R :

- let $x : \text{int} = 3 + 2; x + 1$
 - let $x : \text{int} = 3 + 2; \text{print}(x + 1)$
 - let $x : \text{int} = 3 + 2; \text{print}(x + 1); \text{print}(x + 2)$
-

Exercise 13 (Defining Semantic Rules)

[Definition 4](#) is incomplete. Provide a definition of the missing semantic rules: you should define one rule for each form of expression e that appears in the [Definition 1](#), but is omitted in [Definition 4](#) – such as $<$, $=$, $\text{if } \dots \text{ then } \dots \text{ else } \dots$. Write some examples showing how the new reduction rules work.

Note: You may have noticed that the premises of some rules of the [Definition 4](#) say that $\langle R \bullet e \rangle$ may reduce to $\langle R' \bullet e' \rangle$, so the runtime environments R and R' may be different before and after the reduction. However, none of the rules actually causes reductions where R' is different from R . Therefore, *in principle* we may express the same semantics by having $\langle R \bullet e \rangle$ reduce to $\langle R \bullet e' \rangle$ (i.e. preserving the same R).

This observation is correct. However, in the next modules we will extend the capabilities of Hygge0, and there will be new rules that will actually update R into a different R' ;

by allowing this possibility now, we will be able to seamlessly integrate the new rules with the current ones.

2.3 Type-Checking Hygge0 Programs

As mentioned earlier in *Example 7*, the Hygge0 syntax accepts expressions that, albeit syntactically valid, are “meaningless” and incorrect. When executed with the semantics *Example 12*, such programs may get stuck: see *Example 13*.

Example 13

Consider again the Hygge0 expression in *Example 7*. It is syntactically valid, and it can perform a reduction step:

$$\frac{\langle R \bullet \boxed{\text{let } x : \text{foo} = 2; \\ y + x * \text{”Hello”}} \rangle}{\langle R \bullet y + 2 * \text{”Hello”} \rangle} \quad [\text{R-Let-Subst}]$$

However, the resulting expression $y + 2 * \text{”Hello”}$ is stuck, because it is not a value, and there is no rule in *Definition 4* that can be used to reduce it further.

We now develop a typing system for Hygge0 that rejects “bad” expressions, and guarantees that program behaviours like *Example 13* never occur. The Hygge0 typing system consists of four components, illustrated in the next sections:

- *Types and the Typing Environment*
- a **type resolution judgement** “ $\Gamma \vdash t \triangleright T$ ” used for *Resolving Pretypes into Valid Types*;
- a **typing judgement** “ $\Gamma \vdash e : T$ ” which is the core of *The Hygge0 Typing System (Part 1)*;
- finally, a **subtyping judgement** “ $\Gamma \vdash T \leq T'$ ” which makes the type system more flexible, by introducing *Subtyping*.

With these components in place, the Hygge0 typing system ensures that well-typed programs never get stuck (under some assumptions), and enjoy the *Properties of Well-Typed Hygge0 Programs*.

2.3.1 Types and the Typing Environment

In order to type-check Hygge0 expressions, we need to define our *types* ([Definition 5](#)), and the *typing environment* ([Definition 6](#)) we will use to hold the information we need for typing an Hygge0 expression.

Definition 5 (Hygge0 Types)

We use the symbol T to denote a **type**, which has the following shape:

$$\begin{array}{l} \text{Type } T ::= \text{bool} \mid \text{int} \mid \text{float} \mid \text{string} \mid \text{unit} \quad (\text{Basic types}) \\ \quad \quad \quad \mid x \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{Type variable}) \end{array}$$

According to [Definition 5](#), a type T looks very similar to a *pretype* t in [Definition 1](#); however, types and pretypes have a different role, that we will see shortly.

Definition 6 (Hygge0 Typing Environment)

We use the symbol Γ to denote a **typing environment**, which is a record with two fields:

- $\Gamma.\text{Vars}$ is a mapping from variables to types;
 - $\Gamma.\text{TypeVars}$ is also a mapping from variables to types.
-

The two mappings $\Gamma.\text{Vars}$ and $\Gamma.\text{TypeVars}$ look similar, but they serve different purposes:

- we will use $\Gamma.\text{Vars}$ to remember the type of each variable x introduced by “let $x : t = \dots$ ”. For example: if $\Gamma.\text{Vars}$ contains the entry $x \mapsto \text{int}$, this means that variable x has type int . This information is used e.g. when type-checking the occurrences of x in [Example 3](#), lines 10, 14, 15.
- we will use $\Gamma.\text{TypeVars}$ to remember each *type variable* x introduced by “type $x = \dots$ ”. You can think of such type variables as type aliases. For example: if $\Gamma.\text{TypeVars}$ contains the entry $\text{MyInt} \mapsto \text{int}$, then MyInt is a type variable corresponding to type int . This information is used e.g. to understand what type MyInt represents in [Example 3](#), line 5.

2.3.2 Resolving Pretypes into Valid Types

The Hygge0 grammar is not aware of which types are valid for type checking: it only recognises the shape of “things that syntactically look like types” – which we call *pretypes*. Such pretypes may be *any* identifier (e.g. int , foobar , xyz123) – and in [Example 7](#) we have seen that the grammar accepts the pretype foo , although it is undefined (just like it also accepts the undefined variable y in the same example). We need to distinguish “bad” cases like [Example 7](#) from “good” cases like [Example 3](#), where MyInt is used correctly, after being defined as an alias of int .

For this reason, during type checking we need to *resolve* pretypes into types, if possible. This is achieved as specified in [Definition 7](#) below.

Definition 7 (Type Resolution in Hygge0)

To check whether a pretype is a valid type, we use the following **type resolution judgement**:

$$\Gamma \vdash t \triangleright T$$

which reads: in the typing environment Γ , the pretype t resolves into type T . The judgement is defined by the following rules (where we write pretypes between quotes, as a visual hint to distinguish them from types):

$$\begin{array}{c} \frac{}{\Gamma \vdash \text{"bool"} \triangleright \text{bool}} \text{ [TRes-Bool]} \quad \frac{}{\Gamma \vdash \text{"int"} \triangleright \text{int}} \text{ [TRes-Int]} \\ \\ \frac{}{\Gamma \vdash \text{"float"} \triangleright \text{float}} \text{ [TRes-Float]} \quad \frac{}{\Gamma \vdash \text{"string"} \triangleright \text{string}} \text{ [TRes-String]} \\ \\ \frac{}{\Gamma \vdash \text{"unit"} \triangleright \text{unit}} \text{ [TRes-Unit]} \quad \frac{\Gamma.\text{TypeVars}(x) = T}{\Gamma \vdash \text{"x"} \triangleright x} \text{ [TRes-Var]} \end{array}$$

According to the [Definition 7](#), the judgement $\Gamma \vdash \text{"bool"} \triangleright \text{bool}$ holds without any premise (by rule [TRes-Bool]), and therefore, in *any* typing environment Γ , the pretype "bool" resolves into the valid basic type bool; other rules lead to similar conclusions for types int, float, string, and unit.

Instead, a pretype like "MyInt" can only be resolved by rule [TRes-Var], which has a premise requiring that the typing environment Γ "knows about" the existence of a type variable called MyInt – i.e. the mapping $\Gamma.\text{TypeVars}$ must associate MyInt to some type T . If this premise holds, then the pretype "MyInt" resolves into the valid type variable MyInt in the typing environment Γ .

2.3.3 The Hygge0 Typing System (Part 1)

We now have all the ingredients to define the type system of Hygge0 (at least in a first version that we will improve later).

Definition 8 (Type System of Hygge0 – Part 1)

To check whether an Hygge0 expression is well-typed, we use the following **type checking judgement**:

$$\Gamma \vdash e : T$$

which reads: in the typing environment Γ , the expression e has type T . The judgement

is defined by the following rules.

$$\begin{array}{c}
\frac{v \text{ is an integer value}}{\Gamma \vdash v : \text{int}} \text{ [T-Val-Int]} \quad \frac{v \text{ is a string value}}{\Gamma \vdash v : \text{string}} \text{ [T-Val-String]} \\
\\
\frac{v \in \{\text{true}, \text{false}\}}{\Gamma \vdash v : \text{bool}} \text{ [T-Val-Bool]} \quad \frac{}{\Gamma \vdash () : \text{unit}} \text{ [T-Val-Unit]} \\
\\
\frac{v \text{ is a single-precision float value}}{\Gamma \vdash v : \text{float}} \text{ [T-Val-Float]} \\
\\
\frac{\Gamma.\text{Vars}(x) = T}{\Gamma \vdash x : T} \text{ [T-Var]} \\
\\
\frac{T \in \{\text{int}, \text{float}\} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 + e_2 : T} \text{ [T-Add]} \\
\\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{ [T-Cond]} \\
\\
\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1; e_2 : T'} \text{ [T-Seq]} \\
\\
\vdots \\
\\
\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash e_1 : T \quad \{\Gamma \text{ with Vars } + (x \mapsto T)\} \vdash e_2 : T'}{\Gamma \vdash \text{let } x : t = e_1; e_2 : T'} \text{ [T-Let]} \\
\\
\frac{x \notin \{\text{bool}, \text{int}, \text{float}, \text{string}, \text{unit}\} \quad x \notin \Gamma.\text{TypeVars} \quad \Gamma \vdash t \triangleright T \quad \{\Gamma \text{ with TypeVars } + (x \mapsto T)\} \vdash e : T' \quad x \notin T'}{\Gamma \vdash \text{type } x = t; e : T'} \text{ [T-Type]} \\
\\
\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash e : T}{\Gamma \vdash (e : t) : T} \text{ [T-Ascr]} \\
\\
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{assert}(e) : \text{unit}} \text{ [T-Assert]} \\
\\
\frac{T \in \{\text{bool}, \text{int}, \text{float}, \text{string}\} \quad \Gamma \vdash e : T}{\Gamma \vdash \text{print}(e) : \text{unit}} \text{ [T-Print]}
\end{array}$$

Here is a description of the main rules in [Definition 8](#).

- The first rules assign a type to a value, depending on whether the value is an integer, string, boolean, unit, or float. These rules do not have any requirement on the typing environment Γ .
- Rule [T-Var] says that in order to assign type T to a variable x , we need to check

whether $\Gamma.\text{Var}$ assigns type T to x .

- Rule [T-Add] assigns type T to an addition “ $e_1 + e_2$ ” – but only if both operands have type T in the same environment Γ , and T is either `int` or `float` (hence, an expression like `2 + "Hello"` cannot be typed).
- Rule [T-Cond] assigns type T to a conditional expression “if e_1 then e_2 else e_3 ” – but only if e_1 has type `bool`, and both e_2 and e_3 have the same type T .
- Rule [T-Seq] assigns type T' to a sequencing of expressions “ $e_1; e_2$ ”. One premise of the rule require that e_2 has that type T' ; the other premise of the rule requires that e_1 is also type-checked and has some type T (which is not used).
- Rule [T-Let] assigns a type T' to an expression “let $x : t = e_1; e_2$ ”, where e_2 is in the scope of the declaration of variable x . The premises of the rule require that:
 1. the pretype t of x can be resolved into a valid type T ;
 2. the expression e_1 that initialises x has that type T ;
 3. the expression e_2 (in the scope of x) can be type-checked, and has type T' , in a typing environment that is equal to Γ – except that we extend its `Vars` by adding a mapping from x to T . As a consequence, x can appear as a sub-expression of e_2 , where it is typed as T .
- Rule [T-Type] assigns a type T' to an expression “type $x = t; e$ ”, where e is in the scope of the declaration of the type variable x . The premises of the rule require that:
 1. x is not one of the Hygge0 basic type identifiers;
 2. x is not already defined as a type variable in $\Gamma.\text{TypeVars}$;
 3. the pretype t used to define x can be resolved into a valid type T ;
 4. the expression e (in the scope of x) can be type-checked, and has type T' , in a typing environment that is equal to Γ – except that we extend its `TypeVars` by adding a mapping from x to T . As a consequence, x can be used as a type in e , where it is treated as an alias of T ;
 5. the type T' assigned to e does not contain any reference to x .
- Rule [T-Ascr] assigns a type T to an expression “ $e : t$ ”, which is a type ascription attempting to assign type t to e . The premises of the rule require that:
 1. the pretype t used in the ascription can be resolved into a valid type T ;
 2. the expression e can be type-checked in Γ and indeed has type T . As a consequence, a type ascription like “`42 : int`” type-checks, whereas “`42 : string`” does not type-check.

Example 14 (Typing Derivation of a Hygge0 Expression)

Consider the following Hygge0 expression:

$$\begin{array}{l} \text{let } x : \text{int} = 2; \\ x + 3 \end{array}$$

We can show that the whole expression has type `int`, in an empty typing environment: to achieve this, we apply the typing rules in [Definition 8](#) to construct the following typing derivation.

$$\begin{array}{c}
 \frac{\frac{\frac{}{\{ \text{Vars} = \emptyset \}} \vdash \text{"int"} \triangleright \text{int}}{\{ \text{TypeVars} = \emptyset \}} \quad [\text{TRes-Int}] \quad \frac{}{\{ \text{Vars} = \emptyset \}} \vdash 2 : \text{int} \quad [\text{T-Val-Int}]}{\{ \text{Vars} = \emptyset \}} \vdash \text{let } x : \text{int} = 2; \quad : \text{int} \quad [\text{T-Let}]} \\
 \frac{\frac{\frac{\frac{\text{Vars}(x) = \text{int}}{\{ \text{Vars} = \{x \mapsto \text{int}\}} \vdash x : \text{int}}{\{ \text{TypeVars} = \emptyset \}} \quad [\text{T-Var}] \quad \frac{}{\{ \text{Vars} = \{x \mapsto \text{int}\}} \vdash 3 : \text{int}} \quad [\text{T-Val-Int}]}{\{ \text{TypeVars} = \emptyset \}} \vdash x + 3 : \text{int} \quad [\text{T-Add}]} \\
 \frac{\{ \text{Vars} = \emptyset \}} \vdash \text{let } x : \text{int} = 2; \quad : \text{int} \quad [\text{T-Let}]}{\{ \text{TypeVars} = \emptyset \}} \vdash \text{let } x : \text{int} = 2; \quad : \text{int}
 \end{array}$$

We can read the typing derivation above by starting from the root and moving upwards:

- to show that the expression “let $x : t = 2; \dots$ ” has type `int` in an empty environment, rule [T-Let] checks whether “`int`” is a valid pretype, whether 2 has type `int`, and whether $x + 3$ has type `int`. To type-check the latter, rule [T-Let] extends the typing environment with the information that the newly-declared variable x has type `int`;
 - to show that the expression $x + 3$ has type `int`, rule [T-Add] checks that both operands have type `int`:
 - the left operand is typed by rule [T-Var], which looks in the typing environment and finds that x has type `int`;
 - the right operand is typed by rule [T-Val-Int].
-

Exercise 14 (Typing Expressions)

Determine whether the following typing judgements hold, by writing the typing derivation of each one.

1. $\{ \text{Vars} = \emptyset; \text{TypeVars} = \emptyset \} \vdash (4 + 2) + 1 : \text{int}$
 2. $\{ \text{Vars} = \emptyset; \text{TypeVars} = \emptyset \} \vdash \text{if true then "Hello" else "World"} : \text{string}$
 3. $\{ \text{Vars} = \{x \mapsto \text{int}\}; \text{TypeVars} = \emptyset \} \vdash (x + 2) + 1 : \text{int}$
 4. $\{ \text{Vars} = \emptyset; \text{TypeVars} = \emptyset \} \vdash \text{let } x : \text{int} = 42; (x + 2) + 1 : \text{int}$
 5. $\{ \text{Vars} = \emptyset; \text{TypeVars} = \emptyset \} \vdash \text{let } x : \text{int} = 2 + 1; \text{print}(x + 2); \text{"Bye!"} : \text{string}$
-

Exercise 15 (Inspect a “Bad” Expression)

Consider again the expression [Example 7](#): explain why it does not type-check according to the rules in [Definition 8](#).

Exercise 16 (Defining Typing Rules)

Definition 8 is incomplete. Provide a definition of the missing typing rules: you should define one rule for each form of expression e that appears in the *Definition 1*, but is omitted in *Definition 8* – such as $<$, $=$, $*$, not , $\text{readInt}()$. Write some examples showing how the new typing rules work.

2.3.4 Subtyping

The Hygge0 typing rules introduced in *Definition 8* are very limited in their use of type variables: there is a rule called [T-Type] to type-check declarations of new type variables, by adding them to $\Gamma.\text{TypeVars}$, but there is no rule that uses the information in $\Gamma.\text{TypeVars}$. We can observe the resulting limitation in *Example 15* below.

Example 15 (How Can We Use Type Variables?)

Consider this Hygge0 expression:

```
type MyInt = int;
let x : MyInt = 2;
x + 3
```

This expression declares a new type `MyType` as an alias of `int`, and then tries to declare a variable x of that type. Intuitively, the whole expression should have type `int` (i.e. the type of $x + 3$ on the last line). However, if we try to type-check the expression, we cannot complete a typing derivation: (below we omit part of the derivation with “...”)

$$\begin{array}{c}
 \frac{\text{TypeVars(MyInt) = int}}{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{"MyInt"} \triangleright \text{MyInt}} \text{[TRes-Var]} \quad \frac{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash 2 : \text{MyInt}}{\dots} \text{[T-Let]} \\
 \dots \quad \frac{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{let } x : \text{MyInt} = 2; \quad : \text{int}}{x + 3} \text{[T-Let]} \\
 \hline
 \left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \emptyset \end{array} \right\} \vdash \begin{array}{l} \text{type MyInt} = \text{int}; \\ \text{let } x : \text{MyInt} = 2; \quad : \text{int} \\ x + 3 \end{array} \text{[T-Type]}
 \end{array}$$

The issue on the top-right corner of this tentative derivation (in the failed rule application marked with “???”) is that we would need to assign type `MyInt` to `2`, but there is no typing rule allowing us to do that.

We could address the issue highlighted in *Example 15* by adding some *ad hoc* typing rule for similar cases. Instead, we introduce a more general solution: we equip Hygge0 with **subtyping**, i.e. a relation between types which satisfies the Liskov Substitution Principle (*Definition 9*),

Definition 9 (Liskov Substitution Principle)

If a type T is subtype of T' , then any value of type T can be safely used whenever a value of type T' is expected.

Following the spirit of [Definition 9](#), we introduce a subtyping judgement for Hygge0 types ([Definition 10](#)), and then extend its typing system with a *subsumption rule* that makes use of the subtyping ([Definition 11](#)).

Definition 10 (Subtyping in Hygge0)

To check whether a Hygge0 type is subtype of another type, we use the following **subtyping judgement**:

$$\Gamma \vdash T \leq T'$$

which reads: in the typing environment Γ , the type T is a subtype of T' . The judgement is defined by the following rules.

$$\frac{}{\Gamma \vdash T \leq T} \text{ [TSub-Refl]}$$

$$\frac{\Gamma \vdash \Gamma.\text{TypeVars}(x) \leq T}{\Gamma \vdash x \leq T} \text{ [TSub-Var-L]} \quad \frac{\Gamma \vdash T \leq \Gamma.\text{TypeVars}(x)}{\Gamma \vdash T \leq x} \text{ [TSub-Var-R]}$$

The first rule in [Definition 10](#) says that in any typing environment Γ , any type T is subtype of itself (i.e. the subtyping relation is *reflexive*).

Instead, rule [TSub-Var-L] allows us to check whether a type variable x is subtype of T in a given Γ : to that end, the premise of the rule requires us to check whether $\Gamma.\text{TypeVars}$ maps x to a type which is a subtype of T (i.e. we need to apply the rule recursively). Rule [TSub-Var-R] is similar, but it allows us to check whether a type T is a subtype of a type variable x .

Definition 11 (Type System of Hygge0 – Part 2)

The full typing system of Hygge0 is defined by extending the typing rules in [Definition 8](#) with the following rule, called **subsumption rule**:

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'} \text{ [T-Sub]}$$

Rule [T-Sub] in [Definition 11](#) means: in a given typing environment Γ , if we can type expression e with type T , and show that T is subtype of T' , then we can conclude that expression e has (also) type T' . Or, equivalently: to assign type T' to e , we need to show that e has a type T which is subtype of T' .

We can now type-check the problematic expression in [Example 15](#), as shown in [Example 16](#) below.

Example 16

Consider again the Hygge0 expression in [Example 15](#). By using the Hygge0 type system with subsumption in [Definition 11](#), we can write the following typing derivation: (below we omit part of the derivation with “...”)

$$\begin{array}{c}
\frac{\text{TypeVars(MyInt)} = \text{int}}{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{"MyInt"} \triangleright \text{MyInt}} \text{[TRes-Var]} \quad \frac{\dots \vdash 2 : \text{int}}{\dots \vdash 2 : \text{MyInt}} \text{[T-Val-Int]} \quad \frac{\frac{\frac{\frac{}{\text{int} \leq \text{int}}{\text{[TSub-Refl]}} \quad \frac{}{\text{int} \leq \text{MyInt}}{\text{[TSub-Var-R]}}} \left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\}} \left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{int} \leq \text{MyInt}} \text{[T-Sub]} \\
\frac{\dots \vdash 2 : \text{MyInt}}{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{let } x : \text{MyInt} = 2; \quad : \text{int}} \text{[T-Let]} \\
\frac{\dots \vdash 2 : \text{MyInt} \quad \left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash \text{let } x : \text{MyInt} = 2; \quad : \text{int}}{\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \emptyset \end{array} \right\} \vdash \text{type MyInt} = \text{int}; \\ \text{let } x : \text{MyInt} = 2; \quad : \text{int} \\ x + 3} \text{[T-Type]}
\end{array}$$

Continues...

In this derivation we use the subsumption rule [T-Sub] to show that 2 has type MyInt (which we could not achieve in [Example 15](#)). To reach this conclusion, our application of rule [T-Sub] leverages its two premises: 2 has type int, and by the subtyping rules, int is a subtype of MyInt (because, in the current typing environment, MyInt is a type variable that maps to int).

Exercise 17 (Applying Subtyping and the Subsumption Rule)

Complete the missing part of typing derivation in [Example 16](#), marked with “Continues...” (on the right, last premise of [T-Let]). To this end, you should write a typing derivation for the following typing judgement:

$$\left\{ \begin{array}{l} \text{Vars} = \{x \mapsto \text{MyInt}\} \\ \text{TypeVars} = \{\text{MyInt} \mapsto \text{int}\} \end{array} \right\} \vdash x + 3 : \text{int}$$

Hint: You’ll need to use the typing rules [T-Add], [T-Var], [T-Sub] and [T-Val-Int]. When using [T-Sub], you will need to use the subtyping rules [TSub-Var-L] and [TSub-Refl].

2.3.5 Properties of Well-Typed Hygge0 Programs

We can finally state the main property of the Hygge0 typing system: if a Hygge0 expression e is well-typed in an empty typing environment, then e will reduce to a value (in 0 or more steps) without getting stuck – unless it reads invalid inputs from the console, or it contains an assertion that fails. In other words, well-typed Hygge0 programs can only get stuck due to:

1. external factors (bad console inputs), or
2. programmer mistakes caught by assertions (e.g. the programmer expects that $x = 0$ is true, writes an assertion to check it, but this fails at runtime because the running program computes $x = 3$ instead).

This is formalised in [Theorem 1](#) below.

Theorem 1 (Type Safety and Progress)

Take a runtime environment R where both $R.Reader$ and $R.Printer$ are defined. Take any Hygge0 expression e and type T such that:

$$\left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \emptyset \end{array} \right\} \vdash e : T$$

Now, take any R' and e' such that:

- $\langle R \bullet e \rangle \rightarrow \dots \rightarrow \langle R' \bullet e' \rangle$ (i.e. $\langle R \bullet e \rangle$ reduces to $\langle R' \bullet e' \rangle$ in 0 or more steps); and
- along each reduction step, $R.Reader$ yields values of the type expected by e (e.g. if e is performing `readInt()`, then the value received via $R.Reader$ is an integer).

Then, we have:

- $\emptyset \vdash e' : T$ (i.e. e' preserves the type T of e); and
 - one of the following holds:
 - $\langle R' \bullet e' \rangle$ can perform another reduction step; or
 - e' is a value (i.e. e has fully reduced without getting stuck); or
 - e' is stuck and contains a sub-expression `assert(false)` (i.e. an assertion violation).
-

The proof of *Theorem 1* is outside the scope of this course — but such a proof can be achieved by using standard programming language theory techniques. This is the main payoff of developing a formal specification for a programming language: it helps us understanding and proving whether its design is correct.

2.4 References and Further Readings

The following book (on compiler construction) contains a good overview of context-free grammars:

- Bill Campbell, Iyer Swami, Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. Chapman and Hall/CRC, 2012. [Available on DTU Findit⁹](#).
 - Chapter 3.2 (Context-Free Grammars)

Here is a very popular textbook about programming languages and type systems:

- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002. [Available on DTU Findit¹⁰](#). These chapters, in particular, may be useful:
 - Chapter 2 (Mathematical Preliminaries)

⁹ <https://findit.dtu.dk/en/catalog/5c59eb2fd9001d01e4360926>

¹⁰ <https://findit.dtu.dk/en/catalog/5c34980ed9001d2f3820637f>

- Chapter 3 (Untyped Arithmetic Expressions)
- Chapter 8 (Typed Arithmetic Expressions)

2.5 Lab Activities

1. Try to solve the exercises presented along this module.

Note: The exercises are *not* assessed: their purpose is to practice with the key concepts of this module (grammars, operational semantics, type systems). This experience will be very helpful to understand the Hygge0 compiler internals (in the next module), and to extend the compiler for the course project.

You are welcome (and encouraged!) to discuss the exercises and your solutions with your fellow students – either in person, or on Piazza.

2. During the lab, the teacher will make available a **preliminary** version of the Hygge0 compiler `hyggec` on DTU Learn. You can **optionally** inspect its source code and see whether you can recognise the implementation of (parts of) the language specification presented in this module (grammar, semantics, type system, subtyping). Sometimes, looking at an implementation may clarify how a specification is supposed to work...

Note: This is an **optional** lab activity, and you may not immediately grasp all the implementation details of `hyggec`. We will delve into the `hyggec` internals (and its connection to the Hygge0 specification) in the next module.

Module 3: Hands-On with hyggec

This module outlines the implementation of the hyggec compiler, in its minimal version that implements the *Hygge0 programming language specification*. We explore how hyggec works, and how it translates a Hygge0 program into corresponding RISC-V code. We see how to extend both the Hygge0 language and the hyggec compiler with new features. We conclude this module with some *Project Ideas*.

3.1 Quick Start

1. Go to the hyggec Git repository: <https://gitlab.gbar.dtu.dk/02247/f23/hyggec> (NOTE: you will need to log in with your DTU account)
2. Create your own fork of the repository, by clicking on the “Fork” button (below the hyggec title and logo, to the left)
3. Clone your forked repository on your computer (use the URL below the hyggec title and logo. You may choose between “HTTPS” or “SSH”, and the latter is probably better; you may need to [configure your SSH keys for authenticating](#)¹¹)
4. Follow the instructions in the file `README.md`

3.2 The Compiler Phases of hyggec

In the beginning of the course, we discussed *what is a compiler* and what are the typical phases that allow a compiler to translate an input program into an output program.

hyggec has the same phases illustrated in Fig.1. In this section we will explore how each phase is implemented. The overall picture is illustrated in Fig.3.1, where:

- each compiler phase is annotated with the file that handles it in hyggec, and
- each intermediate compilation product is annotated with the data type that hyggec uses to represent it.

¹¹ <https://gitlab.gbar.dtu.dk/help/ssh/README.md>

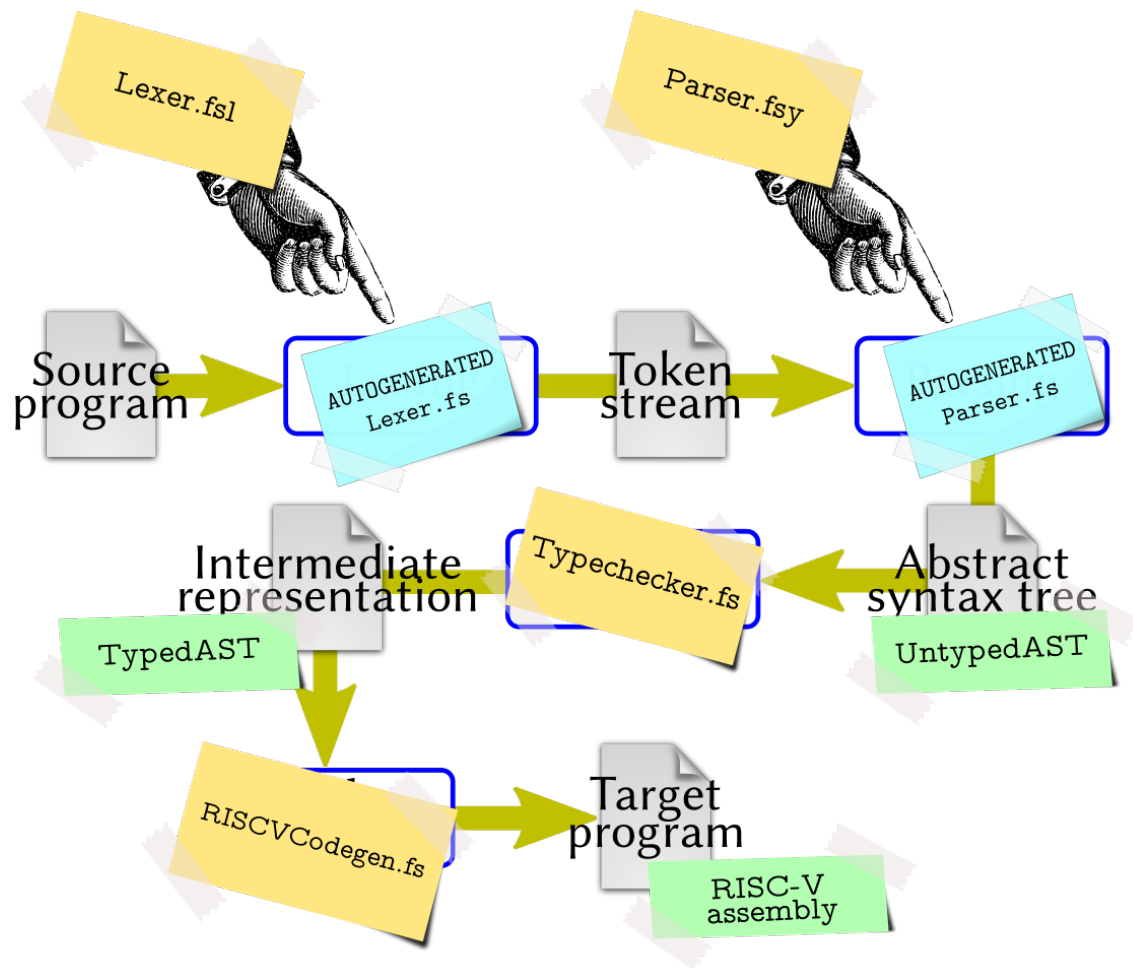


Fig. 3.1: Phases of the hygge compiler (diagram based on Fig.1).

3.3 Overview of the hyggec Source Tree

Here is a quick summary of the hyggec source code structure: the files and directories are roughly ordered depending on how soon (and how often) you will use or modify them.

Tip: When exploring the hyggec source code with Visual Studio Code (or similar IDEs), try to hover with the mouse pointer on types, function names, and variables: in most cases, you will see a brief description of their purpose (and you can jump to their definition, if you want).

Table 3.1: Overview of the hyggec source tree.

File or directory name	Description
hyggec and hyggec.bat	Scripts to run hyggec on Unix-like and Windows operating systems. Both scripts cause the automatic recompilation of hyggec if its source code was changed after the last run.
src/	Directory containing the hyggec source files.
tests/	Directory containing the hyggec test suite. We will discuss the structure of its subdirectories in <i>The Test Suite of hyggec</i> .
src/AST.fs	Representation of the Abstract Syntax Tree (AST) of Hygge0, based on the <i>Definition 1</i> . This file is discussed below in <i>The Abstract Syntax Tree</i> .
src/ASTUtil.fs	Utility functions for manipulating an AST (e.g. apply substitutions).
src/Lexer.fsl and src/Parser.fsy	Specification of the lexer and parser that read Hygge0 source code files and build the corresponding AST. These files are discussed below in <i>The Lexer and Parser</i> .
src/Interpreter.fs	Interpreter for Hygge programs, based on the <i>Structural Operational Semantics of Hygge0</i> .
src/Type.fs	Definition of Hygge0 types (based on <i>Definition 5</i>).
src/Typechecker.fs	Functions for performing type checking on a given Hygge0 program AST, according to <i>Definition 8</i> and <i>Definition 11</i> (thus including <i>Subtyping</i>).
src/RISCVCodegen.fs	Functions for generating RISC-V assembly code from a well-typed Hygge0 program AST.
src/RISCV.fs	Functions for creating and manipulating RISC-V assembly code fragments.
src/PrettyPrinter.fs	Functions to pretty-print various compiler data structures (e.g. ASTs or typing environments)
src/Log.fs	Utility functions for logging messages on the console.
src/Util.fs	Miscellaneous utility functions (e.g. for generating unique strings or numbers).

continues on next page

Table 3.1 – continued from previous page

File or directory name	Description
src/CmdLine.fs	Configuration of the hyggec command line options.
src/Program.fs	The main program, with the main function.
examples/	This directory contains a few examples of Hygge0 programs.
src/Test.fs	Test suite configuration and execution functions. The test suite uses the Expecto testing library ¹² .
hyggec.fsproj	.NET project file.
src/RARS.fs	Utility functions to launch <i>RARS – RISC-V Assembler and Runtime Simulator</i> and immediately execute the compiled RISC-V assembly code.
lib/	This directory contains a copy of <i>RARS – RISC-V Assembler and Runtime Simulator</i> , used in RARS.fs (see above).
rars and rars.bat	Scripts to launch RARS on Unix-based and Windows operating systems. These scripts use the copy of RARS contained in the lib/ directory.
README.md and LICENSE.md	Should be self-explanatory...
fsharplint.json	Configuration file for FSharpLint ¹³ (you can ignore it).
src/Parser.fs, src/Parser.fsi, src/Lexer.fs	Auto-generated parser and lexer files, overwritten when the hyggec source is recompiled. Do not edit!
bin/ and obj/	Auto-generated directories, overwritten when the hyggec source is recompiled. Do not edit!

3.4 The Abstract Syntax Tree

After the specification of a programming language is completed, one of the first steps towards developing a compiler is (typically) defining the data structures needed for the internal representation of a syntactically-correct program, after it has been read from an input file. This internal representation is called **Abstract Syntax Tree (AST)** (where the term “abstract” differentiates it from the *concrete* syntax tree handled by the *parser*, that we discuss later). The AST definition follows the grammar of the language being compiled, with possible adjustments for simplifying the overall compiler implementation.

In the case of hyggec, the AST is defined in src/AST.fs and is based on the Hygge0 grammar in [Definition 1](#). However, the AST definition includes some adjustments, that we now illustrate.

¹² <https://github.com/haf/expecto>

¹³ <https://github.com/fsprojects/FSharpLint>

3.4.1 Defining the AST: First Attempt

In a programming language like F#, we can very naturally specify an AST using *discriminated unions*¹⁴. For instance, we can:

1. define a type `Expr` for representing a generic Hygge0 expression e , with a dedicated named case to represent each different kind of expression in *Definition 1*; and
2. in each named case, use fields of type `Expr` for representing sub-expressions.

For example, starting from the bottom of *Definition 1*, we may define our AST type `Expr` as:

```
type Expr =
  | UnitVal // Unit value
  | BoolVal of value: bool // Boolean value
  | IntVal of value: int // Integer value
  // ...
  | Var of name: string // Variable

  // Addition between left-hand-side and right-hand-side sub-expressions.
  | Add of lhs: Expr
        * rhs: Expr

  // Multiplication between left-hand-side and right-hand-side sub-expressions.
  | Mult of lhs: Expr
          * rhs: Expr

  // ...
```

Using the type definition above, the Hygge0 expression “ $42 + x$ ” would be represented in F# as:

```
Add( IntVal(42), Var("x") )
```

This approach is valid, but it has **two drawbacks**.

1. The definition of `Expr` above does not include information about the position (line and column number) of each sub-expression in the original input file being compiled. We will need this information to generate helpful error messages during type checking, to help Hygge0 programmers find and fix errors in their programs.
2. During type checking, we take a Hygge0 expression e and compute its typing derivation by following the syntactic structure of e . Therefore, the typing derivation of e has pretty much the same structure of the syntax tree of e : see, for instance, *Example 14*. The main difference is that, unlike a syntax tree, a typing derivation includes information about:
 - the type assigned to each sub-expression, and
 - the typing environment used to assign such a type.

¹⁴ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions>

This information will be necessary for code generation, so it is very convenient to save it somewhere, when it is generated during type checking. And given the similarities between an AST and a typing derivation, it would be nice to somehow represent both of them with the same data type, thus avoiding code duplication.

3.4.2 The AST Definition

We address both drawbacks discussed above in the final definition of the Hygge0 AST, which is available in `src/AST.fs`. The key idea is that we “wrap” each `Expr` object with a record of type `Node` (representing a node of the AST) which includes:

1. the position of the expression in the original source file, and
2. information about the typing of the expression. This information is absent before type-checking, and becomes available after type-checking.

As a result, we obtain a unified data type that includes the position of each expression, and can represent ASTs both before and after they are type-checked. This approach is adopted by many compilers (although the details may vary depending on the implementation programming language); here it is inspired by the [Scala 3 compiler](#)¹⁵.

The final definition of AST Nodes and Expressions in `hyggec` is the following.

```
type Node<'E, 'T> =
{
  Expr: Expr<'E, 'T> // Hygge expression contained in this AST node
  Pos: Position     // Position of the expression in the input source file
  Env: 'E           // Typing environment used to type-check the expression
  Type: 'T         // Type assigned to the expression
}

and Expr<'E, 'T> =
| UnitVal // Unit value
| BoolVal of value: bool // Boolean value
| IntVal of value: int // Integer value
// ...
| Var of name: string // Variable

// Addition between left-hand-side and right-hand-side sub-expressions.
| Add of lhs: Node<'E, 'T>
      * rhs: Node<'E, 'T>

// Multiplication between left-hand-side and right-hand-side sub-expressions.
| Mult of lhs: Node<'E, 'T>
        * rhs: Node<'E, 'T>

// ...
```

The two type arguments of `Node<'E, 'T>` and `Expr<'E, 'T>` are used as follows:

¹⁵ <https://github.com/lampepfl/dotty>

- 'E specifies what typing environment information is associated to each expression in the AST;
- 'T specifies what type information is assigned to each expression in the AST.

The hygge compiler handles two kinds of ASTs, both based on the definition of `Node<'E, 'T>` above:

- `UntypedAST`, which contains expressions of type `UntypedExpr` (both defined in `src/AST.fs`). These types are just aliases, respectively, for `Node<unit, unit>` and `Expr<unit, unit>`: they represent AST nodes and expressions without any information (`unit`) about their typing environments or types (see [Example 17](#) below). This kind of AST represents a syntactically-valid Hygge0 expression read from an input file; it is produced by [The Lexer and Parser](#), and it may contain expressions that get stuck, like the ones discussed in [Example 13](#).
- `TypedAST`, which contains expressions of type `TypedExpr` (both defined in `src/Typechecker.fs`). These types are just aliases, respectively, for `Node<TypingEnv, Type>` and `Expr<TypingEnv, Type>`: they represent AST nodes and expressions that have been type-checked, and have typing information available (similarly to a typing derivation). We will discuss `TypedASTs` in [Types and Type Checking](#).

Example 17 (Untyped AST of a Hygge0 Expression)

Consider the Hygge0 expression “ $42 + x$ ”. Its representation in F# as an instance of type `UntypedAST` (i.e. `Node<unit, unit>`) is the following:

```
{
  Pos = ... // Position in the input source file
  Env = ()
  Type = ()
  Expr = Add( {
    Pos = ... // Position in the input source file
    Env = ()
    Type = ()
    Expr = IntVal(42)
  },
  {
    Pos = ... // Position in the input source file
    Env = ()
    Type = ()
    Expr = Var("x")
  } )
}
```

The file `src/AST.fs` also defines two types called `PretypeNode` and `Pretype` that represent, respectively, the syntax tree node of a pretype, and the pretype itself from [Definition 1](#).

```
type PretypeNode =  
  {  
    Pos: Position    // Position of the pretype in the source file  
    Pretype: Pretype // Pretype contained in this Abstract Syntax Tree node  
  }  
  
and Pretype =  
  | TId of id: string // A type identifier
```

Note: This representation of pretypes with two data types may seem a bit redundant – but it will allow us to easily extend the Pretype definition (and the types supported by `hygge`) later in the course.

3.5 The Lexer and Parser

After we establish the *internal representation of the AST*, a logical next step in constructing a compiler is to develop the functions that builds such ASTs by reading some input text (i.e. the source code we wish to compile). To this purpose, we need to develop two components.

- A **lexer** (a.k.a. **tokenizer** or **scanner**) that reads the input text and classifies groups of characters that are either “useful” for building a syntax tree (e.g. identifiers, operators, parentheses...) or “useless” (e.g. sequences of white spaces, newlines, comments...). These groups of characters are typically recognised by matching them against a set of **regular expressions**; the goal of the lexer is to discard useless groups of characters, and turn the useful groups into **tokens** (or **lexemes**): each token captures a group of character and assigns to it a *lexical category*. The lexer should also report a **tokenization error** if it sees a sequence of input characters that it cannot classify.
- A **parser** that reads a stream of tokens produced by a lexer, and tries to match them against a set of **grammar rules**, thus producing a corresponding **syntax tree**. The parser should produce a **syntax error** if it is unable to match the input tokens with any of the grammar rules.

Building lexers and parsers is typically a routine job (unless the programming language we wish to parse has a peculiar syntax...), and is also very time-consuming. For these reasons, there is a wide variety of tools called **lexer generators** and **parser generators** – i.e. programs that:

1. read a configuration file containing a specification of the language we wish to compile (as a description of its tokens and/or grammar rules); and
2. generate the source code of a lexer and/or parser, based on such configuration file. The generated code can then be included and used as part of a compiler.

Here are some examples of lexer and parser generators.

- `Lex`¹⁶ and `Yacc`¹⁷ are standard tools for generating lexers/parser in C, available on all Unix-like operating systems. These tools were initially released in the 1970s!
- `Flex`¹⁸ and `GNU Bison`¹⁹ are more modern, improved replacements for `Lex` and `Yacc`.
- `FsLex` and `FsYacc`²⁰ are a lexer and parser generator for F#: they are used in `hyggec`.
- `ANTLR`²¹ can generate lexers/parsers in various programming languages, and is especially popular in the Java world.

In the following sections we will focus on `FsLex` and `FsYacc`, and how they are used in `hyggec`.

3.5.1 The Parser Configuration File `Parser.fsy` (Simplified)

The file `Parser.fsy` is a configuration file for the `FsYacc` parser generator. This file specifies the rules for transforming a sequence of tokens (obtained from the lexer, discussed *below*) into an AST.

The file `Parser.fsy` has the following structure: here we see a **very simplified** fragment that only accepts additions between integers and variables. (We will address these simplifications later, in *The Real Parser.fsy*).

```

1  %{
2  // Preamble with definitions of types and/or functions. The code appearing here
3  // will be placed on top of the generated parser source code.
4
5  // Auxiliary function to build an untyped AST node for a Hygge expression.
6  let mkNode (... , expr: UntypedExpr) : UntypedAST = // ...
7  %}
8
9  // Name of the grammar rule (defined below) to parse first.
10 %start program
11
12 // Declaration of tokens (values, operators, ...). These tokens are recognised
13 // by the lexer according to its configuration in Lexer.fsl.
14 %token <int> LIT_INT
15 %token PLUS
16 // ...
17
18 %token <string> IDENT // Generic identifier (might be a variable, pretype, etc.)
19 %token EOF // Signals the End-Of-File
20
21 %%

```

(continues on next page)

¹⁶ <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/lex.html>

¹⁷ <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/yacc.html>

¹⁸ <https://github.com/westes/flex>

¹⁹ <https://www.gnu.org/software/bison/>

²⁰ <https://fsprojects.github.io/FsLexYacc/>

²¹ <https://www.antlr.org/>

(continued from previous page)

```

22
23 // After '%' above, we specify the rules of the grammar. When a rule matches,
24 // it produces a value, which is computed by running the snippet of F# code next
25 // to the rule itself (between curly brackets). The snippet of code, in turn,
26 // can use the values produced when matching each symbol in its own rule, by
27 // referring to the symbol position in the rule ($1, $2, ...).
28
29 // Starting point: parsing rule for a whole Hygge program.
30 program:
31   | expr EOF { $1 } // A program is an expression followed by End-Of-File
32
33 expr:
34   | expr PLUS expr { mkNode(..., Expr.Add($1, $3)) }
35   | value          { mkNode(..., Expr.IntVal($1)) }
36   | variable       { mkNode(..., Expr.Var($1)) }
37   // ...
38
39 value:
40   | LIT_INT { $1 } // We just return the integer captured by token 'LIT_INT'
41   // ...
42
43 variable:
44   | ident { $1 }
45
46 ident:
47   | IDENT { $1 } // We just return the string captured by the token 'IDENT'

```

The parser configuration above declares, on lines 14–19, a few tokens:

- LIT_INT (literal integer), which also carries an integer value;
- PLUS, which carries no other value;
- IDENT (identifier), which also carries a string value;
- EOF, which carries no other value.

(We will see later, when discussing the *lexer*, that the values carried by the tokens LIT_INT and IDENT come from the Hygge0 source file we are compiling.)

Then, on lines 33–36, the sample of Parser.fsy above declares grammar rules for parsing an expression, which can be either:

- an expression followed by the token PLUS and another expression;
- a value, which is a token LIT_INT (lines 39–40);
- a variable, which is just an identifier (lines 43–44), which is just a token IDENT (lines 46–47).

(Notice the similarity between the grammar above and the one in *Definition 1*.)

The generated parser will read a stream of tokens — and whenever it matches one of the grammar rules above, it computes a result value by executing the code snippet next to

the rule itself (between curly brackets). Our goal is to set up such code snippets in a way that recursively builds the UntypedAST of the input source code, based on how the grammar rules are matched.

In the sample of `Parser.fsy` above, the code snippets in lines 34–36 create AST nodes via the auxiliary function `mkNode` (here omitted), which inserts the position of the parsed expression. The code snippets use the values produced when parsing each symbol in their own rule, by referring to symbol position in the rule (`$1`, `$2`, ...).

For instance, here is what happens when the parser is trying to parse an “`expr`”, according to its rules (lines 33–36).

- Line 34. When “`expr PLUS expr`” matches, the parser creates an UntypedAST containing an expression `Expr.Add($1, $3)` — where `$1` and `$3` are the results produced when parsing the `exprs` on the left and right of `PLUS` (hence, both `$1` and `$3` are UntypedASTs).
- Line 35. When “`value`” matches, the parser creates an AST node containing an expression `Expr.IntVal($1)`, where `$1` is the result produced when parsing the symbol `value`. To understand what is this result, we observe:
 - the rules for parsing `value` (lines 39–40) say that when `value` is parsed by matching a token `LIT_INT`, the result is `$1`, which is the value carried by the token itself.
- Line 36. When “`variable`” matches, the parser creates an AST node containing an expression `Expr.Var($1)`, where `$1` is the result produced when parsing the symbol `variable`. To understand what is this result, we observe:
 - the rules for parsing `variable` (lines 43–44) say that when `variable` is parsed by matching a symbol `ident`, we just return `$1`, which is the value produced when parsing `ident`;
 - * the rules for parsing `ident` (lines 46–47) say that they say that when `ident` is parsed by matching a token `IDENT`, the result is `$1`, which is the value carried by the token itself.

3.5.2 The Lexer Configuration File `Lexer.fsl`

The tokens declared in the *parser configuration file* are recognised by the lexer — and the lexer configuration file `Lexer.fsl` specifies how this is done.

The file `Lexer.fsl` has the following structure: here we see a simplified fragment that only recognises a few tokens.

```

1 {
2 // Preamble with definitions of types and/or functions. The code appearing here
3 // will be placed on top of the generated lexer source code.
4 }
5
6 // Regular expressions used in the token rules below
7 let letter = ['a'-'z'] | ['A'-'Z']

```

(continues on next page)

(continued from previous page)

```

8 let digit      = ['0'-'9']
9 let litInt    = digit+
10 let ident     = (letter | '_' ) (letter | '_' | digit)*
11 // ...
12
13 // We now define the rules for recognising the language tokens.  When a rule
14 // matches, it produces a token, which is computed by running the snippet of F#
15 // code next to the rule itself (between curly brackets).
16 rule tokenize = parse
17 // ...
18 | litInt  { Parser.LIT_INT(...) }
19 | "+"     { Parser.PLUS  }
20 | ident   { Parser.IDENT(...) }
21 | eof     { Parser.EOF } // End of File

```

The configuration above contains, on lines 18–21, a series of rules: they specify what the lexer should do whenever it sees a sequence of input characters that matches a certain string or regular expression. When a rule matches, the lexer produces a token, which is computed by running the snippet of F# code next to the rule itself (between curly brackets).

For example:

- line 16 says that if the input characters match the regular expression `litInt` (defined on line 9), then the lexer produces a token `LIT_INT` carrying the value matched by the regular expression (this part is omitted with “...”);
- line 17 says that if a “+” character is seen, then the lexer produces a token `PLUS`.

Note: The actual configuration file `Lexer.fs1` used by `hyggec` contains more rules (here omitted) that discard “useless” groups of characters. For example:

- whenever we see a white space or newline, we skip it, and continue lexing from the character that follows;
 - whenever we see “//” (the beginning of a comment) we skip all characters until we see an end-of-line, and continue lexing from the character that follows.
-

3.5.3 Example: the Lexer and Parser in Action

We now have all ingredients to understand how the `hyggec` lexer and parser examine an input file, and turn it into a sequence of tokens, and then into an AST. This is detailed in [Example 18](#).

Example 18 (Lexing and Parsing a Hygge0 Expression)

Let’s create a file called `test.hyg`, with the following content:


```
42 + x
```

When the `hyggec` lexer reads this file, it will process its contents character-by-character, grouping and classifying the characters according to the rules in *The Lexer Configuration File `Lexer.fsl`*. Therefore, the lexer will see the file contents as follows:

- 42, that matches the rule on line 18. Consequently, the lexer produces a token `LIT_INT` carrying the value 42;
- a white space, that is skipped;
- +, that matches the rule on line 19. Consequently, the lexer produces a token `PLUS`;
- a white space, that is skipped;
- x, that matches the rule on line 20; Consequently, the lexer produces a token `IDENT` carrying the value "x";
- the end of the file, which matches the rule on line 21. Consequently, the lexer produces a token `EOF`.

This sequence of tokens can be seen by running:

```
./hyggec tokenize test.hyg
```

The output is:

```
[LIT_INT 42; PLUS; IDENT "x"; EOF]
```

This sequence of tokens is then processed by the parser, according to *The Parser Configuration File `Parser.fsy (Simplified)`*. Therefore, the parser will try to parse the start symbol, called `program` (line 30); the only rule for `program` (line 31) requires to match an `expr` followed by an `EOF` token, and return the result `$1` (produced by parsing `expr`). Consequently, the generated parser will try to match the sequence of tokens `[LIT_INT 42; PLUS; IDENT "x"]` against `expr`, and produce an `UntypedAST` as a result.

[Fig.3.2](#) provides a visual intuition of how the tokens `[LIT_INT 42; PLUS; IDENT "x"]` are matched against `expr`, and how the instance of `UntypedAST` shown in [Example 17](#) is constructed:

- the solid arrows (going up) show how the grammar rules in *The Parser Configuration File `Parser.fsy (Simplified)`* are applied (such rule applications describe the **concrete syntax tree** of the input file);
- the dashed arrows (going down) show how the result of each rule is produced and propagated, building the **abstract syntax tree** instance shown in [Example 17](#) (of type `UntypedAST`).

More in detail, [Fig.3.2](#) depicts the following process:

- among the rules for parsing `expr` (lines 34–36), the input tokens `[LIT_INT 42; PLUS; IDENT "x"]` are only compatible with the rule “`expr PLUS expr`” (line 34), which:
 1. recursively matches `LIT_INT 42` as an `expr` via the rule on line 35, which:

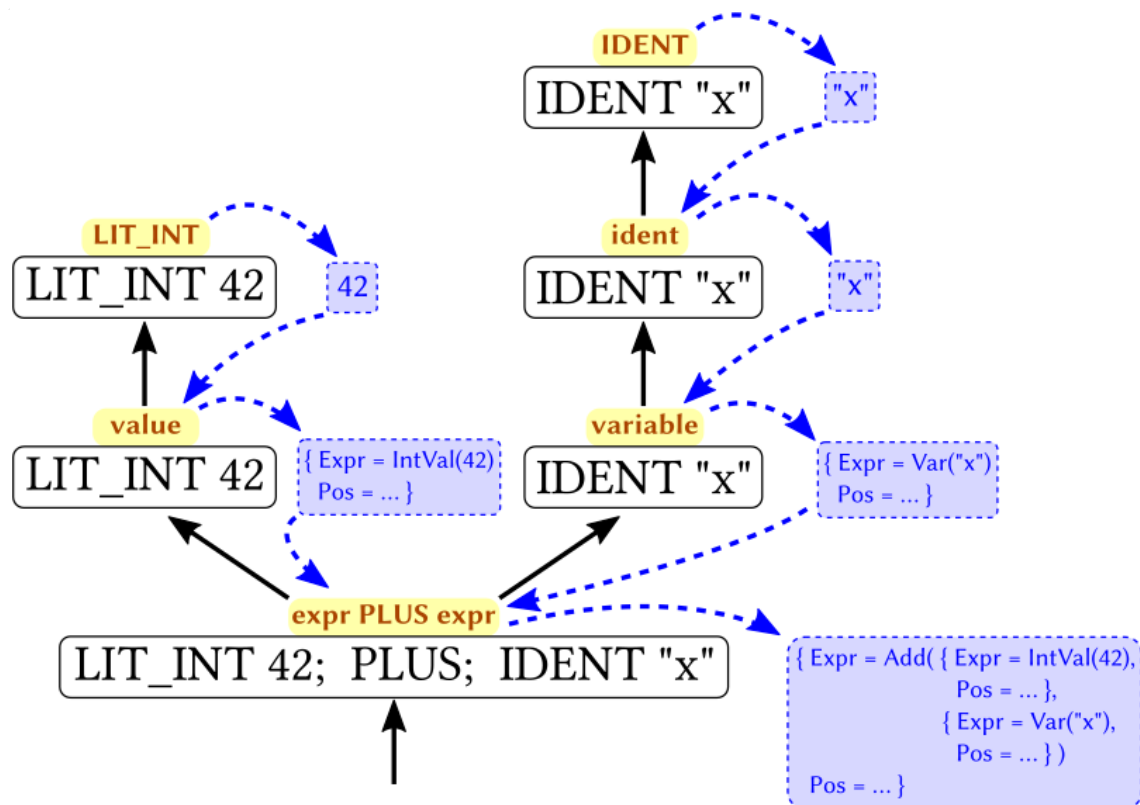


Fig. 3.2: How the parser generated from *The Parser Configuration File Parser.fsy (Simplified)* processes the sequence of tokens [LIT_INT 42; PLUS; IDENT "x"].

- recursively matches LIT_INT 42 as a value via the rule on line 40, which:
 - * matches the token LIT_INT 42, and
 - * produces 42
- produces the AST node containing the expression IntVal(42);
- 2. matches PLUS;
- 3. recursively matches IDENT "x" as an expr via the rule on line 36, which:
 - recursively matches IDENT "x" as a variable via the rule on line 44, which:
 - * recursively matches IDENT "x" as an ident via the rule on line 47, which:
 - matches token IDENT "x", and
 - produces "x"
 - * produces "x";
 - produces the AST node containing the expression Var("x");
- 4. finally, produces the AST node containing the expression Add(\$1, \$3), where \$1 and \$3 are the AST nodes produced by matching the sub-expressions. Therefore:

- \$1 is replaced by the AST node containing `IntVal(42)`, and
- \$3 is replaced by the AST node containing `Var("x")`.

The corresponding AST produced by `hygdec` can be seen by running:

```
./hygdec parse test.hyg
```

The output is the following (where the AST nodes show their position of their expression in `test.hyg`, ranging from their initial to their final line:column coordinates).

```
Add (1:1-1:6)
├─lhs: IntVal 42 (1:1-1:2)
└─rhs: Var x (1:6-1:6)
```

3.5.4 The Real Parser `.fsy`

To conclude this section about the `hygdec` lexer and parser, it is worth highlighting that *The Parser Configuration File `Parser.fsy` (Simplified)* shown above is very simplified. It conveys an intuition of how `FsYacc` works, but it has a crucial flaw: if implemented that way, `Parser.fsy` would describe an *ambiguous grammar*: see [Example 19](#) below.

Example 19 (Ambiguity of the Simplified Parser `.fsy`)

Consider an input file with the following content:

```
1 + 2 + 3
```

If we tokenize this file, we get the following sequence of tokens:

```
[LIT_INT 1; PLUS; LIT_INT 2; PLUS; LIT_INT 3; EOF]
```

If we pass such tokens to a parser generated from `Parser.fsy` as sketched in *The Parser Configuration File `Parser.fsy` (Simplified)*, that parser could build two possible syntax trees.

One syntax tree is obtained by treating `+` as a **left-associative** operator, i.e. by interpreting the input as $(1 + 2) + 3$:

```
Add
├─lhs: Add
│   └─lhs: IntVal 1
│       └─rhs: IntVal 2
└─rhs: IntVal 3
```

The other syntax tree is obtained by treating `+` as a **right-associative** operator, i.e. by interpreting the input as $1 + (2 + 3)$:

```

Add
├lhs: IntVal 1
└rhs: Add
    ├──lhs: IntVal 2
    └rhs: IntVal 3

```

The parser generator FsYacc would signal the ambiguity described in *Example 19* with obscure messages like:

```
shift/reduce error at state 47 on terminal XXXXX ...
```

For this reason, the real configuration file `Parser.fsy` breaks down the Hygge0 grammar into many rules, in order to enforce operator associativity and precedence, and remove any ambiguity. This follows the style of the [Java grammar specification](#)²² (but it is much simpler). For example, the rules for parsing additions and multiplications look as follows:

```

addExpr: // Additive expression
  | addExpr PLUS multExpr  { mkNode(..., Expr.Add($1, $3)) }
  | multExpr                { $1 }

multExpr: // Multiplicative expression
  | multExpr TIMES unaryExpr { mkNode(..., Expr.Mult($1, $3)) }
  | unaryExpr                { $1 }

unaryExpr: // ...

```

This makes addition left-associative, and forces the parser to only look for a multiplication after trying to parse an addition. The effect is that, when parsing e.g. `1 + 2 * 3 + 4`, we get the expected AST:

```

Add
├lhs: Add
|   ├──lhs: IntVal 1
|   └rhs: Mult
|       ├──lhs: IntVal 2
|       └rhs: IntVal 3
└rhs: IntVal 4

```

²² <https://docs.oracle.com/javase/specs/jls/se19/html/jls-19.html>

3.5.5 References and Further Readings

This tutorial²³ by Thanos Papathanasiou gives a nice overview of FsLex and FsYacc. (Note: the implementation uses an old version of .NET, but it also works on newer versions if you tweak its .fsproj file.)

FsYacc generates a parser based on the LALR algorithm. To know more about this parsing algorithm (and others), and understand the limitations of FsYacc and the meaning of its error messages, you can have a look at:

- Bill Campbell, Iyer Swami, Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. Chapman and Hall/CRC, 2012. Available on DTU Findit²⁴.
 - Chapter 3.4 (Bottom-Up Deterministic Parsing)

3.6 The Built-In Interpreter

hyggec includes an **interpreter** that can execute Hygge0 expressions (either typed or untyped). This interpreter is not necessary for developing a compiler: in fact, once hyggec has a typed AST (produced by *Typechecker.fs*), it can proceed directly to *Code Generation*. However, an interpreter can be handy for:

1. having a direct implementation of the *Formal Semantics of Hygge0*, to be used as a reference; and
2. testing whether the compiler respects the *Formal Semantics of Hygge0*: an expression e compiled and executed under RARS must produce the same computations and outputs observed when interpreting e .

The interpreter is implemented in the file `src/Interpreter.fs`, following the *Formal Semantics of Hygge0*. Most of its types and functions take two type arguments 'E and 'T: they have the same meaning discussed in *The AST Definition*, and being generic, they allow the interpreter to support both typed and untyped ASTs.

The type `RuntimeEnv` represents the runtime environment R in *Structural Operational Semantics of Hygge0*:

```
type RuntimeEnv<'E, 'T> = {
  Reader: Option<unit -> string> // Used to read a console input
  Printer: Option<string -> unit> // Used to perform console output
}
```

The function `reduce` is the core of `src/Interpreter.fs`, and it corresponds to the reduction in *Definition 4*. The function takes a runtime environment and an AST node, and attempts to perform one reduction step:

- if it succeeds, it returns `Some` with a pair consisting of a (possibly updated) runtime environment and reduced AST node;

²³ <https://thanos.codes/blog/using-fslexyacc-the-fsharp-lexer-and-parser/>

²⁴ <https://findit.dtu.dk/en/catalog/5c59eb2fd9001d01e4360926>

- if it cannot perform a reduction (because the AST node contains a value, or a stuck expression), it returns None.

```
let rec reduce (env: RuntimeEnv<'E, 'T>)
    (node: Node<'E, 'T>): Option<RuntimeEnv<'E, 'T> * Node<'E, 'T>> =
    match node.Expr with
    // ...
```

Each pattern matching case in reduce corresponds to a possible Hygge0 expression, and the function attempts to reduce the expression according to [Definition 4](#). For example, values cannot reduce (because there is no rule to let them reduce):

```
| UnitVal
| BoolVal(_)
| IntVal(_)
| FloatVal(_)
| StringVal(_) -> None

| Var(_) -> None
```

For an expression “assert(e)”, the function reduce follows rules [R-Assert-Eval-Arg] and [R-Assert-Res] in [Definition 4](#):

- if e is the boolean value true, it proceeds by reducing the whole expression to unit (according to rule [R-Assert-Res]);
- otherwise, it tries to recursively reduce e :
 - if e reduces into e' , then reduce returns an AST node with the updated expression e' , by rule [R-Assert-Eval-Arg] (notice that reduce creates the new AST node by copying and updating its original argument node);
 - if e cannot reduce, then e is stuck, and thus, reduce returns None (because the whole expression is stuck).

```
| Assertion(arg) ->
    match arg.Expr with
    | BoolVal(true) -> Some(env, {node with Expr = UnitVal})
    | _ ->
        match (reduce env arg) with
        | Some(env', arg') -> Some(env', {node with Expr = Assertion(arg')})
        | None -> None
```

For an expression “let $x : t = e; e_2$ ”, the function reduce follows rules [R-Let-Eval-Init] and [R-Let-Subst] in [Definition 4](#). It tries to recursively reduce e , and:

- if e can reduce into e' , it returns an AST node with the updated expression “let $x : t = e'; e_2$ ” (by rule [R-Let-Eval-Init]);
- if e cannot reduce because it is already a value v , it substitutes x with v in e' (by rule [R-Let-Subst]). In this case, reduce invokes `ASTUtil.subst`, which implements substitution according to [Definition 2](#);

- otherwise, e is stuck, and thus, reduce returns None (because the whole expression is stuck).

```

| Let(name, tpe, init, scope) ->
  match (reduce env init) with
  | Some(env', def') ->
    Some(env', {node with Expr = Let(name, tpe, def', scope)})
  | None when (isValue init) ->
    Some(env, {node with Expr = (ASTUtil.subst scope name init).Expr})
  | None -> None

```

Exercise 18

Consider the following groups of reduction rules from *Definition 4*:

- [R-Sub-L], [R-Sub-R], [R-Sub-Res]
- [R-Mul-L], [R-Mul-R], [R-Mul-Res]
- [R-Seq-Eval], [R-Seq-Res]
- [R-Type-Res]
- [R-Ascr-Res]
- [T-Print-Eval], [T-Print-Res],
- [R-Read-Int], [R-Read-Float]

For each group of reduction rules listed above:

1. find the case of the function reduce (in the file `src/Interpreter.fs`) that implements reductions for the corresponding Hygge0 expression (e.g. in the case of [T-Type-Res], find the case of reduce that handles the expression “type $x = t; e$ ”);
 2. identify how the premises and conditions of the reduction rules are checked in reduce; and
 3. identify how the expression returned by reduce corresponds to the reduced expression in the conclusion of the reduction rules.
-

Exercise 19

Consider the reduction rules you wrote when solving Exercise ???. For each of those reduction rules:

1. find the case of the function reduce (in the file `src/Interpreter.fs`) that implements reduction for the corresponding Hygge0 expression;
 2. identify how each premise and condition of your reduction rule is checked in reduce; and
 3. identify how the expression returned by reduce corresponds to the reduced expression in the conclusion of your reduction rule.
-

Do you see any discrepancy? If so, do you think there is a mistake in reduce, or in your reduction rule?

Tip: If you have not yet solved Exercise ??, you can proceed in the opposite direction: see how reduce handles a certain expression, and try to write down the corresponding reduction rule(s).

3.7 Types and Type Checking

The hyggec type checker is implemented in the file `src/Typechecker.fs`, and its goal is to inspect an UntypedAST (produced by *The Lexer and Parser*), and either:

- produce a TypedAST, where each AST node and expression has an associated type and typing environment (similarly to a typing derivation); or
- report **typing errors** pointing at issues in the input source program.

Consequently, we represent the result of type checking as a type called `TypingResult` (defined in `src/Typechecker.fs`). The definition of `TypingResult` uses the standard `Result` type provided by F#²⁵, so a typing result is either `Ok` with a TypedAST, or `Error` with a list of type errors:

```
type TypingResult = Result<TypedAST, TypeErrors>
```

As mentioned in *The AST Definition*, the type `TypedAST` above is just an alias for type `Node<TypingEnv, Type>`, where:

- the type `Type` (defined in the file `src/Type.fs`) is the internal representation of a Hygge0 type in the hyggec compiler, and follows *Definition 5*:

```
type Type =
  | TBool           // Boolean type.
  | TInt           // Integer type.
  | TFloat         // Floating-point type (single-precision).
  | TString        // String type.
  | TUnit          // Unit type.
  | TVar of name: string // Type variable.
```

- the type `TypingEnv` (defined in the file `src/Typechecker.fs`) is the internal representation of a Hygge0 typing environment in the hyggec compiler, and follows *Definition 6*:

```
type TypingEnv = {
  Vars: Map<string, Type> // Variables in the current scope, with their
  ↪ type
```

(continues on next page)

²⁵ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/results>

(continued from previous page)

```

TypeVars: Map<string, Type> // Type vars in current scope, with their
↳def.
}

```

3.7.1 Type Checking or Type Inference?

Before proceeding, it is worth noticing that here (as in most literature about compilers) we often use the term “type checking” in a broad sense, meaning “analysing a source program to see whether it is well-typed”. But more precisely, to implement hyggec we need to solve a **type inference** problem, according to *Definition 12* below.

Definition 12 (Type Checking vs. Type Inference)

Take a set of rules defining a typing judgement $\Gamma \vdash e : T$ (such as the rules in *Definition 8*).

- A **type checking problem** has the following form:
 - given a typing environment Γ , an expression e , and a type T , construct a typing derivation that proves $\Gamma \vdash e : T$
 - A **type inference problem** has the following form:
 - given a typing environment Γ and an expression e , **find a type** T for which we can construct a typing derivation that proves $\Gamma \vdash e : T$
-

Luckily, the Hygge0 typing rules in *Definition 8* directly suggest us how to implement a type inference algorithm, because the rules are **syntax-driven**: just by looking at the shape of an expression e , and reading the rules “bottom-up” (from the conclusion to the premises) we can determine which rule(s) could possibly be used to type e , and what additional checks they require. For instance, suppose we are trying to infer the type of a Hygge0 expression e :

- if e is an integer value 42, we can only type it by rule [T-Val-Int], hence we immediately infer that e has type int;
- if e is a variable x , we can only type it by rule [T-Var], hence we infer that e must have the type contained in $\Gamma.\text{Vars}(x)$. If x is not in $\Gamma.\text{Vars}$, there is no other typing rule we can try, so we report a typing error;
- if e is a logical negation “not e' ”, then we can only type it by trying the typing rule you wrote as part of Exercise ???. Therefore, we recursively infer the type of e' , and check whether e' has type bool. If this is the case, we infer that e has type bool; otherwise, there is no other typing rule we can try, so we report a typing error;
- if e is an addition “ $e_1 + e_2$ ”, then we can only type it by trying rule [T-Add]. Therefore, we recursively infer the types of e_1 and e_2 , and check whether:

- both e_1 and e_2 have type `int` – and in this case, we infer that e has type `int`;
or
- both e_1 and e_2 have type `float` – and in this case, we infer that e has type `float`.

If this doesn't work, there is no other typing rule we can try, so we report a typing error;

- if e is a conditional “if e_1 then e_2 else e_3 ”, we can only type it by trying rule [T-Cond]. Therefore, we recursively infer the types of e_1 , e_2 , and e_3 , and check whether:
 - e_1 has type `bool`, and
 - e_1 and e_2 have a same type T – and in this case, we infer that e has that type T .

If this doesn't work, there is no other typing rule we can try, so we report a typing error.

3.7.2 Implementation of `src/Typechecker.fs`

The core of `src/Typechecker.fs` is a function called `typer`, which implements *the type inference algorithm discussed above*. The function `typer` has the following definition: it takes a typing environment and an AST node, performs a pattern matching against the expression contained in the AST node, and returns a `TypingResult`.

```
let rec typer (env: TypingEnv) (node: UntypedAST): TypingResult =
  match node.Expr with
  // ...
```

The function `typer` is initially called with an empty environment, and with the whole AST of the Hygge0 program being compiled. While running, `typer` recursively traverses the AST, adding information to the environment, as required by the typing rules in *Definition 8*.

Each case in `typer` takes the given AST node (which is untyped), and (if type inference succeeds) produces a *typed* AST node containing the current environment, and the inferred type. Otherwise, `typer` returns a list of typing errors.

For example, `typer` assigns types to boolean or integer values, as follows:

```
| BoolVal(v) ->
  Ok { Pos = node.Pos; Env = env; Type = TBool; Expr = BoolVal(v) }
| IntVal(v) ->
  Ok { Pos = node.Pos; Env = env; Type = TInt; Expr = IntVal(v) }
```

When typing a variable, `typer` checks whether the variable is in the typing environment, and assigns the type it finds there to the variable – or reports an error:

```

| Var(name) ->
  match (env.Vars.TryFind name) with
  | Some(tpe) ->
    Ok { Pos = node.Pos; Env = env; Type = tpe; Expr = Var(name) }
  | None ->
    Error([(node.Pos, $"undefined variable: %s{name}")]])

```

When typing a logical “not” expression, typer recursively infers the type of the argument, and checks whether it is a subtype of bool: if so, typer infers that the whole expression has type bool; otherwise, it reports type errors.

```

| Not(arg) ->
  match (typer env arg) with
  | Ok(targ) when (isSubtypeOf env targ.Type TBool) ->
    Ok { Pos = node.Pos; Env = env; Type = TBool; Expr = Not(targ) }
  | Ok(arg) ->
    Error([(node.Pos, $"expected %0{TBool} arg, found %0{arg.Type}")]])
  | Error(es) -> Error(es)

```

Important: In the last example above (3rd line) we are using `isSubtypeOf` (defined in `src/Typechecker.fs`) to compare the types `targ.Type` and `TBool`. This check corresponds to an application of the subsumption rule in [Definition 11](#): we want `targ` to have type `bool`, so it is OK if `targ` has a *subtype* of `bool` (e.g. via some type alias).

Instead of using `isSubtypeOf`, in the code above we could have simply written `targ.Type = TBool`: this would work, but it would also make our typing system less flexible, leading to limitations similar to those discussed in [Example 15](#).

Therefore, the rule of thumb is: **whenever typer needs to compare two types, it should use `isSubtypeOf`** (instead of plain equality `=`).

One may ask: *instead of checking subtyping in multiple places, can we implement a general type inference case based on the subsumption rule [T-Sub] in [Definition 11](#)?*

The answer, unfortunately, is no — and the reason is that (unlike the typing rules in [Definition 8](#)) rule [T-Sub] is *not syntax-driven*: it can be applied to *any* expression e . Therefore, we need to explicitly check subtyping whenever we compare types.

The rest of the cases of `typer` work similarly. Two things to mention:

- in the cases for “let $x : t = e_1; e_2$ ” and “type $x = t; e$ ”, `typer` need to check whether the pretype t corresponds to some valid type T . To this end, `typer` uses the function `resolvePretype`, which corresponds to the type resolution judgement in [Definition 7](#);
- in various cases, `typer` uses auxiliary functions to avoid code duplication. For instance: “ $e_1 + e_2$ ” and “ $e_1 * e_2$ ” are typed in a similar way (see rule [T-Add] in [Definition 8](#), and Exercise ??), and thus, `typer` uses a function called `binaryNumericalOpTyper` to handle both.

Example 20 (Untyped vs. Typed ASTs)

To see the difference between the untyped AST and the typed AST of a Hygge0 program, you can try to parse the example in [Example 3](#):

```
./hyggec parse examples/hygge0-spec-example.hyg
```

And then compare the untyped AST above with the typed AST produced by `src/Typechecker.fs`:

```
./hyggec typecheck examples/hygge0-spec-example.hyg
```

Exercise 20

Consider the following typing rules from [Definition 8](#):

- [T-Seq]
- [T-Assert]
- [T-Ascr]
- [T-Print]
- [T-Let]
- [T-Type]

For each typing rule listed above:

1. find the case of the function `typer` (in the file `src/Typechecker.fs`) that implements type inference for the corresponding Hygge0 expression (e.g. in the case of [T-Seq], find the case of `typer` that handles the expression $e_1; e_2$);
 2. identify how each premise of the typing rule is checked in `typer`; and
 3. identify how the type inferred by `typer` matches the type expected in the conclusion of the typing rule.
-

Exercise 21

Consider the typing rules you wrote when solving Exercise ??. For each one of those typing rules:

1. find the case of the function `typer` (in the file `src/Typechecker.fs`) that implements type inference for the corresponding Hygge0 expression;
 2. identify how each premise of your typing rule is checked in `typer`; and
 3. identify how the type inferred by `typer` matches the type expected in the conclusion of your typing rule.
-

Do you see any discrepancy? If so, do you think there is a mistake in typer, or in your typing rule?

Tip: If you have not yet solved Exercise ??, you can proceed in the opposite direction: see how typer handles a certain expression, and try to write down a corresponding typing rule.

3.8 Code Generation

The code generation of hyggec is implemented in the file `src/RISCVCodegen.fs`. We illustrate its contents when discussing the *Code Generation Strategy* — but first, we have a look at the *RISC-V Code Generation Utilities*.

3.8.1 RISC-V Code Generation Utilities

The file `src/RISCV.fs` contains various data types and functions to represent RISC-V assembly code, and manipulate assembly code snippets. The hyggec compiler goal is to produce text output containing RISC-V assembly — and this internal representation of the output code makes the job simpler, and prevents possible mistakes.

The key components of `src/RISCV.fs` are the following (we will see them in use in the *Code Generation Strategy*).

- The type `RV` (standing for “RISC-V”) represents a **statement in a RISC-V assembly program**: it is a discriminated union with one named case for each supported RISC-V instruction (e.g. the RISC-V instruction `mv` is represented by the named case `RV.MV(...)`). It also includes named cases for representing labels and comments in RISC-V assembly.
- The types `Reg` and `FReg` represent, respectively, *a base integer register and a floating-point register*. Their purpose is twofold:
 1. they help ensuring that the RISC-V instructions in `RV` above can only be used with registers that exist, and have the correct type (otherwise, the F# compiler will report a type error). For example:
 - we cannot use the instruction `RV.MV(...)` on a floating-point register — if we try, we get an F# type error;
 - to move a value from register `t0` to `t1`, we can write `RV.MV(Reg.t1, Reg.t0)` — and if there is a typo (e.g. if we write “`RV.MV(Reg.t1, Reg.y0)`”) it will be spotted by the F# compiler;
 2. they provide **generic numbered registers** `Reg.r(n)` and `FReg.r(n)`, which range over all “*temporary*” and “*saved*” RISC-V registers. For example, to move

a value from generic register number $n + 1$ to n , one can write: `RV.MV(Reg.r(n), Reg.r(n+1))`. This greatly simplifies the handling of registers during code generation.

- The type `Asm` represents an *assembly program* with its `.data` and `.text` segments. The main features are:
 - the methods `addData` and `addText`, which allow us to **add memory allocations and instructions** in the selected memory segment; and
 - the method `++`, which allows us to **combine two assembly programs** (e.g. produced during code generation) into a unique, well-formed assembly program.

3.8.2 Code Generation Strategy

The core of the file `src/RISCVCodegen.fs` is the function `doCodegen`, which takes a **code generation environment** and a typed AST node, and produces assembly code. Correspondingly, the declaration of `doCodegen` has the following types.

```
let rec doCodegen (env: CodegenEnv) (node: TypedAST): Asm = // ...
```

The function `doCodegen` uses a very simple code generation strategy. In a nutshell, when compiling an expression e :

- after e is computed, its result is written in a **target register number** n ;
- if the computation of e requires the results of other sub-expressions, `doCodegen` recursively compiles each sub-expression by increasing (if necessary) its target register number to $n + 1$, $n + 2$, etc.

Important: This compilation strategy only works if the code produced by each `doCodegen` recursive call *never* modifies any register below its current target.

With this approach, the code generation environment (of type `CodegenEnv`) used by `doCodegen` must contain two pieces of information:

1. which **target register number** should be used to compile the current expression. More precisely, we need one target for integer expressions, and another target for floating-point expressions; and
2. a “**storage**” **mapping** from known variable names, to the location where the value of each variable is stored (e.g. in a register, or in memory).

Consequently, the `CodegenEnv` type is a record that looks as follows:

```
type CodegenEnv = {  
    Target: uint           // Target register for the result of integer expressions  
    FPTarget: uint        // Target register for the result of float expressions  
    VarStorage: Map<string, Storage> // Storage info about known variables  
}
```

The type `Storage` used by `CodegenEnv` tells us whether the value of a variable is stored in an integer register, or in a floating-point register, or in a memory location marked with a label in the generated assembly code.

```
type Storage =
  | Reg of reg: Reg      // The value is stored in an integer register
  | FReg of freg: FReg  // The value is stored in a float register
  | Label of label: string // Value is stored in memory, with a label
```

3.8.3 A Tour of doCodegen

We now have all ingredients to examine how `doCodegen` works. Its implementation is a pattern matching on the expression contained in the AST node being compiled.

```
let rec doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  match node.Expr with
  // ...
```

Here are some examples of how `doCodegen`'s main pattern matching handles various kinds of expressions.

An integer value is immediately loaded into the target register (using the assembly instruction `li`).

```
| IntVal(v) ->
  Asm(RV.LI(Reg.r(env.Target), v))
```

Example 21 (Compiling an Integer)

If we compile the Hygge0 expression 42, the match case for `IntVal` above is executed, and the output of the compiler is:

```
.data:

.text:
  li t0, 42          # <-- Produced by match case for IntVal in doCodegen
  li a7, 10 # RARS syscall: Exit
  ecall # Successful exit with code 0
```

A string value is added to the `.data` segment of the generated assembly code, with its memory address marked by a unique label (generated with the function `Util.genSymbol`). Then, the memory address is loaded into the target register (using the assembly instructions `la`).

```
| StringVal(v) ->
  let label = Util.genSymbol "string_val"
  Asm().AddData(label, Alloc.String(v))
  .AddText(RV.LA(Reg.r(env.Target), label))
```

Example 22 (Compiling a String)

If we compile the Hygge0 expression "Hello, World!", the match case for StringVal above is executed, and the output of the compiler is:

```
.data:
string_val:
    .string "Hello, World!" # <-- Produced by case for StringVal in doCodegen

.text:
    la t0, string_val # <-- Produced by match case for StringVal in doCodegen
    li a7, 10 # RARS syscall: Exit
    ecall # Successful exit with code 0
```

When compiling a variable x , doCodegen produces code to access the value of the variable, depending on where it is stored: to this end, it inspects the VarStorage mapping in the code generation environment. (For clarity of exposition, the code snippet below omits some cases that are present in the implementation).

```
| Var(name) ->
    match node.Type with // Inspect the var type and where it is stored
    | t when (isSubtypeOf node.Env t TFloat) ->
        match (env.VarStorage.TryFind name) with
        | Some(Storage.FPReg(fpreg)) ->
            Asm(RV.FMV_S(FPReg.r(env.FPTarget), fpreg),
                $"Load variable '{s{name}}'")
        | _ -> failwith $"BUG: float var with bad storage: {s{name}}"

    | _ -> // Default case for variables holding integer-like values
        match (env.VarStorage.TryFind name) with
        | Some(Storage.Reg(reg)) ->
            Asm(RV.MV(Reg.r(env.Target), reg), $"Load variable '{s{name}}'")
        | _ -> failwith $"BUG: variable without storage: {s{name}}"
```

When compiling an expression “let $x : t = e_1; e_2$ ” with target register n , doCodegen proceeds as follows:

- recursively generates assembly code for e_1 , targeting the register n ;
- adds the variable x to the env.VarStorage mapping, assigning it to register n (which contains the result of e_1);
- compiles e_2 with the updated env.VarStorage (containing x), targeting register $n + 1$ (because e_2 may use x , and thus the register n);
- copies the result of e_2 from register $n + 1$ to n (thus overwriting the value of x , which is going out of scope).

```
| Let(name, _, init, scope) ->
    let initCode = doCodegen env init // 'let...' initialisation asm code
```

(continues on next page)

(continued from previous page)

```

match init.Type with
| t when (isSubtypeOf init.Env t TFloat) ->
    // ** Omitted code similar to the following, using float registers

| _ ->    // Default case for integer-like initialisation expressions
    let scopeTarget = env.Target + 1u // Target reg. for 'let' scope
    let scopeVarStorage = // Var storage for compiling 'let' scope
        env.VarStorage.Add(name, Storage.Reg(Reg.r(env.Target)))
    let scopeEnv = { env with Target = scopeTarget
                    VarStorage = scopeVarStorage }

    initCode
        ++ (doCodegen scopeEnv scope)
            .AddText(RV.MV(Reg.r(env.Target), Reg.r(scopeTarget)),
                    "Move 'let' scope result to target register")

```

When compiling an addition “ $e_1 + e_2$ ” with target register n , doCodegen proceeds as follows:

1. recursively generates the assembly code for e_1 , targeting the register n ;
2. recursively generates the assembly code for e_2 , targeting the register $n + 1$;
3. generates a RISC-V addition operation that adds the contents of registers n and $n + 1$, and overwrites register n with the result.

The resulting code looks, intuitively, as follows.

```

| Add(lhs, rhs)
    let lAsm = doCodegen env lhs // Generated code for the lhs expression

    match node.Type with // Generated code depends on the type of addition
    | t when (isSubtypeOf node.Env t TInt) ->
        let rtarget = env.Target + 1u // Target register for rhs expression
        let rAsm = doCodegen {env with Target = rtarget} rhs // Asm for rhs
        let opAsm = // Generated code for the addition operation
            Asm(RV.ADD(Reg.r(env.Target),
                      Reg.r(env.Target), Reg.r(rtarget)))
        lAsm ++ rAsm ++ opAsm // Put asm code together: lhs, rhs, operation

    | t when (isSubtypeOf node.Env t TFloat) ->
        // ** Omitted code similar to above, with float regs and instructs

    | t ->
        failwith $"BUG: addition codegen invoked on invalid type %0{t}"

```

Note: In the actual hyggec implementation is a bit different: since addition “ $e_1 + e_2$ ” and multiplication “ $e_1 * e_2$ ” generate very similar code, the pattern matching case above handles both `Add(lhs, rhs)` and `Mult(lhs, rhs)` — and in case of multiplication, it produces the RISC-V instruction `mul` (instead of `add`).

Example 23 (Compiling a “Let...” and an Addition)

Consider the following Hygge0 program:

```
let x: int = 42;
x + 3
```

Its typed AST looks as follows: (we omit Env.TypeVars for brevity)

```
Let x (1:1-2:5)
├Env.Vars: ∅
├Type: int
├Ascription: Pretype Id "int"; pos: (1:8-1:10)
├init: IntVal 42 (1:14-1:15)
├
├├Env.Vars: ∅
├├└Type: int
├└scope: Add (2:1-2:5)
├├├Env.Vars: Map
├├├├└x: int
├├├├Type: int
├├├├lhs: Var x (2:1-2:1)
├├├├├Env.Vars: Map
├├├├├└x: int
├├├├├└Type: int
├├├├├rhs: IntVal 3 (2:5-2:5)
├├├├├├Env.Vars: Map
├├├├├├└x: int
├├├├├├└Type: int
```

When hyggec compiles this program, it produces the following RISC-V assembly code.

```
.data:

.text:
    li t0, 42                # <-- Produced by case for IntVal in doCodegen
    mv t1, t0 # Load variable 'x' # <-- Produced by case for Var in doCodegen
    li t2, 3                 # <-- Produced by case for IntVal in doCodegen
    add t1, t1, t2           # <-- Produced by case for Add in doCodegen
    mv t0, t1 # Move 'let' scope result to 'let' target register
    li a7, 10 # RARS syscall: Exit
    ecall # Successful exit with code 0
```

Note: The function doCodegen includes more cases, which follow the explanations above. The only exceptions are those that require RARS system calls (e.g. Print, ReadInt: their code generation makes use of functions that save register values on the stack, and restore registers values from the stack. We will address these aspects later in the course.

Important: The *Code Generation Strategy* illustrated in this module is quite simple, and it has a relevant flaw: its naive **register allocation** policy tends to increase the target register number for each sub-expression being compiled – and in some cases, it may run out of available registers. When this happens, the hyggec compiler crashes, reporting a “BUG”.

Luckily, it takes some effort to write a Hygge0 program that triggers this limitation: therefore, it is unlikely that, by chance, you will write a program that stumbles into the issue. (See [Exercise 22](#) below.)

We will consider better register allocation strategies later in the course.

Exercise 22 (Running Out of Registers)

Write a Hygge0 expression that, by the *Code Generation Strategy*, causes doCodegen to run out of registers, and makes hyggec crash.

Hint: There is a solution that only uses +, some integer values, and parentheses...

3.9 The Test Suite of hyggec

We conclude this overview by mentioning the hyggec test suite, which is designed to encourage frequent testing of the compiler. To launch the test suite, simply invoke:

```
./hyggec test
```

Or, equivalently:

```
dotnet test
```

When running, the testing framework explores the tree of directories under tests/, and uses each file with extension .hyg as a test case. The outcome of the test depends on the position of the .hyg file in the tests/ directory tree, according to the following table. (Note that the .hyg files under the fail/ directories are expect to cause some error, in specific ways.)

Table 3.2: Overview of the hyggec directory tree for tests.

Directory under tests/	A .hyg file under this directory is a passed test if...
lexer/pass/	The tokenization succeeds.
lexer/fail/	The tokenization fails with a lexer error.
parser/pass/	Parsing succeeds.
parser/fail/	Tokenization succeeds, but parsing fails.

continues on next page

Table 3.2 – continued from previous page

Directory under tests/	A .hyg file under this directory is a passed test if...
interpreter/pass/	Tokenization and parsing succeed, and the interpreter reduces the program into a value.
interpreter/fail/	Tokenization and parsing succeed, but the interpreter reaches a stuck expression (e.g. <code>assert(false)</code>).
typechecker/pass/	Tokenization, parsing, and type checking succeed.
typechecker/fail/	Tokenization and parsing succeed, but type checking fails.
codegen/pass/	Tokenization, parsing, and type checking succeed, and the generated RISC-V assembly program runs under RARS and terminates successfully (exit code 0).
codegen/fail/	Tokenization, parsing, and type checking succeed, and the generated RISC-V assembly program runs under RARS, but it terminates with an assertion violation (<code>assert(false)</code>).

3.10 Example: Extending Hygge0 and hygdec with a Subtraction Operator

The following sections show how to extend the Hygge0 language and the hygdec compiler with a subtraction operator, in 8 steps:

1. Defining the *Formal Specification of Subtraction*
2. *Extending the AST*
3. *Extending the Pretty Printer*
4. *Extending the Lexer*
5. *Extending the Parser*
6. *Extending the Interpreter*
7. *Extending the Type Checker*
8. *Extending the Code Generation*

To perform these steps, we will use as a reference the most similar expression that is already supported by Hygge0 and hygdec – i.e. the addition $e_1 + e_2$.

3.10.1 Formal Specification of Subtraction

Before jumping to the implementation, we specify the formal syntax, semantics, and typing rules of the new subtraction operator.

First, we specify the **syntax** of subtraction expressions by extending the grammar rules of Hygge0 in *Definition 1*. We add a new rule:

$$\begin{array}{l} \text{Expression } e ::= \dots \\ \quad | \quad e_1 - e_2 \quad (\text{Subtraction}) \end{array}$$

Then, we specify the **semantics** of subtraction expressions, in two steps.

1. We specify how to **substitute** variable x with expression e' inside a subtraction $e_1 - e_2$, by extending *Definition 2* with a new case (similar to the existing case for addition):

$$(e_1 - e_2)[x \mapsto e'] = e_1[x \mapsto e'] - e_2[x \mapsto e']$$

2. We extend the **reduction rules** in *Definition 4* with new rules for subtraction expressions (very similar to the existing rules for addition):

$$\begin{array}{c} \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e - e_2 \rangle \rightarrow \langle R' \bullet e' - e_2 \rangle} \quad [\text{R-Sub-L}] \quad \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v - e \rangle \rightarrow \langle R' \bullet v - e' \rangle} \quad [\text{R-Sub-R}] \\ \frac{v_1 - v_2 = v_3}{\langle R \bullet v_1 - v_2 \rangle \rightarrow \langle R \bullet v_3 \rangle} \quad [\text{R-Sub-Res}] \end{array}$$

Finally, we extend the **typing rules** in *Definition 8* with a new rule for subtraction expressions (very similar to the existing typing rule for addition):

$$\frac{T \in \{\text{int}, \text{float}\} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 - e_2 : T} \quad [\text{T-Subtract}]$$

3.10.2 Extending the AST

We can now implement the *Formal Specification of Subtraction*. The first step is to extend the AST, by extending the type `Expr<'E, 'T>` (in the file `src/AST.fs`) with a new case:

```
and Expr<'E, 'T> = // ...
  /// Subtraction between lhs and rhs.
  | Sub of lhs: Node<'E, 'T>
      * rhs: Node<'E, 'T>
```

Tip: By extending the type `Expr<'E, 'T>`, we will cause various warning in several source files, with messages like:

```
Incomplete pattern matches on this expression. For example, the value 'Sub (_, _)
↪' may indicate a case not covered by the pattern(s)
```

These warnings highlight the parts of the hyggec source code we need to adjust to support the new AST case we have just added.

To see all such warnings on the console, we can rebuild hyggec by running:

```
dotnet clean
```

followed by

```
dotnet build
```

3.10.3 Extending the Pretty Printer

hyggec needs to know how to display the new subtraction expression when printing an AST on screen. To this purpose, we add a new pattern matching case to the function `formatASTRec` (in the file `src/PrettyPrinter.fs`) to support the new case `Sub(lhs, rhs)` (we can copy and adapt the existing case for `Add(lhs, rhs)`):

```
| Sub(lhs, rhs) ->
    mkTree "Sub" node [("lhs", formatASTRec lhs)
                      ("rhs", formatASTRec rhs)]
```

We will see the effect of this change shortly, when *Testing the Parser*.

3.10.4 Extending the Lexer

We can now add a new rule to the lexer, to recognise the new token for the subtraction symbol “-”. In the file `src/Lexer.fs1`, we add the following line (e.g. nearby the definition for the token for “+”):

```
| "+"          { Parser.PLUS }
| "-"          { Parser.MINUS } // <-- We add this line
```

Correspondingly, we must add declaration of `Parser.MINUS` in the file `src/Parser.fsy`. For instance, we can add `MINUS` nearby the other arithmetic operators:

```
// Tokens for arithmetic operators
%token TIMES PLUS MINUS // <-- We add "MINUS" here
```

Testing the Lexer

To see whether the lexer recognises the new token, we can add a new test. For example, we can add a file called `tests/lexer/pass/003-minus.hyg` containing just a minus sign:

```
-
```

And now, if we run

```
./hygdec tokenize tests/lexer/pass/003-minus.hyg
```

we should see (besides the warnings about incomplete pattern matches):

```
[MINUS; EOF]
```

Moreover, running `hygdec test` should not report any failed test.

3.10.5 Extending the Parser

We can now add a new grammar rule for subtraction to the file `src/Parser.fsy`. Since subtraction should have the same operator precedence of addition, we can look for the rule for addition, and place the new rule in the same group of “additive expressions”:

```
// Additive expression
addExpr:
| addExpr PLUS multExpr { mkNode(parseState, 2, Expr.Add($1, $3)) }
| addExpr MINUS multExpr { mkNode(parseState, 2, Expr.Sub($1, $3)) }
| multExpr { $1 }
```

Important: Each time we modify the grammar rules of `src/Parser.fsy`, we should check whether we have introduced any parsing issue. To this purpose, we should run:

```
dotnet build
```

The messages produced by `FsYacc` should look like:

```
computing first function...          time: 00:00:00.0571261
building kernels...                  time: 00:00:00.0253194
building kernel table...             time: 00:00:00.0077130
computing lookahead relations.....
↔.....                               time: 00:00:00.0250216
building lookahead table...          time: 00:00:00.0066406
building action table...             time: 00:00:00.0114085
    building goto table...           time: 00:00:00.0022120
    returning tables.
writing tables to log
    building tables
    94 states
```

(continues on next page)

(continued from previous page)

```

19 nonterminals
36 terminals
47 productions
#rows in action table: 94

```

The output above has no errors, and this means that `src/Parser.fsy` is correct.

Instead, if we see an error message like the following, then our change to `src/Parser.fsy` has introduced a problem (probably a grammar ambiguity) and we should revise our change.

```
shift/reduce error at state 47 on terminal PLUS between...
```

Testing the Parser

To see whether the parser recognises subtraction expressions, we can add a new test. For example, we can add a file called `tests/parser/pass/011-minus.hyg` containing:

```
42 - x
```

And now, if we run

```
./hygdec parse tests/parser/pass/011-minus.hyg
```

we should see (besides the warnings about incomplete pattern matches) a representation of the AST node for `Sub` (with the formatting we added by *Extending the Pretty Printer*):

```

Sub (1:1-1:6)
├lhs: IntVal 42 (1:1-1:2)
└rhs: Var x (1:6-1:6)

```

If the type of the subtraction does not match the type ascription, `hygdec` will report a typing error. After we add this test, running `./hygdec test` should not report any failed test.

3.10.6 Extending the Interpreter

In the *Formal Specification of Subtraction*, we have defined new reduction rules that are quite similar to those for addition. Correspondingly, we can extend the `hygdec` interpreter by adapting existing cases for addition, in two steps:

1. we extend the function `subst` in `src/ASTUtil.fs` by adding a new case for `Sub(lhs, rhs)` (adapted from the existing case for `Add(lhs, rhs)`):

```

| Sub(lhs, rhs) ->
  {node with Expr = Sub((subst lhs var sub), (subst rhs var sub))}

```


2. we extend the function `reduce` in `src/Interpreter.fs` by adding a new case for `Sub(lhs, rhs)` (adapted from the existing case for `Add(lhs, rhs)`):

```

| Sub(lhs, rhs) ->
    match (lhs.Expr, rhs.Expr) with
    | (IntVal(v1), IntVal(v2)) -> Some(env, {node with Expr = IntVal(v1
↵- v2)})
    | (FloatVal(v1), FloatVal(v2)) -> Some(env, {node with Expr =
↵FloatVal(v1 - v2)})
    | (_, _) ->
        match (reduceLhsRhs env lhs rhs) with
        | Some(env', lhs', rhs') -> Some(env', {node with Expr = Sub(lhs
↵', rhs')})
        | None -> None

```

Testing the Interpreter

To test whether the interpreter handles subtraction expressions correctly, we can check whether the result of a subtraction is the value we expect. We can obtain this by writing a test case called e.g. `tests/interpreter/pass/008-sub.hyg`, with the following content:

```

assert(42 - 10 = 32);
assert(3.14f - 1.0f = 2.14f) // Careful when comparing floats! This case is OK

```

If the any of the comparisons inside the assertions is false, the interpreter will reach a stuck expression `assert(false)` and report an error. After we add this test, running `./hygdec test` should not report any failed test.

3.10.7 Extending the Type Checker

In the *Formal Specification of Subtraction*, we have defined new typing rule [T-Subtract] that is quite similar to rule [T-Add] in *Definition 8*. Correspondingly, we can extend the `hygdec` function `typer` (in the file `src/Typechecker.fs`) with a new pattern matching case, based on the existing case for `Add(lhs, rhs)`:

```

| Sub(lhs, rhs) ->
    match (binaryNumericalOpTyper "subtraction" node.Pos env lhs rhs) with
    | Ok(tpe, tlhs, trhs) ->
        Ok { Pos = node.Pos; Env = env; Type = tpe; Expr = Sub(tlhs, trhs) }
    | Error(es) -> Error(es)

```

Testing the Type Checker

To test whether the type checking for subtraction expressions works as intended, we can check whether the subtraction of two integers has type `int`, and whether the subtraction of two floating-point values has type `float`. We can obtain this by writing a test case called e.g. `tests/typechecker/pass/011-sub.hyg`, with the following content:

```
(2 - 1): int;
(3.14f - 1.0f): float
```

3.10.8 Extending the Code Generation

Code generation for subtraction is very similar to addition (and also multiplication): the only difference is that we need to emit the RISC-V assembly instruction `sub` (for integers) or `fsub.s` (for floating point values). Consequently, it is enough to edit the function `doCodegen` (in the file `src/RISCVCodegen.fs`) and find the pattern matching case for `Add(lhs, rhs)` (and also `Mult(lhs, rhs)`) and apply the following three changes:

1. we extend the pattern matching case to also cover `Sub(lhs, rhs)`:

```
| Add(lhs, rhs)
| Sub(lhs, rhs) // <-- We add this line
| Mult(lhs, rhs) as expr ->
```

2. we find the internal pattern matching that generates the assembly instruction for a numerical operation on integer values. We extend that pattern matching with a new case for `Sub(_, _)`:

```
match expr with
| Add(_, _) ->
    Asm(RV.ADD(Reg.r(env.Target),
              Reg.r(env.Target), Reg.r(rtarget)))
| Sub(_, _) -> // <-- We add this case
    Asm(RV.SUB(Reg.r(env.Target),
              Reg.r(env.Target), Reg.r(rtarget)))
| Mult(_, _) ->
    Asm(RV.MUL(Reg.r(env.Target),
              Reg.r(env.Target), Reg.r(rtarget)))
```

3. finally, we find the internal pattern matching that generates the assembly instruction for a numerical operation on float values. We extend that pattern matching with a new case for `Sub(_, _)`:

```
match expr with
| Add(_, _) ->
    Asm(RV.FADD_S(FPReg.r(env.FPTarget),
                 FPReg.r(env.FPTarget), FPReg.r(rfptarget)))
| Sub(_, _) -> // <-- We add this case
    Asm(RV.FSUB_S(FPReg.r(env.FPTarget),
```

(continues on next page)

(continued from previous page)

```

                                FPReg.r(env.FPtarget), FPReg.r(rfptarget)))
| Mult(_,_) ->
    Asm(RV.FMUL_S(FPReg.r(env.FPtarget),
                FPReg.r(env.FPtarget), FPReg.r(rfptarget)))

```

Testing the Code Generation

To test the code generation, we can often reuse the same test cases of the interpreter, with assertions that stop the program execution if the result of a computation is not what we expect.

Consequently, we can create test case called e.g. `tests/codegen/pass/008-sub.hyg` with the same content used for *Testing the Interpreter*. After we add this test, running `./hygdec test` should not report any failed test.

Note: When we reach this point, we should have fixed all the warnings caused by the addition of case `Sub(lhs, rhs)` in `src/AST.fs`. To double-check, we can rebuild `hygdec` by executing `dotnet clean` and then `dotnet build`. The result should be:

```

Build succeeded.
    0 Warning(s)
    0 Error(s)

```

3.11 Project Ideas

For your group project, you should implement *all* the following project ideas (but notice that some of them give you a choice between different options):

- *Project Idea: Extend Hygge0 and hygdec with New Arithmetic Operations*
- *Project Idea: Extend Hygge0 and hygdec with New Relational Operations*
- *Project Idea: Extend Hygge0 and hygdec with the Logical Operator “Exclusive Or”*

There is also an *Optional Challenge: “And” and “Or” with Short-Circuit-Semantics*. If you want to work on this challenge *instead* of some project idea, please talk with the teacher.

Note: These project ideas are tailored for *project groups*. If you have not yet joined a group, you can address part of them (e.g. implement only one new arithmetic operator instead of 3), and later combine your work with your group.

3.11.1 Project Idea: Extend Hygge0 and hyggec with New Arithmetic Operations

Add some new arithmetic operations to the Hygge0 language and to the hyggec compiler, by following the steps described in *Example: Extending Hygge0 and hyggec with a Subtraction Operator*. Choose at least 3 operations between:

- Division “ e_1/e_2 ” (both integer and floating point)
- Remainder “ $e_1 \% e_2$ ” (only between integers)
- Square root “ $\text{sqrt}(e)$ ” (only floating point)

Hint: To perform lexing and parsing of the new sqrt operation, you will need to:

- define a new token matching “sqrt”; you may call this token e.g. Sqrt. Then,
- in `src/Parser.fsy`, add a new rule to parse an occurrence of Sqrt, followed by LPAR (left parenthesis), followed by an expression, followed by RPAR (right parenthesis). You can use as a reference the existing rule that parses e.g. a print expression: it is located under `unaryExpr` and it looks like the following.

```
// Unary expression
unaryExpr:
  // ... some rules omitted ...
  | PRINT LPAR simpleExpr RPAR { mkNode(parseState, 1, Expr.Print(
  ↪$3)) }
```

The new “ $\text{sqrt}(e)$ ” expression should have the same precedence of “ $\text{print}(e)$ ”, so it should also be placed under `unaryExpr`.

- Maximum “ $\max(e_1, e_2)$ ” and minimum “ $\min(e_1, e_2)$ ” (both integer and floating point)

Hint:

- To perform lexing and parsing of the new “ $\max(e_1, e_2)$ ” and “ $\min(e_1, e_2)$ ” operations, you can follow the hints given for sqrt above. In addition, you will need to define a token for recognising the comma “,” between the two arguments: you may call this token e.g. COMMA.
- There are several ways to implement both “ $\max(e_1, e_2)$ ” and “ $\min(e_1, e_2)$ ” in hyggec. Depending on your approach, you may achieve the result without extending the interpreter, nor the code generation...
- To implement code generation for “ $\max(e_1, e_2)$ ” and “ $\min(e_1, e_2)$ ” on integers, the generated RISC-V code will need to perform a conditional jump with a branching instruction, depending e.g. on whether e_1 is smaller than e_2 . The implementation for floats is simpler. For an inspiration, see the

pattern matching case for `Less(lhs, rhs)` of `doCodegen` (in the file `src/RISCVCodegen.fs`).

3.11.2 Project Idea: Extend Hygge0 and hyggec with New Relational Operations

Add some new relational operations to the Hygge0 language and to the hyggec compiler, by following the steps described in *Example: Extending Hygge0 and hyggec with a Subtraction Operator* — except that you should use the existing expression “ $e_1 < e_2$ ” as a reference. Choose at least one operation between:

- Less than or equal to “ $e_1 \leq e_2$ ” (both integer and floating point)
- Greater than “ $e_1 > e_2$ ” (both integer and floating point)
- Greater than or equal to “ $e_1 \geq e_2$ ” (both integer and floating point)

Hint: There are several ways to implement these expressions in hyggec. Depending on your approach, you may implement all of them without extending the interpreter, nor the code generation...

3.11.3 Project Idea: Extend Hygge0 and hyggec with the Logical Operator “Exclusive Or”

Add the “exclusive or²⁶” operator “ $e_1 \text{ xor } e_2$ ” to the Hygge0 language and to the hyggec compiler, by following the steps described in *Example: Extending Hygge0 and hyggec with a Subtraction Operator* — except that you should use the existing expression “ $e_1 \text{ or } e_2$ ” as a reference.

Hint: There are several ways to implement this operator in hyggec. Depending on your approach, you may achieve the result without extending the interpreter, nor the code generation...

²⁶ https://en.wikipedia.org/wiki/Exclusive_or

3.11.4 Optional Challenge: “And” and “Or” with Short-Circuit-Semantics

The Hygge0 formal semantics (*Definition 4*) omits the reduction rules for the expressions “ e_1 and e_2 ” and “ e_1 or e_2 ” (but you may have written them down as part of Exercise ??, and compared them against their implementation in `hygdec` in [Exercise 19](#)).

`hygdec` implements the following “eager” semantics for the logical “and” and “or”: they reduce both arguments to values, and then reduce to true or false accordingly.

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e \text{ and } e_2 \rangle \rightarrow \langle R' \bullet e' \text{ and } e_2 \rangle} \text{ [R-And-L]} \quad \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v \text{ and } e \rangle \rightarrow \langle R' \bullet v \text{ and } e' \rangle} \text{ [R-And-R]}$$

$$\frac{}{\langle R \bullet \text{true and true} \rangle \rightarrow \langle R \bullet \text{true} \rangle} \text{ [R-And-Res1]}$$

$$\frac{v \text{ is a boolean value}}{\langle R \bullet \text{false and } v \rangle \rightarrow \langle R \bullet \text{false} \rangle} \text{ [R-And-Res3]}$$

$$\frac{v \text{ is a boolean value}}{\langle R \bullet v \text{ and false} \rangle \rightarrow \langle R \bullet \text{false} \rangle} \text{ [R-And-Res2]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e \text{ or } e_2 \rangle \rightarrow \langle R' \bullet e' \text{ or } e_2 \rangle} \text{ [R-Or-L]} \quad \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v \text{ or } e \rangle \rightarrow \langle R' \bullet v \text{ or } e' \rangle} \text{ [R-Or-R]}$$

$$\frac{v \text{ is a boolean value}}{\langle R \bullet \text{true or } v \rangle \rightarrow \langle R \bullet \text{true} \rangle} \text{ [R-Or-Res1]}$$

$$\frac{v \text{ is a boolean value}}{\langle R \bullet v \text{ or true} \rangle \rightarrow \langle R \bullet \text{true} \rangle} \text{ [R-Or-Res2]}$$

$$\frac{}{\langle R \bullet \text{false or false} \rangle \rightarrow \langle R \bullet \text{false} \rangle} \text{ [R-Or-Res3]}$$

Example 24

The “eager” semantics of “and” and “or” in `hygdec` can be observed e.g. by running the following Hygge0 program (available under `examples/and-or-evaluation.hyg`):

```
{println("Left of 'and'"); false} and {println("Right of 'and'"); true};
{println("Left of 'or'"); true} or {println("Right of 'or'"); true}
```

The output will be:

```
Left of 'and'
Right of 'and'
Left of 'or'
Right of 'or'
```

However, many programming languages implement a **short-circuit evaluation semantics**²⁷ for both “and” and “or”. The intuition is the following:

- the expression “ e_1 and e_2 ” reduces to false when e_1 is false. The expression e_2 is only considered (and reduced, if needed) when e_1 is true;
- the expression “ e_1 or e_2 ” reduces to true when e_1 is true. The expression e_2 is only considered (and reduced, if needed) when e_1 is false.

Example 25

With short-circuit semantics for “and” and “or”, the program in *Example 24* would only output:

```
Left of 'and'  
Left of 'or'
```

Write down the reduction semantics rules for “short-circuit and” and “short-circuit or” expressions, and implement them in hyggec. You can choose to either:

- modify the built-in interpreter and the code generation in hyggec to give new short-circuit semantics to the existing “and” and “or” expressions; or
- **(recommended)** add two new operators “&&” and “||” to hyggec. These new operators mean respectively, “short-circuit and” and short-circuit or”, and you should add them to hyggec without altering the existing “and” and “or” expressions (this is similar to the [Kotlin programming language](#)²⁸).

Hint: There are several ways to implement these operators in hyggec. Depending on your approach, you may achieve the result without changing the interpreter, nor the code generation. You may even make them simpler by *removing* some of their existing code...

²⁷ https://en.wikipedia.org/wiki/Short-circuit_evaluation

²⁸ <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-boolean/and.html>

Module 4: Lab Day

This module does not introduce new contents: **on 23 February, from 8:00 to 12:00, you can work on your project or past exercises.** The teacher and TA will be present in the **classroom**, and you can request help or ask them questions about your project, or course topics, or technical issues.

You can also use this Lab Day to **join (or create) your project group.**

We can arrange **mini-sessions during the Lab Day** to address specific questions and topics requested by two or more students. To propose a question/topic for these mini-sessions, please use the **poll on the course website on DTU Learn**, under “Contents” → “Module 4”.

Module 5: Mutability and Loops

In this module we extend the Hygge0 programming language with two new features: we add support for *Mutable Variables*, and then we add a “*While*” Loop. In principle, the two features are independent – but in practice, most programs using a while loop need to update the value of some variable that is checked in the loop condition.

These extensions are the starting point for the *Project Ideas* of this module.

5.1 Overall Objective

Our goal is to interpret, compile and run Hygge programs like the one shown in *Example 26* below, which computes and displays the first n terms of the Fibonacci sequence.

Example 26 (A Hygge Program with Mutable Variables and a Loop)

```
1 // Number of terms of the Fibonacci sequence to print (minimum 2).
2 let n: int = 16;
3
4 let mutable t0: int = 0; // First term in the Fibonacci sequence
5 let mutable t1: int = 1; // Second term in the Fibonacci sequence
6
7 println(t0);
8 println(t1);
9
10 let mutable i: int = 2; // Counter: how many terms we printed
11 let mutable next: int = 0; // Next term in the Fibonacci sequence
12
13 while (i < n) do {
14     next <- t0 + t1;
15     println(next);
16     t0 <- t1;
17     t1 <- next;
```

(continues on next page)

(continued from previous page)

```

18   i <- i + 1
19 }

```

Important: The extensions described in this module (*Mutable Variables* and “*While Loop*”) are already implemented in the *upstream Git repository of hygge*: you should pull and merge the latest changes into your project compiler. The *Project Ideas* of this module further extend Hygge with more assignment operators and more loop constructs.

5.2 Mutable Variables

We extend Hygge0 with an **assignment expression** “ $x \leftarrow e$ ” that, intuitively, works as follows:

1. the expression e on the right-hand-side of the assignment is reduced to a value v ;
2. then, in the expression “ $x \leftarrow v$ ”, the value v is assigned to variable x , and the whole expression “ $x \leftarrow v$ ” reduces to v . This way, it is possible to chain assignments — e.g. the expression “ $x \leftarrow y \leftarrow z \leftarrow 2 + 3$ ” assigns the value 5 to variables x , y , and z .

5.2.1 Design Considerations

Typical imperative programming languages (like C, C++, Java, Python...) have mutable variables, whose value can be freely changed while a program runs. Instead, Hygge0 is inspired by programming languages with functional elements (like F#, Scala, Haskell, Erlang, Elixir, Kotlin...) where variables are (by default) *immutable*: their value is determined once and for all, when they are initialised.

Extending the Hygge0 syntax in *Definition 1* with a new assignment operation is quite straightforward. However, mutable variables introduce **side effects** that cannot be easily captured by the Hygge0 semantics. In fact, introducing mutable variables adds significant complexity to the specification of Hygge — because in general, specifying the behaviour of programs with mutable variables (and reasoning about such a behaviour) is significantly harder.

To see the issue, consider what would happen if we simply tried to use assignments within the existing Hygge0 semantics from *Definition 4*: we would not be able to write any meaningful programs with mutable variables! This is because in Hygge0, each (immutable) variable x is introduced by a “let $x : t = \dots$ ” binder — and according to the semantics in *Definition 4* (rule [R-Let-Subst]), whenever we start reducing the scope where x is defined, then x “disappears” because it is substituted with its initialisation value (see *Example 27* below).

Example 27 (Assignment to a Variable Bound By “let”)

Consider the following Hygge program with an assignment:

```
let x : int = 1;
x ← 2;
print(x)
```

If we naively extended the Hygge0 semantics in *Definition 4* with an assignment expression, program would reduce as follows (in any runtime environment R):

$$\frac{}{\langle R \bullet \begin{array}{l} \text{let } x : \text{int} = 1; \\ x \leftarrow 2; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{l} 1 \leftarrow 2; \\ \text{print}(1) \end{array} \rangle} \text{[R-Let-Subst]}$$

Therefore, the program would get stuck. We could try to tweak the substitution rules to avoid substituting x on the left-hand-side of the assignment, as follows:

$$\frac{}{\langle R \bullet \begin{array}{l} \text{let } x : \text{int} = 1; \\ x \leftarrow 2; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{l} x \leftarrow 2; \\ \text{print}(1) \end{array} \rangle} \text{[R-Let-Subst]}$$

Still, this is not what we want, because we would expect to have `print(2)` after the assignment – but instead, we have `print(1)`.

Therefore, there is an overall language design decision to make:

- Do we want to change the semantics in *Definition 4* to make *all* Hygge variables potentially mutable?
- Or rather, do we want Hygge to keep its immutable variables, and let programmers specify when they want a mutable variable instead?

Here we follow the second approach, inspired by the [F# programming language design](https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/values/#why-immutable)²⁹:

1. we introduce a new binder expression “let mutable $x : t = e; e'$ ”, which introduces x as a **mutable variable** that is initialised with the result of e , and is only defined in the scope e' ;
2. to give a semantics to mutable variables, we **extend the runtime environment R** (used by the semantic rules in *Definition 4*) to keep track of mutable variables in the current scope and their current values; and
3. we define the semantics of “let mutable $x : t = e_1; e_2$ ” and “ $x \leftarrow e_3$ ” to make use of the extended runtime environment R :
 - we limit the visibility of the mutable variable x to the expression e_2 , and

²⁹ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/values/#why-immutable>

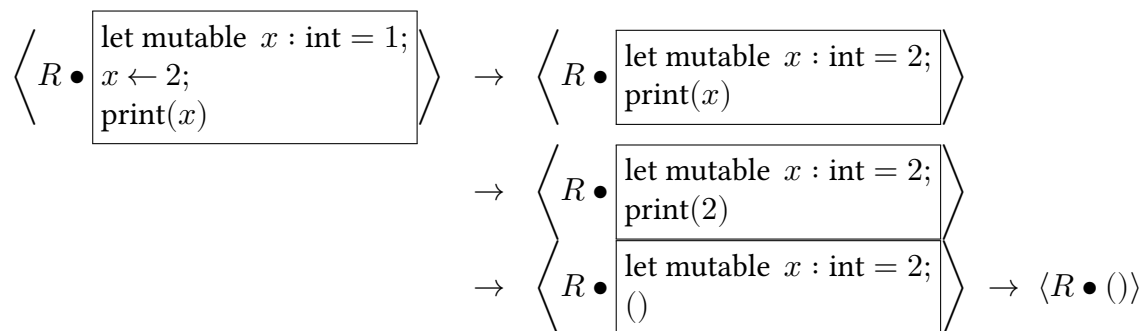
- we define the semantics of assignment to cause the desired **side effect**: besides producing a value, the reduction of the assignment “ $x \leftarrow v$ ” also updates the runtime environment R in its scope, causing the mutable variable x to be re-assigned the new value v .

Example 28 below outlines how the resulting extension of Hygge0 will behave, while *Example 29* shows some tricky cases that our specification must get right.

Example 28 (An Intuition of Mutable Variable Binding)

Consider again the program in *Example 27*. By introducing a new “let mutable $x : t = \dots$ ” expression, we want it to reduce as follows.

Observe that the “let mutable $x : \text{int} = 1; \dots$ ” binder does not substitute the variable x in its scope – and when the assignment changes the value of x from 1 to 2, then the new value is reflected in the updated binder “let mutable $x : \text{int} = 2; \dots$ ”.



Example 29 (Some Tricky Cases of Mutable Variable Binding)

Here are some tricky cases that we want to get right, when specifying how mutable variable binding and assignment should work.

In the program below, an immutable variable is shadowed by a mutable variable. We don’t want the inner assignment to influence the outer variable.

```

let x: int = 0;
{
  let mutable x: int = 3;
  x <- x + 1;
  assert(x = 4)
}
assert(x = 0)

```

The following program is similar, but now a mutable variable is shadowed by another mutable variable.

```

let mutable x: int = 0;
x <- x + 1;
{

```

(continues on next page)

(continued from previous page)

```

let mutable x: int = 3;
x <- x + 1;
assert(x = 4)
}
assert(x = 1)

```

The program below updates two mutable variables with different scopes: we want the the assignments to be correctly propagated.

```

let mutable x: int = 0;
{
  let mutable y: int = 3;
  x <- x + 1;
  y <- y + 2;
  assert(x + y = 6)
}
assert(x = 1)

```

We also want to chain assignments, and make sure that when a variable is read and reassigned, the result is correct.

```

let mutable x: float = 1.0f;
let mutable y: float = 2.0f;
let mutable z: float = 3.0f;

x <- y <- z <- x + y + z;

assert(x = y);
assert(y = z);
assert(z = 1.0f + 2.0f + 3.0f)

```

5.2.2 Syntax

We now extend the syntax of Hygge0 according to the *Design Considerations* above.

Definition 13 (Mutable Variable Binding and Assignment)

We define the syntax of Hygge0 with mutable variable binding and assignment by extending *Definition 1* with two new expressions:

Expression	$e ::= \dots$	
	let mutable $x : t = e; e'$	(Declare x as a mutable variable)
	$e \leftarrow e'$	(Assignment)

Note: In *Definition 13*, the use of a generic expression e for the target of the assignment

“ $e \leftarrow e'$ ” is not strictly necessary at this stage: we will see that the semantics of assignment in [Definition 15](#) gets stuck if the assignment target is not a variable, and the typing rules in [Definition 16](#) only support assignments that have a variable on their left-hand-side. Consequently, since we only really support assignments to a variable x , we could have defined the syntax of assignments as just “ $x \leftarrow e'$ ”.

However, later in the course we will extend assignments to support targets that are not simple variables — hence, it is useful to define a more generic syntactic rule now.

5.2.3 Operational Semantics

We now extend the semantics of Hygge0 according to [Definition 13](#) and the [Design Considerations](#) above. This involves two steps:

- extending the definition of substitution to cover the new expressions ([Definition 14](#)), and
- extending the semantic rules ([Definition 15](#)). This is the trickiest part, because the [Design Considerations](#) above make the behaviour of mutable variables non-trivial.

Definition 14 (Substitution for Mutable Variable Binding and Assignment)

We extend [Definition 2](#) (substitution) with the following new cases:

$$\begin{aligned} (\text{let mutable } x : t = e_1; e_2) [x \mapsto e'] &= \text{let mutable } x : t = e_1 [x \mapsto e']; e_2 \\ (\text{let mutable } y : t = e_1; e_2) [x \mapsto e'] &= \text{let mutable } y : t = e_1 [x \mapsto e']; e_2 [x \mapsto e'] \quad (\text{when } y \neq x) \\ (e_1 \leftarrow e_2) [x \mapsto e'] &= (e_1 [x \mapsto e']) \leftarrow (e_2 [x \mapsto e']) \end{aligned}$$

The substitution a variable x defined in [Definition 14](#) works as follows:

- the substitution on “let mutable $x : t = \dots$ ” works like the substitution on “let $x : t = \dots$ ” in [Definition 2](#);
- the substitution on the assignment “ $e_1 \leftarrow e_2$ ” propagates the substitution on both sub-expressions e_1 and e_2 .

Definition 15 (Semantics of Mutable Variable Binding and Assignment)

We extend the definition of the runtime environment R in the [Structural Operational Semantics of Hygge0](#) by adding the following field to the record R :

- R .Mutables is a mapping from variables x to values v , specifying which mutable variables are known in the current scope, and what is their current value.

Then, we define the semantics of Hygge0 with mutable variable binding and assignment by extending [Definition 4](#) to use the extended runtime environment R above, and by adding the following rules:

$$\begin{array}{c}
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{let mutable } x : t = e; e_2 \rangle \rightarrow \langle R' \bullet \text{let mutable } x : t = e'; e_2 \rangle} \text{ [R-LetM-Eval-Init]} \\
 \\
 \frac{R' = \{R \text{ with Mutables} + (x \mapsto v)\} \quad \langle R' \bullet e \rangle \rightarrow \langle R'' \bullet e' \rangle \quad \begin{array}{l} R''.\text{Mutables}(x) = v' \\ R.\text{Mutables}(x) = v_? \\ R''' = \{R'' \text{ with Mutables}(x) = v_?\} \end{array}}{\langle R \bullet \text{let mutable } x : t = v; e \rangle \rightarrow \langle R''' \bullet \text{let mutable } x : t = v'; e' \rangle} \text{ [R-LetM-Eval-Scope]} \\
 \\
 \frac{}{\langle R \bullet \text{let mutable } x : t = v; v' \rangle \rightarrow \langle R \bullet v' \rangle} \text{ [R-LetM-Res]} \\
 \\
 \frac{R.\text{Mutables}(x) = v}{\langle R \bullet x \rangle \rightarrow \langle R \bullet v \rangle} \text{ [R-Var-Res]} \\
 \\
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet x \leftarrow e \rangle \rightarrow \langle R' \bullet x \leftarrow e' \rangle} \text{ [R-Assign-Eval-Arg]} \\
 \\
 \frac{R.\text{Mutables}(x) = v' \quad R' = \{R \text{ with Mutables} + (x \mapsto v)\}}{\langle R \bullet x \leftarrow v \rangle \rightarrow \langle R' \bullet v \rangle} \text{ [R-Assign-Res]}
 \end{array}$$

The rules in [Definition 15](#) work as follows (see also [Example 30](#) below to see them in action).

- Rule [R-LetM-Eval-Init] is very similar to rule [R-Let-Eval-Init] in [Definition 4](#): it reduces the expression e that initialises a mutable variable x ;
- Rule [R-LetM-Eval-Scope] can be used to reduce “let mutable $x : t = v; e$ ” where x is initialised with a value v (that cannot be reduced further): in this case, the rule reduces the scope e of the mutable variable. Observe the crucial difference with [R-Let-Eval-Init] in [Definition 4](#): **the new rule [R-LetM-Eval-Scope] does *not* substitute x with v in the scope e** . Instead, the new rule proceeds as follows:
 1. in its first premise, the rule computes a runtime environment R' that is equal to R , except that $R'.\text{Mutables}$ maps x to the initialisation value v ;
 2. in its second premise, the rule recursively attempts to reduce the scope e in the runtime environment R' , obtaining $\langle R'' \bullet e' \rangle$;
 3. in its third premise, the rule does the following:
 - takes the value v' assigned to x in $R''.\text{Mutables}$;
 - takes the value $v_?$ assigned to x in $R.\text{Mutables}$ (this value may be undefined); and
 - computes a runtime environment R''' that is equal to R'' , except that $R'''.\text{Mutables}(x)$ has the value $v_?$ that was given to x by $R.\text{Mutables}(x)$ (if the latter was defined; if not, then $R'''.\text{Mutables}(x)$ is also undefined);
 4. in its conclusion, the rule produces the reduction result “ $\langle R \bullet \text{let mutable } x : t = v'; e' \rangle$ ” (with v' taken from the third premise above).

With this rule, the mutable variable x is only known in the scope expression e , and the reduction of e into e' might change the value assigned to x : in fact, the values v and v' may differ, if x is reassigned in the reduction from e to e' (using rule [R-Assign-Res] below). Moreover, the original runtime environment R reduces to R''' , where:

- if there was already a mutable variable also called x in R , then the value of x in R''' is unchanged (because the x defined in R is outside the scope of the “let mutable” binder, so it is *not* the same x defined in R' and R'');
 - instead, other mutable variables defined in R might be changed when e reduces to e' , and such changes are reflected in R''' .
- Rule [R-LetM-Res] reduces the whole expression “let mutable $x : t = v; v'$ ” into the value v' (i.e. this rule can only be used when the scope of the “let mutable $x : t = \dots$ ” is a value, hence x is useless).
 - Rule [R-Var-Res] allows a variable x to reduce into a value v in a runtime environment R – but the rule premise requires that x is assigned value v in the mapping R .Mutables. This rule may be used e.g. when the expression e in the scope of “let mutable $x : t = v; e$ ” reduces in the premise of rule [R-LetM-Eval-Scope] above, because e might contain a sub-expression that uses x , like “ $42 + x$ ”.
 - Rule [R-Assign-Eval-Arg] reduces an expression e being assigned to a variable x .
 - Rule [R-Assign-Res] performs the assignment of value v to variable x in the runtime environment R , as follows:
 1. the first premise of the rule requires that x has some value v' already assigned in the mapping R .Mutables (therefore, x must be a known mutable variable in the current scope);
 2. the second premise of the rule computes an updated runtime environment R' that is equal to R , except that R' .Mutables maps x to the new assigned value v ;
 3. in the conclusion, the rule reduces “ $x \leftarrow v$ ” into v , in the updated runtime environment R' .

Example 30 (A Program with a Mutable Variable)

Let us examine the reductions of the following Hygge expression, according to the semantic rules in [Definition 15](#) and [Definition 4](#):

```
let mutable  $x : \text{int} = 1 + 1;$   
 $x \leftarrow x + 40;$   
print( $x$ )
```

Let us use a runtime environment R where:

- R .Printer is defined (i.e. we can produce console output), and
- R .Mutables = \emptyset (i.e. there are no known mutable variables in the current scope).

For the first reduction, the only rule we can apply is [R-LetM-Eval-Init], which produces:

$$\frac{\frac{1 + 1 = 2}{\langle R \bullet 1 + 1 \rangle \rightarrow \langle R \bullet 2 \rangle} \text{ [R-Add-Res]}}{\langle R \bullet \text{let mutable } x : \text{int} = 1 + 1; \begin{array}{l} x \leftarrow x + 40; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \text{let mutable } x : \text{int} = 2; \begin{array}{l} x \leftarrow x + 40; \\ \text{print}(x) \end{array} \rangle} \text{ [R-LetM-Eval-Init]}$$

For the second reduction, the only rule we can apply is [R-LetM-Eval-Scope], which reduces the expression in the scope of “let mutable $x : \text{int} = \dots$ ”. To this purpose, let us now define R' as a runtime environment equal to R , except that $R'.\text{Mutables}$ maps x to the initialisation value 2 (this is omitted with “...” below). Notice that:

- in the scope of “let mutable $x : \text{int} = \dots$ ”, we use rule [R-Assign-Eval-Arg] to reduce the expression “ $x + 40$ ” being assigned to x ;
- moreover, in order to reduce the sub-expression “ $x + 40$ ” into “ $2 + 40$ ”, we use rule [R-Var-Res] to retrieve the current value of x .

$$\begin{array}{c} \frac{\frac{\frac{R'.\text{Mutables}(x) = 2}{\langle R' \bullet x \rangle \rightarrow \langle R' \bullet 2 \rangle} \text{ [R-Var-Res]}}{\langle R' \bullet x + 40 \rangle \rightarrow \langle R' \bullet 2 + 40 \rangle} \text{ [R-Add-L]}}{\langle R' \bullet x \leftarrow x + 40 \rangle \rightarrow \langle R' \bullet x \leftarrow 2 + 40 \rangle} \text{ [R-Assign-Eval-Arg]}} \\ \frac{\langle R' \bullet x \leftarrow x + 40; \text{print}(x) \rangle \rightarrow \langle R' \bullet x \leftarrow 2 + 40; \text{print}(x) \rangle} \text{ [R-Seq-Eval]} \\ R' = \dots \quad \frac{\langle R' \bullet \text{let mutable } x : \text{int} = 2; \begin{array}{l} x \leftarrow x + 40; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R' \bullet \text{let mutable } x : \text{int} = 2; \begin{array}{l} x \leftarrow 2 + 40; \\ \text{print}(x) \end{array} \rangle} \text{ [R-LetM-Eval-Scope]} \quad R'.\text{Mutables}(x) = 2 \end{array}$$

For the third reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope]; in the scope of “let mutable $x : \text{int} = \dots$ ”, we use (again) rule [R-Assign-Eval-Arg] to reduce the expression being assigned to x .

$$\begin{array}{c} \frac{\frac{\frac{2 + 40 = 42}{\langle R' \bullet 2 + 40 \rangle \rightarrow \langle R' \bullet 42 \rangle} \text{ [R-Add-Res]}}{\langle R' \bullet x \leftarrow 2 + 40 \rangle \rightarrow \langle R' \bullet x \leftarrow 42 \rangle} \text{ [R-Assign-Eval-Arg]}} \\ \frac{\langle R' \bullet x \leftarrow 2 + 40; \text{print}(x) \rangle \rightarrow \langle R' \bullet x \leftarrow 42; \text{print}(x) \rangle} \text{ [R-Seq-Eval]} \\ R' = \dots \quad \frac{\langle R' \bullet \text{let mutable } x : \text{int} = 2; \begin{array}{l} x \leftarrow 2 + 40; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R' \bullet \text{let mutable } x : \text{int} = 2; \begin{array}{l} x \leftarrow 42; \\ \text{print}(x) \end{array} \rangle} \text{ [R-LetM-Eval-Scope]} \quad R'.\text{Mutables}(x) = 2 \end{array}$$

For the fourth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope] — but now, notice that:

- in the scope of “let mutable $x : \text{int} = \dots$ ”, there is an assignment that changes the value assigned to x from 2 to 42;
- correspondingly, we use rule [R-Assign-Res] to update the runtime environment: we define R'' as a runtime environment equal to R' , except that $R''.\text{Mutables}$ maps x to the newly-assigned value 42;
- besides its side effect (i.e. updating R' into R''), the reduction of the assignment produces the assigned value 42;
- in the conclusion of this reduction, the original runtime environment R is unchanged (because x is not visible there). However, the original “let mutable $x : \text{int} = 2; \dots$ ” has now become “let mutable $x : \text{int} = 42; \dots$ ”: as a consequence, further reductions (that we explore below) will use the updated value of x .

$$\begin{array}{c}
 \frac{R'.\text{Mutables}(x) = 2 \quad R'' = \{R' \text{ with Mutables} + (x \mapsto 42)\}}{\langle R' \bullet x \leftarrow 42 \rangle \rightarrow \langle R'' \bullet 42 \rangle} \text{ [R-Assign-Res]} \\
 \frac{\langle R' \bullet \begin{array}{|l} x \leftarrow 42; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R'' \bullet \begin{array}{|l} 42; \\ \text{print}(x) \end{array} \rangle}{\langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 2; \\ x \leftarrow 42; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 42; \\ 42; \\ \text{print}(x) \end{array} \rangle} \text{ [R-LetM-Eval-Scope]} \\
 R' = \dots \qquad \qquad \qquad R''.\text{Mutables}(x) = 42
 \end{array}$$

For the fifth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and now we simplify the sequencing of expressions inside the scope of “let mutable $x : \text{int} = \dots$ ”.

$$\begin{array}{c}
 \frac{\langle R'' \bullet \begin{array}{|l} 42; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R'' \bullet \text{print}(x) \rangle}{\langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 42; \\ 42; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 42; \\ \text{print}(x) \end{array} \rangle} \text{ [R-LetM-Eval-Scope]} \\
 R'' = \dots \qquad \qquad \qquad R''.\text{Mutables}(x) = 42
 \end{array}$$

For the sixth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and now we use (again) [R-Var-Res] to retrieve the current value of x .

$$\begin{array}{c}
 \frac{R''.\text{Mutables}(x) = 42}{\langle R'' \bullet x \rangle \rightarrow \langle R'' \bullet 42 \rangle} \text{ [R-Var-Res]} \\
 \frac{\langle R'' \bullet \text{print}(x) \rangle \rightarrow \langle R'' \bullet \text{print}(42) \rangle}{\langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 42; \\ \text{print}(x) \end{array} \rangle \rightarrow \langle R \bullet \begin{array}{|l} \text{let mutable } x : \text{int} = 42; \\ \text{print}(42) \end{array} \rangle} \text{ [R-LetM-Eval-Scope]} \\
 R'' = \dots \qquad \qquad \qquad R''.\text{Mutables}(x) = 42
 \end{array}$$

For the seventh reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and now we reduce $\text{print}(42)$ to the unit value (while printing 42 on the console).

$$R'' = \dots \frac{\frac{R''.Printer \text{ is defined}}{\langle R'' \bullet \text{print}(42) \rangle \rightarrow \langle R'' \bullet () \rangle} \text{ [R-Print-Res]} \quad R''.Mutables(x) = 42}{\langle R \bullet \boxed{\text{let mutable } x : \text{int} = 42; \text{print}(42)} \rangle \rightarrow \langle R \bullet \boxed{\text{let mutable } x : \text{int} = 42; ()} \rangle} \text{ [R-LetM-Eval-Scope]}$$

For the eighth and last reduction, the only rule we can apply is [R-LetM-Res], which replaces the whole “let mutable $x : t = \dots$ ” with the value in its scope, which is the unit value $()$.

$$\frac{}{\langle R \bullet \boxed{\text{let mutable } x : \text{int} = 42; ()} \rangle \rightarrow \langle R \bullet () \rangle} \text{ [R-LetM-Res]}$$

Therefore, we were able to reduce the original program into a value, without getting stuck.

Exercise 23

Write the reductions of the following expression, in a runtime environment R . Show all reductions until the expression reduces into a value.

```
let mutable x : int = 0;
let mutable y : int = 0;
  x ← y ← 1;
  x + y
```

Exercise 24

Write the reductions of the following expression, in a runtime environment R . Show all reductions until the expression reduces into a value.

```
let mutable x : int = 0;
{
  let mutable x : int = 1;
  x ← 2
};
assert(x = 0)
```

5.2.4 Typing Rules

We now extend the typing rules of Hygge0 to support the mutable variables introduced in [Definition 13](#) according to the *Design Considerations* above. Our goal is to type-check programs that use the new assignment “ $x \leftarrow e$ ”, but only when x is in the scope of “let mutable $x : t = \dots$ ”: this way, Hygge programmers can be sure that, if their code type-checks, then it does not accidentally modify any immutable variable. Also notice that, by the semantics in [Definition 15](#), the runtime environment and expression $\langle R \bullet x \leftarrow e \rangle$ gets stuck if x is not a known mutable variables in R – hence our typing rules should prevent the possibility of assigning a value to an unknown or immutable variable.

To achieve this result, we extend the typing environment Γ in [Definition 6](#) with a new entry, called “Mutables”, which keeps track of the known mutable variables in the current scope. Then, we add two new typing rules, and slightly tweak the existing rule for (immutable) let-binders.

Definition 16 (Typing Rules for Mutable Variable Binding and Assignment)

We extend the typing environment Γ in [Definition 6](#) by adding the following entry:

- Γ .Mutables is a set of variables.

Then, we define the **typing rules of Hygge0 with mutable variable binding and assignment in two steps**.

First, we extend [Definition 11](#) with the following new rules:

$$\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash e_1 : T \quad \{\Gamma \text{ with Vars} + (x \mapsto T) \text{ and Mutables} \cup \{x\}\} \vdash e_2 : T'}{\Gamma \vdash \text{let mutable } x : t = e_1; e_2 : T'} \quad [\text{T-MLet}]$$

$$\frac{x \in \Gamma.\text{Mutables} \quad \Gamma \vdash x : T \quad \Gamma \vdash e : T}{\Gamma \vdash x \leftarrow e : T} \quad [\text{T-Assign-Var}]$$

Then, we replace the rule [T-Let] in [Definition 8](#) with the following rule:

$$\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash e_1 : T \quad \{\Gamma \text{ with Vars} + (x \mapsto T) \text{ and Mutables} \setminus \{x\}\} \vdash e_2 : T'}{\Gamma \vdash \text{let } x : t = e_1; e_2 : T'} \quad [\text{T-Let2}]$$

In [Definition 16](#), the typing rule [T-MLet] is very similar to [T-Let] in [Definition 8](#): the only difference is that, when type-checking the expression e_2 in the scope of “let mutable $x : t = \dots$ ”, rule [T-MLet] also adds the variable x to the set of known mutable variables in Γ .

The set of mutable variables in Γ is used by rule [T-Assign-Var]: it checks whether x (the target of the assignment) is mutable, and whether the type of x matches the type of the expression e being assigned to it. If all these conditions hold, then the whole assignment has type T – because, as specified in [Definition 15](#), it will produce the value being assigned.

Finally, we need to replace rule [T-Let] in [Definition 8](#) with the new rule [T-Let2]: the only difference between the two rules is that [T-Let2] removes the declared variables x

from the set of known mutable variables in the typing environment (using the standard set subtraction operation “\”). This is necessary to address [Example 31](#) below.

Example 31 (Shadowing Mutable Variables)

The following Hygge program declares a mutable variable x , and then shadows it with a regular (immutable) variable having the same name.

```

1 let mutable x: int = 0;
2 let x: int = 1;
3 x <- 2

```

We do *not* want this program to type-check, because it is attempting to use the assignment operator on a variable that has been (re-)defined as immutable by the inner “let $x : \text{int} = \dots$ ”. For this reason, we use the new rule [T-Let2] to ensure that, whenever we introduce a regular (immutable) variable, we remove it from the known mutable variables in the typing environment.

Exercise 25 (Why We Need the Typing Rule [T-Let2])

Write a typing derivation for the program in [Example 31](#) by using the “old” typing rule [T-Let] from [Definition 8](#) (instead of [T-Let2] in [Definition 16](#)).

Then, try to write a typing derivation for the same program, now using [T-Let2] in [Definition 16](#).

(Before trying this exercise, it may be helpful to have a look at the derivation in [Example 32](#) below.)

Example 32 (Type-Checking a Program with Mutable Variables)

We now see how to type-check a program with mutable variables. To this purpose, let us define the following typing environments (with the extension introduced in [Definition 16](#)):

- Γ as the empty typing environment where:
 - $\Gamma.\text{Vars} = \emptyset$;
 - $\Gamma.\text{TypeVars} = \emptyset$;
 - $\Gamma.\text{Mutables} = \emptyset$;
- Γ' as the typing environment obtained from Γ by mapping x to int in the field Vars , and adding x to the field Mutables . Therefore, we have:
 - $\Gamma'.\text{Vars} = \{x \mapsto \text{int}\}$;
 - $\Gamma'.\text{TypeVars} = \emptyset$;
 - $\Gamma'.\text{Mutables} = \{x\}$.

Here is a typing derivation that type-checks the expression in *Example 30*, according to the rules in *Definition 16*, *Definition 11*, *Definition 8*, and *Definition 7*.

$$\frac{\frac{\frac{\Gamma \vdash \text{"int"} \triangleright \text{int}}{\Gamma \vdash \text{"int"} \triangleright \text{int}} \text{[TRes-Int]} \quad \frac{\frac{\Gamma \vdash 1 : \text{int}}{\Gamma \vdash 1 + 1 : \text{int}} \text{[T-Val-Int]} \quad \dots \text{[T-Add]}}{\Gamma \vdash 1 + 1 : \text{int}} \quad \frac{\frac{\frac{\frac{\Gamma'.\text{Vars}(x) = \text{int}}{x \in \Gamma'.\text{Mutables}} \text{[T-Var]} \quad \frac{\frac{\Gamma'.\text{Vars}(x) = \text{int}}{\Gamma' \vdash x : \text{int}} \text{[T-Var]} \quad \frac{\frac{\Gamma'.\text{Vars}(x) = \text{int}}{\Gamma' \vdash 40 : \text{int}} \text{[T-Val-Int]} \quad \frac{\Gamma' \vdash x : \text{int}}{\Gamma' \vdash x + 40 : \text{int}} \text{[T-Add]}}{\Gamma' \vdash x + 40 : \text{int}} \text{[T-Assign]} \quad \frac{\Gamma'.\text{Vars}(x) = \text{int}}{\Gamma' \vdash x : \text{int}} \text{[T-Var]} \quad \frac{\Gamma'.\text{Vars}(x) = \text{int}}{\Gamma' \vdash \text{print}(x) : \text{unit}} \text{[T-Print]} \quad \frac{\Gamma'.\text{Vars}(x) = \text{int}}{\Gamma' \vdash \text{print}(x) : \text{unit}} \text{[T-Seq]}}{\Gamma' \vdash \begin{array}{l} x \leftarrow x + 40; \\ \text{print}(x) \end{array} : \text{unit}} \text{[T-LetM]}}{\Gamma \vdash \begin{array}{l} \text{let mutable } x : \text{int} = 1 + 1; \\ x \leftarrow x + 40; \\ \text{print}(x) \end{array} : \text{unit}} \text{[T-LetM]}$$

Exercise 26 (Type-Checking Mutable Variables and Assignments)

Write a typing derivation that type-checks the following expression:

```

let x : float = 2.0f;
let mutable y : float = 3.14f;
y ← x * y

```

5.2.5 Implementation

We now have a look at how `hyggec` is extended to implement mutable variables, according to the specification illustrated in the previous sections.

Tip: To see a summary of the changes described below, you can inspect the differences in the [hyggec Git repository](#) between the tags `hygge0` and `mutable-vars`.

Lexer, Parser, Interpreter, and Type Checking

These parts of the `hyggec` compiler are extended along the lines of *Example: Extending Hygge0 and hyggec with a Subtraction Operator*.

- We extend the data type `Expr<'E, 'T>` (in `AST.fs`) with two new cases, according to *Definition 13*:
 - `LetMut` for “`let mutable x : t = ...`”, and
 - `Assign` for “`e ← e'`”.
- We extend `PrettyPrinter.fs` to support the new expressions `LetMut` and `Assign`, and also to display the contents of the new typing context entry `Mutables` (see below).
- We extend `Lexer.fsl` to support two new tokens:
 - `MUTABLE` for the keyword `mutable`, and

- LARROW (left arrow) for the assignment operator `<-`.
- We extend `Parser.fsy` to recognise the desired sequences of tokens according to [Definition 13](#), and generate AST nodes for the new expressions `LetMut` and `Assign`.
- We extend the function `subst` in `ASTUtil.fs` to support the new expressions `LetMut` and `Assign`, according to [Definition 14](#).
- We extend the function `reduce` in `Interpreter.fs` according to [Definition 15](#):
 - we add new cases for `LetMut` and `Assign`, and
 - we add a new case for `Var(x)` that reduces a mutable variable `x` to its value taken from `env.Mutables` (therefore, if `x` is not present in the mapping `env.Mutables`, the expression `Var(x)` is stuck).
- We extend `Typechecker.fs` according to [Definition 16](#):
 - we extend the definition of the record `TypingEnv` with a new entry called `Mutables`;
 - we extend the function `typer` to support the new cases for:
 - * `LetMut` (since their type-checking logic is very similar to `Let`, it is implemented in function called `letTyper`); and
 - * `Assign`;
 - we update the existing typing rule for `Let` to match the revised typing rule `[T-Let2]`.

Besides, the type checking for `Let` and `LetMut` is very similar, so it is implemented in a common auxiliary function called `letTyper`.

- We also add new tests; in particular, we add the tricky cases illustrated in [Example 29](#) as tests for both the interpreter, and the [Code Generation](#) (described below).

Code Generation

The code generation for the expression “let mutable $x : t = e_1; e_2$ ” is not different from the immutable “let $x : t = \dots$ ”: this is because the mutability (or immutability) of a variable is a concept that only exists in the Hygge programming language and typing system – whereas in RISC-V assembly (just like in most other assembly variants for other CPUs), registers and memory locations are generally mutable. Moreover, the code generation of “let $x : t = \dots$ ” is already taking care of the correct scoping of variables – hence, it handles the tricky cases discussed in [Example 29](#) out-of-the-box.

Therefore, in the function `doCodegen` (in the file `RISCVCodegen.fs`) we simply add the following pattern matching case, that reuses the code generation already implemented for “let $x : t = \dots$ ”:

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  match node.Expr with
  // ...
```

(continues on next page)

(continued from previous page)

```
| LetMut(name, tpe, init, scope) ->
  // The code generation is not different from 'let...', so we recycle it
  doCodegen env {node with Expr = Let(name, tpe, init, scope)}
```

The code generation for the expression “ $x \leftarrow e$ ” is straightforward: it compiles expression e (using the current target register), and then moves (i.e. copies) the result of e where the value of x is stored (e.g. in a register). Notice that we don’t check whether x is mutable: we assume that the type checking phase has already taken care of it. (Here we omit some cases for clarity)

```
| Assign(lhs, rhs) ->
  /// Code for the 'rhs', leaving its result in the target register
  let rhsCode = doCodegen env rhs
  match lhs.Expr with
  | Var(name) ->
    match (env.VarStorage.TryFind name) with
    | Some(Storage.Reg(reg)) ->
      rhsCode.AddText(RV.MV(reg, Reg.r(env.Target)),
        $"Assignment to variable %{name}")
    | Some(Storage.FPReg(reg)) ->
      rhsCode.AddText(RV.FMV_S(reg, FPReg.r(env.FPTarget)),
        $"Assignment to variable %{name}")
    // ...
  | _ ->
    failwith ($"BUG: assignment to invalid target:%s{Util.nl}"
      + $"%s{PrettyPrinter.prettyPrint lhs}")
```

5.3 “While” Loop

We now extend the Hygge0 programming language with a “while” loop, in the style of most imperative programming languages (such as C, Java, Python, ...). The “while” expression in Hygge is written “while e_1 do e_2 ”, and it means that we want to repeat the execution of expression e_2 as long as e_1 is true.

More in detail:

1. we reduce the condition e_1 ;
2. if the condition reduces to true:
 - we start reducing the loop body e_2 until it becomes a value;
 - then, we repeat from point 1 above;
3. otherwise, if the condition reduces to false, we end the loop by reducing the whole expression to the unit value $()$.

5.3.1 Syntax

The syntax of the “while” loop (*Definition 17*) is straightforward.

Definition 17 (While Loop)

We define the syntax of Hygge0 with while loops by extending *Definition 1* with a new expressions:

$$\begin{array}{l} \text{Expression } e ::= \dots \\ \quad | \text{ while } e_1 \text{ do } e_2 \quad (\text{“While” loop}) \end{array}$$

5.3.2 Operational Semantics

To substitute variables in “while e_1 do e_2 ” we simply propagate the substitution through the loop condition e_1 and loop body e_2 , according to *Definition 18* below.

Definition 18 (Substitution for “While” Loops)

We extend *Definition 2* (substitution) with the following new case:

$$(\text{while } e_1 \text{ do } e_2)[x \mapsto e'] = \text{while } (e_1[x \mapsto e']) \text{ do } (e_2[x \mapsto e'])$$

The reduction semantics of “while e_1 do e_2 ” (*Definition 19* below) realises the behaviour described in the beginning of this section by rewriting the “while” loop expression into the following expression:

$$\text{if } e_1 \text{ then } \{e_2; \text{while } e_1 \text{ do } e_2\} \text{ else } ()$$

In other words, the “while” loop becomes an “if” expression that (according to the semantics you should have defined as part of Exercise ??) behaves as follows:

1. the “if” semantics tries to reduce the loop condition e_1 into a value;
2. then, if the condition value is true, the “if” expression executes the loop body e_2 , followed by the “while” loop again;
3. otherwise, if the condition reduces to false, the “if” expression produces the unit value $()$ (hence, it does not execute the loop body).

Definition 19 (Semantics of “While” Loops)

We define the semantics of Hygge0 with “while” loops by extending *Definition 4* with the following rule:

$$\frac{}{\langle R \bullet \text{while } e_1 \text{ do } e_2 \rangle \rightarrow \langle R \bullet \text{if } e_1 \text{ then } \{e_2; \text{while } e_1 \text{ do } e_2\} \text{ else } () \rangle} \text{ [R-While]}$$

Example 33 (A Program with a “While” Loop)

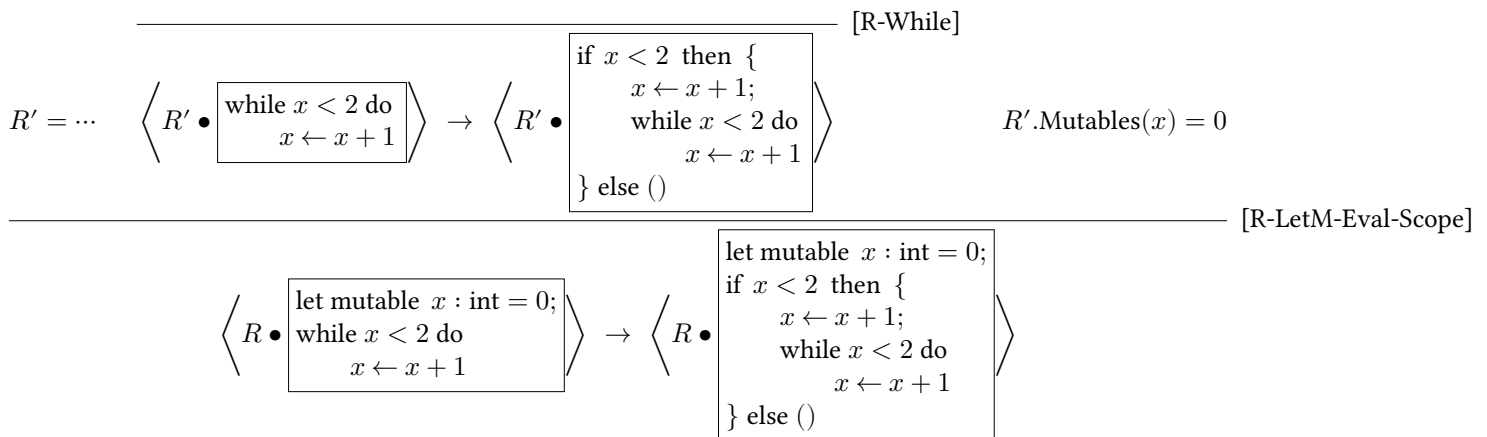
Let us examine the reductions of the following Hygge expression, according to the semantic rules in *Definition 19*, *Definition 15* and *Definition 4*:

```
let mutable x : int = 0;
while x < 2 do
  x ← x + 1
```

Let us use a runtime environment R where:

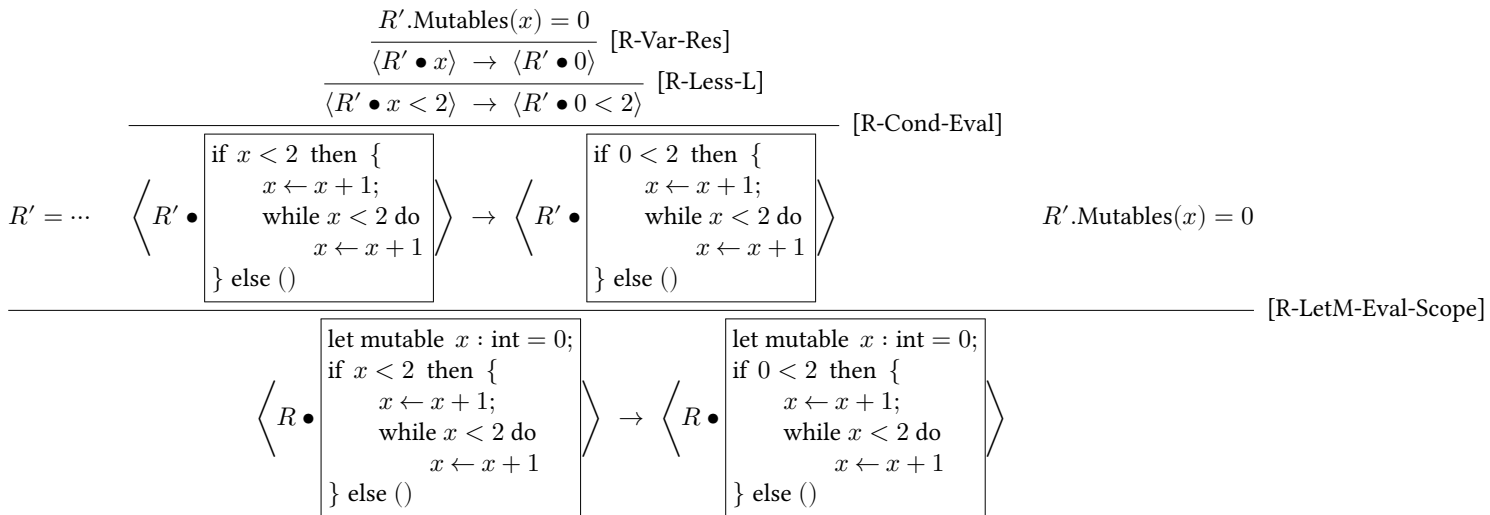
- $R.\text{Mutables} = \emptyset$ (i.e. there are no known mutable variables in the current scope).

For the first reduction, the only rule we can apply is [R-LetM-Eval-Scope], which reduces the expression in the scope of “let mutable $x : \text{int} = \dots$ ”. To this purpose, let us now define R' as a runtime environment equal to R , except that $R'. maps x to the initialisation value 0 (this is omitted with “...” below). Observe that, to reduce the expression in the scope of “let mutable $x : \text{int} = \dots$ ”, we use the new rule [R-While], which rewrites the “while” loop into an “if”.$

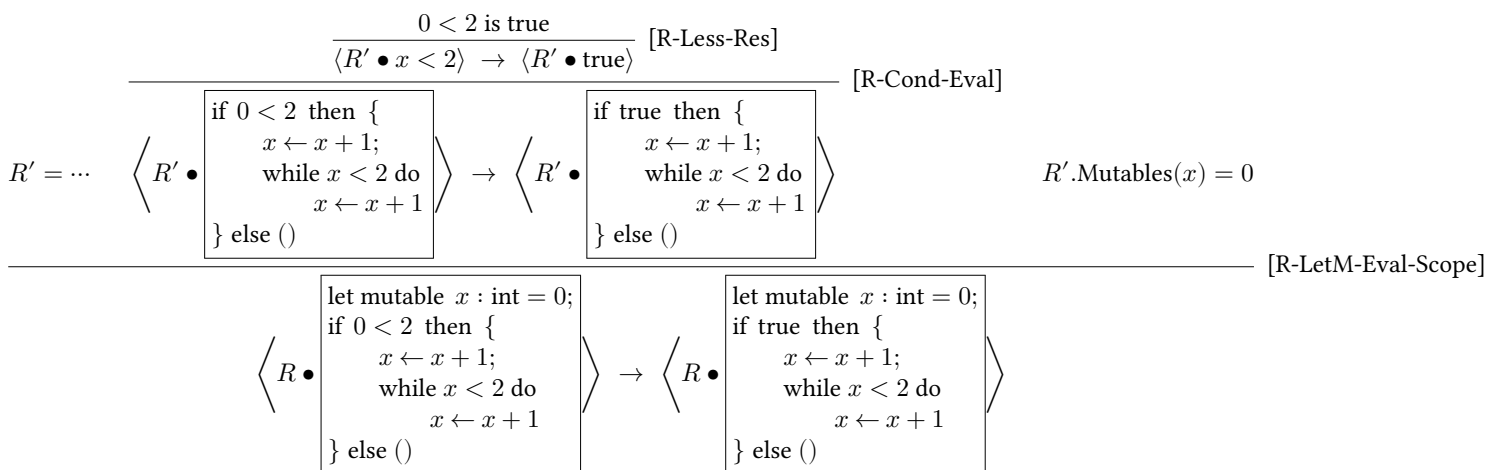


For the second reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope]. Notice that:

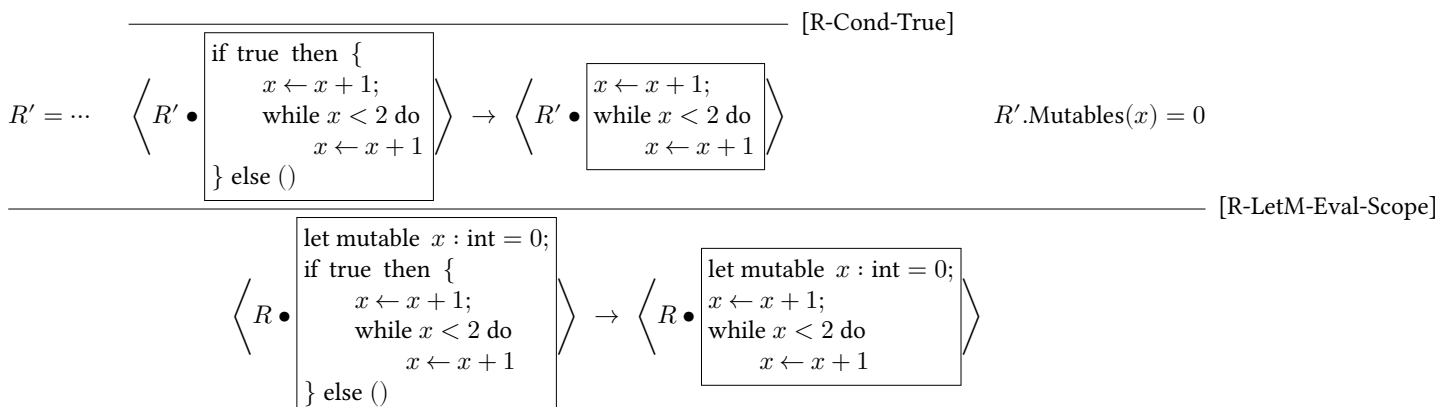
- to reduce the scope of the “let mutable $x : \text{int} = \dots$ ” we use two rules that you should have defined as part of Exercise ??:
 - one rule to reduce the “if” condition: let us call this rule [R-Cond-Eval];
 - another rule to reduce the left-hand-side of a comparison: let us call this rule [R-Less-L];
- we also use rule [R-Var-Res] from *Definition 15* to access the current value of the mutable variable x .



For the third reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we keep reducing the condition of the “if”. We use another rule that you should have defined as part of Exercise ??, to compare two values: let us call this rule [R-Less-Res].



For the fourth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we now select the “then” branch of the “if”: to do this, we use another rule that you should have defined as part of Exercise ??, which we call [R-Cond-True].



For the fifth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we now we replace the mutable variable x (on the right-hand-side of the assignment) with its current value 0.

$$\begin{array}{c}
 \frac{\frac{\frac{R'.\text{Mutables}(x) = 0}{\langle R' \bullet x \rangle \rightarrow \langle R' \bullet 0 \rangle} \text{ [R-Var-Res]}}{\langle R' \bullet x + 1 \rangle \rightarrow \langle R' \bullet 0 + 1 \rangle} \text{ [R-Add-L]}}{\langle R' \bullet x \leftarrow x + 1 \rangle \rightarrow \langle R' \bullet x \leftarrow 0 + 1 \rangle} \text{ [R-Assign-Eval-Arg]} \\
 \text{ [R-Seq-Eval]} \\
 \hline
 R' = \dots \left\langle R' \bullet \begin{array}{l} x \leftarrow x + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R' \bullet \begin{array}{l} x \leftarrow 0 + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \quad R'.\text{Mutables}(x) = 0 \\
 \hline
 \text{ [R-LetM-Eval-Scope]} \\
 \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 0; \\ x \leftarrow x + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 0; \\ x \leftarrow 0 + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle
 \end{array}$$

For the seventh reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we now fully reduce the right-hand-side of the assignment into the value 1.

$$\begin{array}{c}
 \frac{\frac{\frac{0 + 1 = 1}{\langle R' \bullet 0 + 1 \rangle \rightarrow \langle R' \bullet 1 \rangle} \text{ [R-Add-Res]}}{\langle R' \bullet x \leftarrow 0 + 1 \rangle \rightarrow \langle R' \bullet x \leftarrow 1 \rangle} \text{ [R-Assign-Eval-Arg]} \\
 \text{ [R-Seq-Eval]} \\
 \hline
 R' = \dots \left\langle R' \bullet \begin{array}{l} x \leftarrow 0 + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R' \bullet \begin{array}{l} x \leftarrow 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \quad R'.\text{Mutables}(x) = 0 \\
 \hline
 \text{ [R-LetM-Eval-Scope]} \\
 \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 0; \\ x \leftarrow 0 + 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 0; \\ x \leftarrow 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle
 \end{array}$$

For the eighth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we now re-assign variable x , changing its value from 0 to 1. To this end, we use rule [R-Assign-Res] to update the runtime environment: we define R'' as a runtime environment equal to R' , except that $R''.\text{Mutables}$ maps x to the newly-assigned value 1.

$$\begin{array}{c}
 \frac{R'.\text{Mutables}(x) = 0 \quad R'' = \{R' \text{ with Mutables} + (x \mapsto 1)\}}{\langle R' \bullet x \leftarrow 1 \rangle \rightarrow \langle R'' \bullet 1 \rangle} \text{ [R-Assign-Res]} \\
 \text{ [R-Seq-Eval]} \\
 \hline
 R' = \dots \left\langle R' \bullet \begin{array}{l} x \leftarrow 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R' \bullet \begin{array}{l} 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \quad R'.\text{Mutables}(x) = 0 \\
 \hline
 \text{ [R-LetM-Eval-Scope]} \\
 \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 0; \\ x \leftarrow 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 1; \\ 1; \\ \text{while } x < 2 \text{ do} \\ x \leftarrow x + 1 \end{array} \right\rangle
 \end{array}$$

For the ninth reduction, the only rule we can apply is (again) [R-LetM-Eval-Scope], and we now and now we simplify the sequencing of expressions inside the scope of “let mutable $x : \text{int} = \dots$ ”.

$$\begin{array}{c}
 \overline{\hspace{10em}} \text{ [R-Seq-Res]} \\
 R' = \dots \left\langle R' \bullet \begin{array}{l} 1; \\ \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R' \bullet \begin{array}{l} \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle \quad R'.\text{Mutables}(x) = 0 \\
 \overline{\hspace{10em}} \text{ [R-LetM-Eval-Scope]} \\
 \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 1; \\ 1; \\ \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 1; \\ \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle
 \end{array}$$

We have now reached an expression that is very similar to the one at the beginning of this example – except that the mutable variable x is now initialised with 1 instead of 0.

If we perform 9 more reductions similar to the ones above, we cause a further increment of variable x ; then, we can again expand the “while” into an “if” (by rule [R-While]) thus obtaining:

$$\left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 1; \\ \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \dots \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{while } x < 2 \text{ do} \\ \quad x \leftarrow x + 1 \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{if } x < 2 \text{ then } \{ \\ \quad x \leftarrow x + 1; \\ \quad \text{while } x < 2 \text{ do} \\ \quad \quad x \leftarrow x + 1 \\ \} \text{ else } () \end{array} \right\rangle$$

This last expression will now reduce by making the “if” condition false:

$$\left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{if } x < 2 \text{ then } \{ \\ \quad x \leftarrow x + 1; \\ \quad \text{while } x < 2 \text{ do} \\ \quad \quad x \leftarrow x + 1 \\ \} \text{ else } () \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{if } 2 < 2 \text{ then } \{ \\ \quad x \leftarrow x + 1; \\ \quad \text{while } x < 2 \text{ do} \\ \quad \quad x \leftarrow x + 1 \\ \} \text{ else } () \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{if false then } \{ \\ \quad x \leftarrow x + 1; \\ \quad \text{while } x < 2 \text{ do} \\ \quad \quad x \leftarrow x + 1 \\ \} \text{ else } () \end{array} \right\rangle$$

And now, the last expression takes the “else” branch of the “if” (using a rule that you should have defined as part of Exercise ??): therefore, it reduces into the final unit value $()$:

$$\left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ \text{if false then } \{ \\ \quad x \leftarrow x + 1; \\ \quad \text{while } x < 2 \text{ do} \\ \quad \quad x \leftarrow x + 1 \\ \} \text{ else } () \end{array} \right\rangle \rightarrow \left\langle R \bullet \begin{array}{l} \text{let mutable } x : \text{int} = 2; \\ () \end{array} \right\rangle \rightarrow \langle R \bullet () \rangle$$

5.3.3 Typing Rules

We now extend the typing rules of Hygge0 to support the “while” loop introduced in [Definition 17](#). This only requires a new typing rule, that is quite straightforward (see [Definition 20](#) below): when type-checking “while e_1 do e_2 ”, we make sure that e_1 has type `bool`, and e_2 is well-typed with some type T ; if these premises hold, then the “while” loop has type `unit` (because whenever the loop terminates, it produces a unit value `()`), according to its semantics in [Definition 19](#).

Definition 20 (Typing Rules for “While” Loops)

We define the typing rules of Hygge0 with “while” loops by extending [Definition 11](#) with the following rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}} \text{ [T-While]}$$

Exercise 27

Write a typing derivation that type-checks the following expression (giving to each operator its conventional mathematical precedence):

```
let mutable x : int = 0;
while x * 2 < x + 4 do
  x ← x * 2 + 1
```

5.3.4 Implementation

We now have a look at how `hyggec` can be extended to implement “while” loops, according to the specification illustrated in the previous sections.

Tip: To see a summary of the changes described below, you can inspect the differences in the [hyggec Git repository](#) between the tags `mutable-vars` and `while`.

Lexer, Parser, Interpreter, and Type Checking

These parts of the `hyggec` compiler are extended along the lines of [Example: Extending Hygge0 and hyggec with a Subtraction Operator](#).

- We extend the data type `Expr<'E, 'T>` (in `AST.fs`) with a new named case `While` for “while e_1 do e_2 ” (according to [Definition 17](#)).
 - We extend `PrettyPrinter.fs` to support the new expression `While`.
-

- We extend `Lexer.fs` to support two new tokens:
 - `WHILE` for the keyword `while`, and
 - `DO` for the keyword `do`.
- We extend `Parser.fsy` to recognise the desired sequences of tokens according to [Definition 17](#), and generate AST nodes for the new expression `While`.
- We extend the function `subst` in `ASTUtil.fs` to support the new expressions `LetMut` and `Assign`, according to [Definition 18](#).
- We extend the function `reduce` in `Interpreter.fs` to support `While`, according to [Definition 19](#).
- We extend `Typechecker.fs` according to [Definition 20](#).

Code Generation

The RISC-V assembly code generated for the expression “`while e_1 do e_2` ” must mimic the semantics in [Definition 19](#). Therefore, it must:

1. execute the assembly code generated for the condition expression e_1 ;
2. check the result produced by the code of e_1 :
 - if the result is 0 (false), the assembly code must jump to the end of the loop, and produce no result (corresponding to the unit value). Hence, we need to place an **assembly label to mark the end of the loop**;
 - otherwise, the assembly code must:
 - execute the expression e_2 (i.e. the body of the loop); and
 - afterwards, re-evaluate the loop condition. To this purpose, the assembly code can jump back to point 1 above – hence, we need to place an **assembly label to mark the beginning of the loop**.

Therefore, in the function `doCodegen` (in the file `RISCVCodegen.fs`) we add the following pattern matching case.

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  match node.Expr with
  // ...
  | While(cond, body) ->
    /// Label to mark the beginning of the 'while' loop
    let whileBeginLabel = Util.genSymbol "while_loop_begin"
    /// Label to mark the end of the 'while' loop
    let whileEndLabel = Util.genSymbol "while_loop_end"
    // Check the 'while' condition, jump to 'whileEndLabel' if it is false
    Asm(RV.LABEL(whileBeginLabel))
      ++ (doCodegen env cond)
        .AddText(RV.BEQZ(Reg.r(env.Target), whileEndLabel),
                 "Jump if 'while' loop condition is false")
```

(continues on next page)

(continued from previous page)

```
++ (doCodegen env body)
  .AddText([
    (RV.J(whileBeginLabel), "Next iteration of the 'while' loop")
    (RV.LABEL(whileEndLabel), "")
  ])
)
```

5.4 Project Ideas

For your group project, you should implement *all* the following project ideas (but notice that some of them give you a choice between different options):

- *Project Idea: C-Style Increment/Decrement Operators*
- *Project Idea: C-Style Compute-Assign Operators*
- *Project Idea: “Do...While” Loop*
- *Project Idea: “For” Loop*

There is also an *Optional Challenge: a Better “Do...While” Loop* (which can replace the “do...while” project idea).

Hint: There are several ways to implement the following project ideas. Depending on your approach, you may achieve the result without extending the interpreter, nor the code generation...

5.4.1 Project Idea: C-Style Increment/Decrement Operators

Add increment or decrement operators to the Hygge0 language and to the `hygdec` compiler, by following the steps described in *Example: Extending Hygge0 and `hygdec` with a Subtraction Operator*. Choose at least two of the following.

- **Pre-increment expression** “ $++x$ ”, which increments the value of the mutable variable x by 1, and reduces to the value of x *after* the increment. The value assigned to variable x can only be an integer or a float.
- **Post-increment expression** “ $x++$ ”, which increments the value of the mutable variable x by 1, and reduces to the value that x had *before* the increment. The value assigned to variable x can only be an integer or a float.
- **Pre-decrement expression** “ $--x$ ”, which decrements the value of the mutable variable x by 1, and reduces to the value of x *after* the decrement. The value assigned to variable x can only be an integer or a float.
- **Post-decrement expression** “ $x--$ ”, which decrements the value of the mutable variable x by 1, and reduces to the value that x had *before* the decrement. The value assigned to variable x can only be an integer or a float.

Hint:

- For the increment and decrement operators, you should define new tokens for the symbols ++ and/or -- (called e.g. PLUSPLUS and MINUSMINUS).
- When modifying Parser.fsy, the increment and decrement operators should be placed in the syntactic category unaryExpr.
- You will need to be careful with the precedence of the increment operators. For example, consider the following expression:

```
++ x ++
```

Should it be parsed as “++ (x ++)” or as “(++ x) ++”?

To avoid ambiguities and select the first option (i.e. give higher precedence to the postfix increment), you could define parsing rules like:

```
unaryExpr:
  // ...
  | PLUSPLUS unaryExpr      { ... }
  | ascriptionExpr PLUSPLUS { ... }

ascriptionExpr:
  // ...
```

5.4.2 Project Idea: C-Style Compute-Assign Operators

Add compute-assign operators to the Hygge0 language and to the hygge compiler, by following the steps described in *Example: Extending Hygge0 and hygge with a Subtraction Operator*. Choose at least two of the following.

- **Add-assign expression** “ $x += e$ ”, which computes the value of $x + e$, assigns the result to the mutable variable x , and reduces to the updated value of x . This expression only reduces to a value when both x and e are integers or floats.
- **Minus-assign expression** “ $x -= e$ ”, which computes the value of $x - e$, assigns the result to the mutable variable x , and reduces to the updated value of x . This expression only reduces to a value when both x and e are integers or floats.
- **Multiply-assign expression** “ $x *= e$ ”, which computes the value of $x * e$, assigns the result to the mutable variable x , and reduces to the updated value of x . This expression only reduces to a value when both x and e are integers or floats.
- **Divide-assign expression** “ $x /= e$ ”, which computes the value of x/e , assigns the result to the mutable variable x , and reduces to the updated value of x . This expression only reduces to a value when both x and e are integers or floats.
- **Modulo-assign expression** “ $x \% = e$ ”, which computes the value of the modulo $x \% e$, assigns the result to the mutable variable x , and reduces to the updated

value of x . This expression only reduces to a value when both x and e are integers.

Hint: To add these new assignment operators, you should define new tokens for their symbols — e.g. you could add a token for `+=` called `ADD_ASSIGN`. These new assignment operators should have the same priority and associativity of the regular assignment, hence they can be added in `Parser.fsy` under the syntactic category `simpleExpr`.

5.4.3 Project Idea: “Do..While” Loop

Add a “do e_1 while e_2 ” expression to the Hygge0 language and to the `hyggec` compiler, by following the steps described in *Example: Extending Hygge0 and hyggec with a Subtraction Operator*. The semantics of “do e_1 while e_2 ” is:

1. reduce e_1 into a value;
2. then, check the condition e_2 :
 - if e_2 is true, repeat from point 1;
 - otherwise, reduce the whole “do..while” expression to the unit value `()`.

5.4.4 Project Idea: “For” Loop

Add a C-style loop expression “for ($e_i; e_c; e_u$) e_b ” to the Hygge0 language and to the `hyggec` compiler, by following the steps described in *Example: Extending Hygge0 and hyggec with a Subtraction Operator*. The behaviour of the expression is the following:

1. reduce the initialisation expression e_i into a value;
2. reduce the condition expression e_c into a value;
 - if e_c reduces to true:
 - reduce the expression e_b (the body of the loop) into a value;
 - execute the update expression e_u ;
 - repeat from point 2;
 - otherwise, if e_c reduces to false, reduce the whole for-loop expression into the unit value `()`.

Note: Unlike the “for” loop in C, the initialisation expression e_i is *not* expected to introduce new variables that are also visible in e_c , e_u , and e_b . In other words, the “for” loop of this project idea should *not* act like a new binder in the style of “let”.

If you want to implement a more faithful C-style version of “for” loops, where e_i can introduce new variables that are also visible in e_c , e_u , and e_b , please speak with the teacher.

5.4.5 Optional Challenge: a Better “Do...While” Loop

This optional challenge can be selected instead of the “do...while” *Project Idea above*.

Improve the specification of the “do e_1 while e_2 ” loop outlined in the *Project Idea above*, as follows:

1. reduce e_1 into a value;
2. then, check the condition e_2 :
 - if e_2 is true, repeat from point 1;
 - otherwise, reduce the whole “do...while” expression to the **value produced by the last execution of e_1** .

Define the semantics and typing rules that reflect this improved specification, and implement them in `hygdec`.

Module 6: Functions and the RISC-V Calling Convention

In this module we study how to add functions to the Hygge0 programming language. On the specification side, functions are not very complicated. However, their code generation is very dependent on the specifics of the target hardware architecture — and for this reason, we will need to study the *RISC-V calling convention*.

6.1 Overall Objective

Our goal is to interpret, compile and run Hygge programs like the one shown in *Example 34* below.

Example 34 (A Hygge Program with Functions)

```
1 // A simple function: takes two integer arguments and returns an integer.
2 fun f(x: int, y: int): int = x + y;
3
4 // A function instance (a.k.a. lambda term), with a function type, saying:
5 // g is a function that takes two integer arguments and returns an integer.
6 let g: (int, int) -> int = fun (x: int, y: int) -> x + y + 1;
7
8 let x: int = f(1, 2); // Applying ("calling") a function
9 let y: int = g(1, 2); // Applying ("calling") a lambda abstraction
10 assert(x + 1 = y);
11
12 // A function that defines a nested function and calls it.
13 fun h(x: int): int = {
14     fun privateFun(z: int): int = z + 2;
15     privateFun(x)
16 };
17
18 let z: int = h(40);
19 assert(z = 42);
```

(continues on next page)

(continued from previous page)

```
20
21 // A function that takes a function as argument, and calls it.
22 fun applyFunToInt(f: (int) -> int,
23                 x: int): int =
24     f(x);
25
26 assert(applyFunToInt(h, 1) = 3);
27
28 // A function that defines a nested function and returns it.
29 fun makeFun(addOne: bool): (int) -> int =
30     if (addOne) then {
31         fun inc(x: int): int = x + 1;
32         inc
33     } else {
34         fun (x: int) -> x + 2
35     };
36
37 let plusOne: (int) -> int = makeFun(true);
38 let plusTwo: (int) -> int = makeFun(false);
39 assert(plusOne(42) = 43);
40 assert(plusTwo(42) = 44);
41 assert((makeFun(true))(42) = 43)
```

To introduce functions in the Hygge specification, and make it possible to write programs like the one in *Example 34*, we follow typical conventions of functional programming languages:

- we treat **functions as first-class values** that can be passed and returned. Such values are also called **lambda terms** or **lambda abstractions**;
- to “call” a function, we **apply a function** (or an expression that reduces into a function) to other values; result of the application is a new value;
- we introduce a new **function type** to specify what type of arguments a function expects, and what type of value it returns.

Important: The extension described in this module is already (partially) implemented in the *upstream Git repository of hyggec*: you should pull and merge the latest changes into your project compiler. The *Project Ideas* of this module further extend Hygge with better support for functions.

6.2 Syntax

Definition 21 below extends the syntax of Hygge with functions and function applications, and with a new pretype that specifies the syntax of a new function type. It also introduces a short-hand notation for defining functions.

Definition 21 (Functions and Applications)

We define the **syntax of Hygge0 with functions** by extending *Definition 1* with a new expressions, a new value, and a new pretype:

Expression	$e ::= \dots$	
	$e(e_1, \dots, e_n)$	(Application, with $n \geq 0$)
Value	$v ::= \dots$	
	$\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$	(Function instance, a.k.a. lambda term, with $n \geq 0$)
Pretype	$t ::= \dots$	
	$(t_1, \dots, t_n) \rightarrow t$	(Function type, with $n \geq 0$)

We also define the following **syntactic sugar** that provides a shorter way to define a **named function**:

$$\text{fun } name(x_1 : t_1, \dots, x_n : t_n) : t = e_1; e_2$$

which is shorthand for the following Hygge expression:

$$\text{let } name : (t_1, \dots, t_n) \rightarrow t = (\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e_1); e_2$$

The expansion of syntactic sugar into actual Hygge grammar elements is called **desugaring**.

In *Definition 21* above, a **function instance** (a.k.a. **lambda term**) is written “ $\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ ”, and it consists of:

- zero or more **formal arguments** x_1, \dots, x_n , each one annotated with a pretype; and
- the **function body** e , which is executed when the function is applied to **actual arguments** in order to produce a value (see below).

A function instance (a.k.a. lambda term) can be “called” through an **application** “ $e(e_1, \dots, e_n)$ ”, which means that the expression e (which is expected to reduce into a function instance) is applied to zero or more expressions e_1, \dots, e_n : each of these expressions should reduce into a value, that becomes an **actual argument** of the function application. (For more details, see the *Operational Semantics* below.)

A **function pretype** “ $(t_1, \dots, t_n) \rightarrow t$ ” is the syntactic description of the type of a function that takes zero or more arguments having type t_1, \dots, t_n , and returns a value of type t .

Just like the *other pretypes in Hygge0*, we will need to resolve this new pretype into a valid function type: we will address this later, when discussing the *Typing Rules*.

Finally, the **syntactic sugar** in *Definition 21* provides a convenient syntax for a very typical case: the definition of a **named function** with a certain *name* and arguments. This syntax is desugared into a (more verbose) “let...” binder that defines a variable called *name*, having a function type, and initialised with a lambda term. This way, Hygge programmers can have a convenient syntax, without extending the Hygge grammar with a new dedicated expression. (Notice that, had we extended the grammar with a new dedicated expression, we would have also needed to specify how to handle the new expression in the semantics, type checking rules, etc.)

Example 35 (Syntactic Sugar for Function Definitions)

Using the syntactic sugar in *Definition 21*, we can write:

```
fun f(x : int, y : int) : int = {  
    println("Called!");  
    x + y  
};  
f(2, 3)
```

The expression above is just an alias for the following desugared expression:

```
let f: (int, int) → int = fun (x : int, y : int) → {  
    println("Called!");  
    x + y  
};  
f(2, 3)
```

In other words, the syntactic sugar above defines regular “let” binding that:

1. introduces a variable with the given name (in this case, *f*) and a function type, and
2. initialises the variable with a function instance (a.k.a. lambda term) of the corresponding type.

In the scope of this “let”, it is possible to call the function instance by simply applying the newly-defined variable *f* to some arguments, e.g. as *f(2, 3)*.

6.3 Operational Semantics

Definition 22 formalises how substitution works for function instances and applications.

Definition 22 (Substitution for Functions and Applications)

We extend *Definition 2* (substitution) with the following new cases:

$$\begin{aligned}
 (\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e) [x \mapsto e'] &= \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e \quad (\text{when } x = x_i, \text{ for some } i \in 1..n) \\
 (\text{fun } (y_1 : t_1, \dots, y_n : t_n) \rightarrow e) [x \mapsto e'] &= \text{fun } (y_1 : t_1, \dots, y_n : t_n) \rightarrow e [x \mapsto e'] \quad (\text{when } x \neq y_i, \text{ for all } i \in 1..n) \\
 (e(e_1, \dots, e_n)) [x \mapsto e'] &= (e[x \mapsto e'])(e_1[x \mapsto e'], \dots, e_n[x \mapsto e'])
 \end{aligned}$$

According to [Definition 22](#):

- if a substitution is applied to a function instance “ $\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ ”, then the substitution is propagated through the body of the function e – unless the variable being substituted coincides with one of the function arguments x_i (for some $i \in 1..n$). This is because the function instance acts as a binder, hence if an argument is called x , then any reference to x in the function body e is referring to that argument x (i.e. any other variable x in the surrounding scope is shadowed);
- if a substitution is applied to an application $e(e_1, \dots, e_n)$, then the substitution is propagated throughout e (the expression being applied) and the application arguments (e_1, \dots, e_n) .

We can now introduce the semantics of function application, in [Definition 23](#) below.

Definition 23 (Semantics of Functions and Applications)

We define the semantics of Hygge0 with functions and applications by extending [Definition 4](#) with the following rules:

$$\begin{aligned}
 &\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e(e_1, \dots, e_n) \rangle \rightarrow \langle R' \bullet e'(e_1, \dots, e_n) \rangle} \quad [\text{R-App-Eval-L}] \\
 &\frac{\langle R \bullet e_i \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v(v_1, \dots, v_{i-1}, e_i, e_{i+1}, \dots, e_n) \rangle \rightarrow \langle R' \bullet v(v_1, \dots, v_{i-1}, e', e_{i+1}, \dots, e_n) \rangle} \quad [\text{R-App-Eval-Ri}] \\
 &\frac{}{\langle R \bullet (\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e)(v_1, \dots, v_n) \rangle \rightarrow \langle R \bullet e[x_1 \mapsto v_1] \dots [x_n \mapsto v_n] \rangle} \quad [\text{R-App-Res}]
 \end{aligned}$$

Given an application “ $e(e_1, \dots, e_n)$ ”, the reduction rules in [Definition 23](#) work as follows:

- rule [R-App-Eval-L] requires us to first reduce e , i.e. the expression being applied;
- when the expression being applied reduces into a value, rule [R-App-Eval-Ri] allows us to reduce the application arguments e_1, \dots, e_n , proceeding from left to right, until all of them become values;
- finally, rule [R-App-Res] allows us to perform the actual application. This rule requires that:
 1. the value being applied is a function instance (i.e. a lambda term) $\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$; and
 2. the number of values used as arguments for the application is n , i.e. the same number of arguments expected by the function instance.

When these conditions are met, the application proceeds by taking the body of the function instance e , and replacing each formal argument x_i with the corresponding actual argument value v_i .

(To see function application in action, see [Example 36](#) below.)

Example 36 (Reduction of Function Application)

Consider again the Hygge program in [Example 35](#) (in its desugared version). Take a runtime environment R where $R.\text{Printer}$ is defined. The program reduces as follows.

The first reduction uses rule [R-Let-Subst] (from [Definition 4](#)) to substitute each occurrence of variable f in the scope of the “let...” with the lambda term that initialises f (recall that by [Definition 21](#), lambda terms are values and cannot be further reduced).

$$\left\langle R \bullet \left[\begin{array}{l} \text{let } f : (\text{int}, \text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}, y : \text{int}) \rightarrow \{ \\ \quad \text{println}(\text{"Called!"}); \\ \quad x + y \\ \} \\ \}; \\ f(2, 3) \end{array} \right] \right\rangle \rightarrow \left\langle R \bullet \left[\begin{array}{l} \left(\text{fun } (x : \text{int}, y : \text{int}) \rightarrow \{ \\ \quad \text{println}(\text{"Called!"}); \\ \quad x + y \\ \} \right) (2, 3) \end{array} \right] \right\rangle \quad \text{[R-Let-Subst]}$$

The second reduction uses rule [R-App-Res] (from [Definition 23](#)) to apply lambda term to the arguments (2, 3): after the reduction, we obtain the body of the function with argument x replaced by 2, and argument y replaced by 3.

$$\left\langle R \bullet \left[\begin{array}{l} \left(\text{fun } (x : \text{int}, y : \text{int}) \rightarrow \{ \\ \quad \text{println}(\text{"Called!"}); \\ \quad x + y \\ \} \right) (2, 3) \end{array} \right] \right\rangle \rightarrow \left\langle R \bullet \left[\begin{array}{l} \text{println}(\text{"Called!"}); \\ 2 + 3 \end{array} \right] \right\rangle \quad \text{[R-App-Res]}$$

Then, the program continues reducing as expected, by performing the `println(...)` and reaching the final value 5.

Exercise 28 (Reduction of Function Application)

Write the reductions of the following Hygge expression: (note that you will need to desugar it first)

$$\begin{array}{l} \text{fun } f(x : \text{int}, y : \text{int}) : \text{int} = \{ \\ \quad \text{fun } g(x : \text{int}) : \text{int} = x + 1; \\ \quad g(x) + y \\ \}; \\ f(2, 3) \end{array}$$

6.4 Typing Rules

We now discuss how to type-check lambda terms and applications. First, we need to introduce a new **function type**, formalised in *Definition 24* below.

Definition 24 (Function Type)

We extend the Hygge0 typing system with a **function type** by adding the following case to *Definition 5*:

$$\begin{array}{l} \text{Type } T ::= \dots \\ \quad | (T_1, \dots, T_n) \rightarrow T \quad (\text{Function type, with } n \geq 0) \end{array}$$

In a function type $(T_1, \dots, T_n) \rightarrow T$, the types T_1, \dots, T_n are the **argument types**, while T is the **return type**.

Example 37

The following function type denotes a function that takes two arguments (an integer and a boolean) and returns a string.

$$(\text{int}, \text{bool}) \rightarrow \text{string}$$

The following function type, instead, denotes a function that:

- takes three arguments:
 1. a float,
 2. an integer, and
 3. a function that takes two booleans and returns an integer;
- returns a function that takes zero arguments, and returns a unit value.

$$(\text{float}, \text{int}, (\text{bool}, \text{bool}) \rightarrow \text{int}) \rightarrow (() \rightarrow \text{unit})$$

We also need a way to resolve a syntactic function pretype (from *Definition 21*) into a valid function type (from *Definition 24*): this is formalised in *Definition 25* below.

Definition 25 (Resolution of Function Types)

We extend *Definition 7* (type resolution judgement) with this new rule:

$$\frac{\forall i \in 1..n : \Gamma \vdash t_i \triangleright T_i \quad \Gamma \vdash t \triangleright T}{\Gamma \vdash (t_1, \dots, t_n) \rightarrow t \triangleright (T_1, \dots, T_n) \rightarrow T} \quad [\text{TRes-Fun}]$$

Rule [TRes-Fun] in *Definition 25* says that, in order to resolve a function pretype “ $(t_1, \dots, t_n) \rightarrow t$ ” into a valid function type, we need to:

- resolve each argument pretype t_i (for $i \in 1..n$) into a valid type T_i , and
- resolve the return pretype t into a valid type T .

We now have all the ingredients to define the typing rules for lambda terms and applications.

Definition 26 (Typing Rules for Function Instances and Applications)

We define the typing rules of Hygge0 with functions and applications by extending [Definition 11](#) with the following rules (which use the function type introduced in [Definition 24](#) above):

$$\frac{x_1, \dots, x_n \text{ pairwise distinct} \quad \forall i \in 1..n : \Gamma \vdash t_i \triangleright T_i \quad \{\Gamma \text{ with Vars} + \{x_i \mapsto T_i\}_{i \in 1..n}\} \vdash e : T}{\Gamma \vdash \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e : (T_1, \dots, T_n) \rightarrow T} \text{ [T-Fun]}$$

$$\frac{\Gamma \vdash e : (T_1, \dots, T_n) \rightarrow T \quad \forall i \in 1..n : \Gamma \vdash e_i : T_i}{\Gamma \vdash e(e_1, \dots, e_n) : T} \text{ [T-App]}$$

The rules in [Definition 26](#) work as follows.

- Rule [T-Fun] type-checks a lambda term “ $\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ ” in a typing environment Γ . In its premises, the rule ensures that all argument variables x_1, \dots, x_n are distinct from each other, and each argument pretype t_i (for $i \in 1..n$) is resolved into a valid type T_i . Then, the rule recursively checks whether the function body e has type T , in a typing environment that extends Γ with the argument variables x_1, \dots, x_n and their corresponding types: this allows the function body e to make use of the function argument variables. If all these premises hold, then the rule concludes that the lambda term has the function type $(T_1, \dots, T_n) \rightarrow T$.
- Rule [T-App] type-checks an application “ $e(e_1, \dots, e_n)$ ”. The first premise of the rule checks whether e (the expression being applied) has a function type $(T_1, \dots, T_n) \rightarrow T$. Notice that:
 - the number of argument types of the function type must be n , which is also the number of arguments in the application; and
 - the return type T must be the same type used in the conclusion of the rule.

The second premise of the rule checks whether each application argument e_i has type T_i (for $i \in 1..n$). If all these premises hold, it means that the application involves an actual function which is given the correct number of arguments, all having the expected types; therefore, the rule concludes that the application has type T (i.e. the return type of the function being applied).

Example 38 (Type-Checking a Program with Functions and Applications)

Consider this Hygge expression, that defines a function f and applies it:

$$\text{fun } f(x : \text{int}, y : \text{int}) : \text{int} = \\ \quad x + y; \\ f(2, 3)$$

To type-check this expression, we first need to desugar it (according to [Definition 21](#)), thus obtaining:

$$\text{let } f : (\text{int}, \text{int}) \rightarrow \text{int} = \\ \quad \text{fun } (x : \text{int}, y : \text{int}) \rightarrow \\ \quad \quad x + y; \\ f(2, 3)$$

For brevity, let us define the following typing environments:

$$\Gamma = \left\{ \begin{array}{l} \text{Vars} = \emptyset \\ \text{TypeVars} = \emptyset \\ \text{Mutables} = \emptyset \end{array} \right\} \quad \Gamma' = \left\{ \begin{array}{l} \text{Vars} = \{x \mapsto \text{int}, y \mapsto \text{int}\} \\ \text{TypeVars} = \emptyset \\ \text{Mutables} = \emptyset \end{array} \right\} \quad \Gamma'' = \left\{ \begin{array}{l} \text{Vars} = \{f \mapsto (\text{int}, \text{int}) \rightarrow \text{int}\} \\ \text{TypeVars} = \emptyset \\ \text{Mutables} = \emptyset \end{array} \right\}$$

Then, we have the following typing derivation for the desugared expression:

$$\frac{\frac{\frac{\Gamma \vdash \text{"int"} \triangleright \text{int} \quad \dots}{\Gamma \vdash \text{"int"} \triangleright \text{int}} \quad \dots}{\Gamma \vdash \text{"(int, int) \to int"} \triangleright (\text{int}, \text{int}) \to \text{int}} \quad \text{[TRes-Int]} \quad \text{[TRes-Fun]} \quad \frac{\frac{\Gamma \vdash \text{"(int, int) \to int"} \triangleright (\text{int}, \text{int}) \to \text{int} \quad \dots}{\Gamma \vdash \text{"fun (x : int, y : int) \to x + y"} \triangleright (\text{int}, \text{int}) \to \text{int}} \quad \text{[TRes-Int]} \quad \text{[T-Fun]} \quad \frac{\frac{\frac{\Gamma'(x) = \text{int}}{\Gamma' \vdash x : \text{int}} \quad \text{[T-Var]} \quad \frac{\Gamma'(y) = \text{int}}{\Gamma' \vdash y : \text{int}} \quad \text{[T-Var]}}{\Gamma' \vdash x + y : \text{int}} \quad \text{[T-Add]} \quad \frac{\Gamma''(f) = (\text{int}, \text{int}) \rightarrow \text{int}}{\Gamma'' \vdash f : (\text{int}, \text{int}) \rightarrow \text{int}} \quad \text{[T-Var]} \quad \frac{\Gamma'' \vdash 2 : \text{int}}{\Gamma'' \vdash 2 : \text{int}} \quad \text{[T-Val-Int]} \quad \frac{\Gamma'' \vdash 3 : \text{int}}{\Gamma'' \vdash 3 : \text{int}} \quad \text{[T-Val-Int]}}{\Gamma'' \vdash f(2, 3) : \text{int}} \quad \text{[T-App]}}{\Gamma \vdash \text{let } f : (\text{int}, \text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y; f(2, 3) : \text{int}} \quad \text{[T-Let2]}$$

Let us explore the derivation above, going bottom-up. We begin with the instance of rule [T-Let2] (from [Definition 16](#)).

- The first premise of [T-Let2] resolves the function pretype into an actual function type $(\text{int}, \text{int}) \rightarrow \text{int}$, using [T-Res-Fun] from [Definition 25](#).
- The second premise of [T-Let2] checks whether the expression that initialises f has the expected type $(\text{int}, \text{int}) \rightarrow \text{int}$. Since that expression is a lambda term, we type-check it using rule [T-Fun] from [Definition 26](#), which:
 - resolves each pretype assigned to the lambda term arguments x and y ;
 - type-checks the body of the lambda term (i.e. the expression $x + y$) in a typing environment Γ' , where x and y are mapped to their type int ;
 - concludes that the lambda term has type $(\text{int}, \text{int}) \rightarrow \text{int}$;
- The third premise of [T-Let2] type-checks the expression in the scope of the “let...”, i.e. $f(2, 3)$. To this end, it uses a typing environment Γ'' where the declared variable f maps to its type $(\text{int}, \text{int}) \rightarrow \text{int}$; moreover, since $f(2, 3)$ is an application, we type-check it using rule [T-App] from [Definition 26](#), which:
 - checks whether the expression being applied (i.e. f) has a function type;

- checks whether f is applied to exactly two arguments (as expected by its function type), and whether each argument (the expressions 2 and 3) has the expected type (int); and
- concludes that the application $f(2, 3)$ has type int (i.e. the return type of f).

Since all premises of this instance of [T-Let2] hold, the rule concludes that the whole expression has type int.

6.5 The RISC-V Memory Layout, Stack, and Calling Convention

To extend hyggec with functions and applications (according to the specification above) we follow the usual steps, that we discuss later (in the *Implementation* section).

However, the code generation step is far from trivial: in order to generate the correct code for functions and applications, we need to be aware of *The RISC-V Memory Layout*, and the details of *Implementing Functions in RISC-V* — in particular, the *RISC-V calling convention*. We now discuss all these topics.

6.5.1 The RISC-V Memory Layout

When discussing the *RISC-V Assembly Program Structure* we mentioned that, when a program runs on a RISC-V architecture, its memory is divided into **segments** — and we highlighted the **data segment** and the **text segment**. Fig.6.1 shows a more detailed view.

Fig.6.1 outlines the following memory layout:

- the **reserved** memory area (placed on low memory addresses) is typically used by the operating system;
- the **text segment** contains the program code; the **program counter (pc)** register should always point to a memory address within this segment;
- the **data segment** contains statically-allocated program data (e.g. constant strings and values);
- the **heap** is a memory area for data allocated *dynamically*, while the program is running. The heap grows upwards: new data is allocated in higher memory addresses; (we will talk about the heap in the next module)
- the **stack** is also a memory area for dynamically-allocated data — but unlike the heap, it grows downwards (from high to low memory addresses) and its data is allocated and removed in LIFO (last-in-first-out) order. Its main use is to store the local state of a running function: when a function is called, it will add its data on the stack, and then remove it before returning to the caller. When a program uses the stack, it leverages two registers:

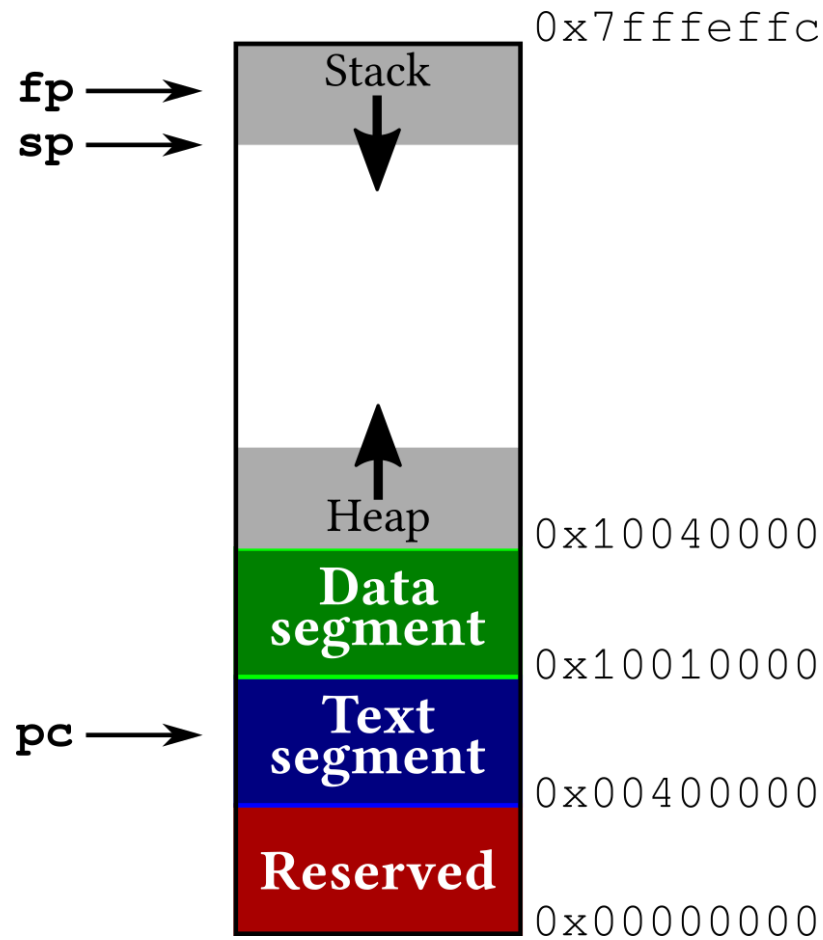


Fig. 6.1: Typical memory layout of a program running on a RISC-V system. Memory addresses are only indicative.

- the **stack pointer (sp)** register points to the last-allocated memory address on the stack;
- the **frame pointer (fp)** points to the beginning of the current **stack frame**, i.e. the portion of the stack used by the function that is currently running.

Being a register-based architecture, RISC-V favours using registers instead of memory. However, using the stack becomes hardly avoidable as soon as a program needs to call a function, or perform a system call: we now see why.

6.5.2 Implementing Functions in RISC-V

We illustrate how functions can be implemented in a RISC-V architecture by first discussing a series of scenarios of growing complexity, and concluding with *The RISC-V Calling Convention and Its Code Generation*.

Calling a Function: a Simple Case

Consider the following Hygge program, where the “main” program body calls a function `f` that immediately returns the same value.

```
fun f(x: int): int = x;  
  
f(42)
```

We could compile this program by following a simple convention on how to call a function and return a value, as follows:

- the **caller** of function `f` (i.e. the main body of the program) must:
 1. write the **function call integer arguments in the registers a0 to a7** (in this case, we only need to use register `a0` for the argument `42`);
 2. write the **return address in the register ra** (which in fact stands for “return address”). More precisely, the return address is the memory address of the assembly instruction that follows the function call `f(1, 2)`;
 3. **jump** to the memory address where the code of the function `f` begins; and
 4. when `f` returns (i.e. jumps back to the memory address in register `ra`), find its **return value in register a0**;
- the function `f` (i.e. the **callee**) must:
 1. find its **integer arguments in registers a0 to a7** (in this case, it only uses register `a0`);
 2. compute its result and write it in the **return value register a0**; and
 3. **jump** to the memory address in register `ra`, thus returning to the caller.

Calling a Function While Saving Temporary Registers

Consider the following Hygge program, where the “main” body has some local variables and computation, and calls a function `f` which also performs some computation.

```

fun f(x: int, y: int): int = {
  // ... other expressions ...
  x + y
};

let a: int = 1 + 1;
let b: int = a + 2;
// ... other expressions ...
f(a, b)
a + b

```

If we compile this program according to the simple calling convention outlined in *above*, we run into a problem. Observe that (according to the *Code Generation Strategy* of `hygge`):

- in the “main” body of the program:
 - the values of variables `a` and `b` will be written in the temporary registers `t0` and `t1`;
 - the “other expressions” may use further registers — including registers `s0`, `s1`, etc.;
- in the body of `f`:
 - the result of its “other expressions”, and the result of `x + y`, will be written (at least temporarily, before returning) in register `t0`, and possibly in other registers.

As a consequence, **the execution of function `f` will overwrite the value of register `t0` (and possibly other registers) used by its caller**; consequently, the result of the final expression `a + b` will be incorrect!

To avoid this scenario, we need to refine the *simple calling convention outlined above*: we need to **save and restore registers to preserve their value**. More concretely, the RISC-V conventions specify that some registers should be saved by the caller of a function, while others should be saved by the function being called:

- the caller of function `f` must:
 1. **save on the stack** all used registers which are marked as “caller-saved” in the table of RISC-V *Base and Floating-Point Registers*;
 2. write the integer function arguments in the registers `a0` to `a7` (in this case, we only need to use registers `a0` and `a1` for arguments 2 and 42);
 3. write the return address (i.e. the memory address of the instruction that follows the call `f(1, 2)`) in the register `ra` (return address);
 4. jump to the memory address where the code of the function `f` begins; and

5. when `f` returns (i.e. jumps back to the memory address in register `ra`)
 - restore from the stack all caller-saved registers that were saved before calling `f`;
 - find the return value of `f` in register `a0`;
- the function `f` (i.e. the callee) must:
 1. save on the stack all the registers it uses which are marked as “callee-saved” in the table of RISC-V *Base and Floating-Point Registers*;
 2. find its integer arguments in registers `a0` to `a7` (in this case, it only uses `a0` and `a1`);
 3. compute its result, and write its return value in the register `a0`;
 4. restore from the stack all callee-saved registers that were saved earlier; and
 5. jump to the return memory address in register `ra`.

A Function That Performs a System Call

Consider the following Hygge program:

```
fun f(x: int, y: int): int = {  
  let z = readInt();  
  x + y + z  
};  
  
f(1, 2)
```

Notice that, in order to perform the system call `readInt`, the function `f` needs to do the following (according to the specification of *RARS System Calls*):

- write the system call number in register `a7`, and
- find the system call result in register `a0`.

However, by doing so, the system call will overwrite the register `a0`, that holds the function argument `x`!

For this reason, when performing a system call, `f` needs to proceed as follows:

1. before the system call, save on the stack any used register between `a0` and `a7` that is also needed by the system call;
2. after the system call, restore from the stack any register between `a0` and `a7` saved earlier.

This can be seen as a special case of *Calling a Function While Saving Temporary Registers*: registers `a0` to `a7` are “caller-saved”, so they must be saved before calling a function (or performing a system call).

Calling a Function with More Than 8 Arguments

Consider the following Hygge program:

```
fun g(x1: int, x2: int, ..., x9: int, x10: int): int = {
  x1 + x2 + ... + x9 + x10;
};

g(1, 2, ..., 8, 9, 10)
```

If a function `g` takes more than 8 arguments, then registers `a0` to `a7` are not sufficient for storing all arguments needed to call `g`. Therefore, we need to further refine the calling convention outlined *above*:

- the caller of function `g` must:
 1. save on the stack all used registers which are marked as “caller-saved” in the table of RISC-V *Base and Floating-Point Registers*;
 2. write the **first 8 integer function arguments** in the registers `a0` to `a7`;
 3. **write on the stack** all integer function arguments above the 8th;
 4. write the return address (i.e. the memory address after the call `g(1, 2, ...)`) in the register `ra` (return address);
 5. jump to the memory address where the function `g` begins; and
 6. when `g` returns (i.e. jumps back to the memory address in register `ra`)
 - restore from the stack all caller-saved registers that were saved before calling `f`;
 - find the return value of `f` in register `a0`;
- the function `g` (i.e. the callee) must:
 1. **save on the stack** all the registers it uses which are marked as “callee-saved” in the table of RISC-V *Base and Floating-Point Registers*;
 2. find its **first 8 integer arguments** in registers `a0` to `a7`;
 3. find the **integer arguments above the 8th** on the stack;
 4. compute its result, and write its return value in the register `a0`;
 5. **restore from the stack all callee-saved registers** that were saved earlier; and
 6. jump to the return memory address in register `ra`.

The RISC-V Calling Convention and Its Code Generation

We can now specify in more detail the RISC-V calling convention, by illustrating how it impacts code generation. To this end, we explore:

- the *Code Generation for a Function Instance*;
- the *Code Generation for a Function Call*; and
- a *Detailed Example: the RISC-V Calling Convention in Action*.

Important: When discussing the RISC-V calling convention below, we say “integer argument” to actually mean “any function argument whose value (as represented in the compiled assembly code) can be stored in a base RISC-V integer register”. In the case of Hygge, such “integer arguments” includes *all types of values except float*: so, an “integer argument” (for code generation) includes values of type `int`, `bool` (represented as integer value 0 or 1), and also `string` (because strings are represented by their memory address). Values of type `unit` have no fixed representation (in fact, Hygge typically discards values of type `unit`), but they can harmlessly stored in integer registers.

Code Generation for a Function Instance

Consider a Hygge function instance (a.k.a. lambda term) $\text{fun } (x_1 : T_1, \dots, x_n : T_n) \rightarrow e$. When generating the assembly code for this function instance, we need to follow these steps.

1. Generate an **assembly label** to mark the memory address of the function assembly code (so it can be called later).
2. Generate the assembly code of a **function prologue** that:
 - saves on the stack all registers which are marked as **callee-saved** in the table of RISC-V *Base and Floating-Point Registers*, and
 - initialises the **frame pointer register fp** with the value of the stack pointer `sp` *before* the callee-saved registers were saved.
3. Generate the assembly the code of the **function body** e , making sure that it accesses the arguments x_1, \dots, x_n by using:
 - registers `a0...a7` for the **first 8 integer arguments**;
 - registers `fa0...fa7` for the **first 8 floating-point arguments**;
 - the **stack for the remaining arguments** (if any):
 - the first argument passed on the stack is located at the memory address contained in the frame pointer register `fp`;
 - the following arguments are stored at correspondingly higher addresses.
4. Generate the assembly the code of an **function epilogue** that:

- copies the value produced by the function body e into the register $a0$ (for integer values) or $fa0$ (for float values);
 - restores from the stack all registers that were saved in the prologue;
 - returns to the caller, by jumping to the memory address in register ra ;
5. Finally, produce the result of the whole function instance, which is the memory address marked with a label at point 1 above.

Remark 1 (Why Are We Updating the Frame Pointer Register?)

In the explanation above, the use of the frame pointer register fp may seem redundant: it is just a copy of the value of the stack pointer sp at the beginning of the function. So, why not just use sp , and keep the fp register (a.k.a. $s0$) available for other uses?

In fact, this is actually possible: using a register as a frame pointer is not necessary, and RISC-V allows us to use register fp (a.k.a. $s0$) for any other purpose. Correspondingly, compilers like `gcc` and `clang` include options (such as `-fomit-frame-pointer` and `-fno-omit-frame-pointer`) to disable or enable the use of a register as frame pointer in the generated code.

When used as frame pointer, the register fp contains the memory address of the beginning of the portion of the stack used by a function: this portion of the stack is called **function frame**. Knowing where the current function frame begins serves two main purposes.

1. Using a frame pointer register is convenient for us, compiler writers. This is because:
 - the value of the frame pointer register fp never changes while a function is being executed. Therefore, the code of the function body e (generated in point 4 above) can use fp to access e.g. arguments passed on the stack with fixed offsets: the first argument will be always at memory address $0(fp)$ (i.e. address in fp plus offset 0), the second argument at $4(fp)$ (i.e. fp plus offset 4), etc.;
 - instead, the value of the stack pointer sp changes every time the code of the function body e adds more data to the stack. For instance:
 - the first function argument passed on the stack is initially at position $0(sp)$;
 - however, if the code of e allocates 1024 bytes on the stack, the value of sp decreases by 1024: from that point, the compiler must remember that the location of the first argument on the stack is $1024(sp)$.

Keeping track of the changes of sp is possible, but it is tedious and makes the generated assembly harder to understand.
2. The frame pointer register can be used to inspect the stack of a running program, e.g. by debuggers. By reading the current value of fp , and knowing the calling convention in use, a debugger can find and identify the local data of a function:

- the arguments passed on the stack are located in memory addresses equal to or higher than fp;
 - other data is located in memory addresses lower than fp.
-

Code Generation for a Function Call

Now, consider a function application $e(e_1, \dots, e_n)$. When generating code for this function application, we need to follow these steps.

1. Generate the assembly code of e (i.e. the expression being applied as a function): this code should return a memory address produced by a function instance (see above). We will jump to that address to “call” the function.
2. Generate the assembly code of each function argument e_1, \dots, e_n .
3. Generate “before-call” assembly code that:
 - saves on the stack all registers which are marked as **caller-saved** in the table of RISC-V *Base and Floating-Point Registers*; (when doing this, we can omit saving the target register of the application: its value is not used and it will be overwritten by the return value of the call)
 - makes sure that the values produced by argument expressions e_1, \dots, e_n are in the right place:
 - the **first 8 integer arguments** go into registers a0...a7;
 - the **first 8 floating-point arguments** go into registers fa0...fa7;
 - the **remaining arguments are written on the stack**, in reverse order (i.e. the first argument passed on the stack is written last, in the lowest memory address, and is pointed by the sp register).
4. Generate the assembly code that performs the function call, with a **jump and link register instruction (jalr)** which does two things:
 - writes the **return address** in register ra (such return address is automatically computed by the jalr instruction, as the current value of the program counter register pc + 4); and
 - jumps to the memory address returned by expression e (see point 1 above).
5. Generate **after-call assembly code** which, immediately after the function call returns:
 - moves the result of the function call from register a0 (or fa0) into the target register for the whole function application; and
 - restores from the stack all registers that were saved in the “before-call” assembly code.

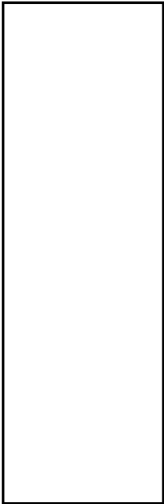
Detailed Example: the RISC-V Calling Convention in Action

Example 39 below shows how a program with a function call is executed according to the RISC-V calling convention.

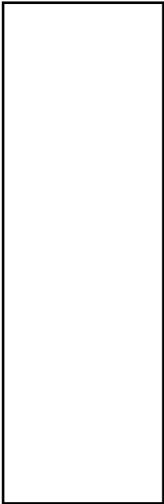
Example 39 (The RISC-V Calling Convention in Action)

Let us explore how a simple Hygge program is expected to manipulate the stack and registers, according to *The RISC-V Calling Convention and Its Code Generation*. Each execution step below is analysed by showing the current state of the program, registers, and stack, and then discussing them. In the program, the comment “//<” highlights the current execution point.

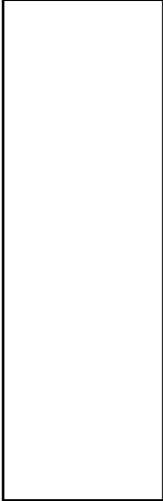
Important: Unfortunately the tables in this example are not formatted correctly in the PDF version of these lecture notes. Please refer to the HTML version, until I find a way to fix the formatting of this example. Apologies for the trouble!

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): ↪ int = { x1 + ... + ↪ x10; }; ↪ //< let x = 42; let y = f(1, ..., 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x00400000 sp = 0x7fffeffc fp = 0x7fffeffc ra = ? a0 = ? ... a7 = ? t0 = ? t1 = ? t2 = ? ... t6 = ? s1 = ? ... s4 = ? s5 = ? ... </pre>	<p>sp = fp → </p>

The code, registers, and stack above show that, at the beginning of the execution, the program counter `pc` points to the first instruction of the program, and the stack pointer `sp` points at the beginning of the stack. The frame pointer `fp` is initialised with the same value of `sp`. The other registers may be uninitialised, hence they may contain arbitrary values (which we ignore).

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): ↪ int = { x1 + ... + ↪ x10; }; let x = 42; ↪ //< let y = f(1, ..., 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x00400004 sp = 0x7fffeffc fp = 0x7fffeffc ra = ? a0 = ? ... a7 = ? t0 = 42 t1 = ? t2 = ? ... t6 = ? s1 = ? ... s4 = ? s5 = ? ... </pre>	<p>sp = fp →  0x7fffeffc</p>

After the first execution step, the program loads value 42 in register t0, and the program counter pc advances to the next instruction.

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> ↪ int = { x1 + ... + <u>x10</u> ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x) // < print(x + y) </pre>	<pre> pc = 0x00400030 sp = 0x7fffeffc fp = 0x7fffeffc ra = ? a0 = ? ... a7 = ? t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>sp = fp →  0x7fffeffc</p>

Then, the program loads the memory address of function `f` in register `t1`, and the values of the function call arguments in registers `t2...t6` and `s1...s5`.

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> ↪ int = { x1 + ... + <u>x10</u>; ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x) // < print(x + y) </pre>	<pre> pc = 0x00400080 sp = 0x7ffefbc fp = 0x7ffeffc ra = ? a0 = ? ... a7 = ? t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	

Now, the program starts preparing the function call: it executes the **before-call** code that saves on the stack all registers that are “caller-saved” (according to the table of *RISC-V Base and Floating-Point Registers*). Such caller-saved registers consist of:

- the argument registers `a0...a7`,
- the return address register `ra`, and
- the temporary registers `t0...t6`.

(For simplicity, here we are only considering integer registers, and we are ignoring the fact that some of them are not currently in use, so saving them is not necessary.)

Observe that the stack pointer `sp` decreases by 64 bytes (remember that the stack grows downwards) to make room for saving such 16 registers (of 32 bits each); the frame pointer `fp` is not changed.

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): ↪ int = { x1 + ... + ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x)//< print(x + y) </pre>	<pre> pc = 0x004000a0 sp = 0x7ffefbc fp = 0x7ffeffc ra = ? a0 = 1 ... a7 = 8 t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>The stack diagram shows a vertical stack of memory. At the top, the frame pointer fp points to the address <code>0x7ffeffc</code>. Below this, the registers <code>a0...a7</code> are shown in a grey box. Below that, the return address <code>ra</code> is shown in a grey box. Below <code>ra</code>, the temporary registers <code>t0...t6</code> are shown in a grey box. The stack pointer sp points to the address <code>0x7ffefbc</code>, which is the address immediately below the <code>t0...t6</code> register box. The area below <code>sp</code> is empty, representing the rest of the stack.</p>

The program keeps preparing the function call: it copies the first 8 arguments of the call into registers `a0...a7`.

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> ↪ int = { x1 + ... + <u>x10</u>; ↪ x10; }; let x = 42; let y = f(1, ..., 9, ↪ x) // < print(x + y) </pre>	<pre> pc = 0x004000ac sp = 0x7fffefb4 fp = 0x7fffeffc ra = ? a0 = 1 ... a7 = 8 t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>Stack diagram showing memory layout. The stack grows downwards. The frame pointer fp points to address <code>0x7fffeffc</code>, which is the start of the <code>a0...a7</code> register window. Below it is the <code>ra</code> register window. Below that is the <code>t0...t6</code> register window. Below that are two argument slots, <code>Arg #10</code> and <code>Arg #9</code>, which are highlighted in yellow. The stack pointer sp points to address <code>0x7fffefb4</code>, which is the start of the <code>Arg #9</code> slot. Below <code>Arg #9</code> is an empty slot, and below that is another empty slot.</p>

Next, the program copies the function call arguments above the 8th on the stack, in reverse order (i.e. first argument passed on the stack becomes last, getting a lower memory address.). To this purpose, the program advances the stack pointer `sp` by 8 bytes (to make room for 2 arguments, taking 4 bytes each). (Notice that the frame pointer `fp` does not change).

Program	Registers	Stack
<pre> fun f(x1: int, ↪ //< ..., x10: int): ↪int = { x1 + ... + ↪x10; }; let x = 42; let y = f(1, ... 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x00401000 sp = 0x7ffefb4 fp = 0x7ffeffc ra = 0x004000b0 a0 = 1 ... a7 = 8 t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>The stack diagram shows a vertical stack of memory. At the top, the frame pointer fp points to address 0x7ffefffc. Below this, a stack frame is shown with the following components from top to bottom: registers a0...a7, register ra, registers t0...t6, Arg #10, and Arg #9. The stack pointer sp points to address 0x7ffefb4, which is the bottom of the stack frame. The stack grows downwards.</p>

Now the program performs the function call, invoking the RISC-V assembly instruction `jalr ra, 0(t1)`, which means:

1. save in register `ra` the return address (which is the value of `pc + 4`), and
2. jump to the memory address computed by taking the value of register `t1` and adding the offset `0`.

After `jalr ra, 0(t1)` is executed, we have that:

- `pc` is updated with the memory address in `t1` (plus offset `0`), and
- the old value of `pc + 4` is written in register `ra`.

Program	Registers	Stack
<pre> fun f(x1: int, ↪ //< ..., x10: int): ↪ int = { x1 + ... + ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x00401038 sp = 0x7ffef84 fp = 0x7ffeffc ra = 0x004000b0 a0 = 1 ... a7 = 8 t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	

Next, the called function executes its **prologue**, which decreases the stack pointer `sp` by 48 bytes to save its callee-saved registers. Such callee-saved registers consist of:

- the frame pointer `fp`,
- the temporary registers `s1...s11`,
- the stack pointer `sp` — although we do not need to actually save `sp` on the stack, because we can easily compute its old value later.

(For simplicity, here we are only considering integer registers, and we ignore the fact the function `f` may not use use all registers between `s1` and `s11`, so saving all of them might not be necessary.)

Program	Registers	Stack
<pre> fun f(x1: int, ↪ //< ..., x10: int): ↪ int = { x1 + ... + ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x0040103c sp = 0x7ffef84 fp = 0x7fff0b4 ra = 0x004000b0 a0 = 1 ... a7 = 8 t0 = 42 t1 = 0x00401000 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>Stack diagram showing registers and their addresses:</p> <ul style="list-style-type: none"> a0...a7: 0x7ffefffc ra: 0x7ffeffbc t0...t6: 0x7ffefb4 Arg #10: 0x7ffefb4 Arg #9: 0x7ffefb4 fp: 0x7ffefb4 s1...s11: 0x7ffef84 <p>Registers fp and sp are shown with arrows pointing to their respective values in the stack diagram.</p>

Then, the function prologue updates the frame pointer `fp` to the value of the stack pointer `sp` *before* its last update (i.e. before the callee-saved registers were saved on the stack): in this example, the frame pointer is set to `sp + 48`.

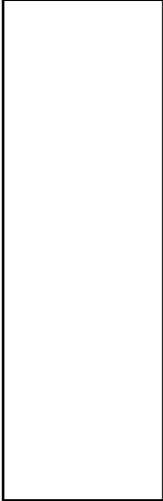
Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> ↪ int = { x1 + ... + <u>x10</u> ↪ x10; //< }; let x = 42; let y = f(1, ..., 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x0040106c sp = 0x7fffef84 fp = 0x7ffff0b4 ra = 0x004000b0 a0 = 1 ... a7 = 8 t0 = 87 t1 = 1 t2 = 2 ... t6 = 6 s1 = 7 ... s4 = 42 s5 = 42 ... </pre>	<p>Stack diagram showing memory layout:</p> <ul style="list-style-type: none"> Address 0x7fffeffc: a0...a7 Address 0x7fffefbc: ra Address 0x7fffefbc: t0...t6 Address 0x7fffefb4: Arg #10 Address 0x7fffefb4: Arg #9 Address 0x7fffefb4: fp Address 0x7fffef84: s1...s11 <p>Registers fp and sp are shown with arrows pointing to their respective stack frames.</p>

Now, the body of the function begins its execution. To compute the sum, it will load its arguments from registers `r0...r7` and from the stack, and write the result of their addition (which is 87) in the target register `t0`. As a consequence, the values in registers `t0...t6` and `s1...s4` are overwritten.

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> { ↪ int = { x1 + ... + <u>x10</u>; //< }; let x = 42; let y = f(1, ..., 9, ↪ x); print(x + y) </pre>	<pre> pc = 0x004010ac sp = 0x7fffefb4 fp = 0x7ffefffc ra = 0x004000b0 a0 = 87 ... a7 = 8 t0 = 87 t1 = 1 t2 = 2 ... t6 = 6 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	

The function assembly code can now prepare for returning back to the caller, by executing the function **epilogue**, which performs the following steps:

- it copies the result of the function body (available in its target register `t0`) into the return value register `a0`;
- it restores all callee-saved registers to the values they had before the function was called:
 - it retrieves the old values of registers `fp` and `s1...s11` from the stack, and
 - it restores the old value of `sp` by adding the number of restored registers multiplied by 4 (their size) to the current value of `sp` (hence, by adding $12 * 4 = 48$ to the current value of `fp`).

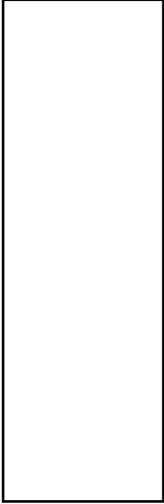
Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): <u>int</u> { ↪ int = { x1 + ... + <u>x10</u>; ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x) // < print(x + y) </pre>	<pre> pc = 0x0040110c sp = 0x7fffeffc fp = 0x7fffeffc ra = ? a0 = ? ... a7 = ? t0 = 42 t1 = 87 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>sp = fp → </p>

Now the execution jumps back to the caller, i.e. to the address stored in register ra — which is the first instruction after the function call. That instruction is the beginning of the **after-call** assembly code, which:

- copies the function's return value from register a0 to the target register for the call (in this example, t1), and
- restores all caller-saved registers *excluding the target register t1* from the stack, and
- updates the stack pointer by adding to the current value of sp:
 - the number of restored registers (16) multiplied by 4 (their size), and
 - the number of arguments passed on the stack (2) multiplied by 4 (their size).

Observe that now all registers are back to the same values they had before the function call — with the exception of the program counter pc (which has advanced to the current

instruction) and the target register t1 (which contains the return value of the function call).

Program	Registers	Stack
<pre> fun f(x1: int, ..., x10: int): int { ↪ int = { x1 + ... + ↪ x10; }; let x = 42; let y = f(1, ... 9, ↪ x); print(x + y) ↪ //< </pre>	<pre> pc = 0x00401110 sp = 0x7fffeffc fp = 0x7fffeffc ra = ? a0 = ? ... a7 = ? t0 = 42 t1 = 87 t2 = 1 ... t6 = 5 s1 = 6 ... s4 = 9 s5 = 42 ... </pre>	<p>sp = fp →  0x7fffeffc</p>

Now function call is complete and the execution can continue regularly: the code generation for the expression `print(x, y)` will find the values of variables `x` and `y` in registers `t0` and `t1`, respectively.

6.6 Implementation

We now have a look at how `hygdec` is extended to implement function instances (i.e. lambda terms) and applications, according to the specification illustrated in the previous sections.

Tip: To see a summary of the changes described below, you can inspect the differences in the [hygdec Git repository](#) between the tags `while` and `functions`.

6.6.1 Lexer, Parser, Interpreter, and Type Checking

These parts of the `hygdec` compiler are extended along the lines of *Example: Extending Hygge0 and hygdec with a Subtraction Operator*.

- We extend `AST.fs` in two ways, according to *Definition 21*:
 - we extend the data type `Expr<'E, 'T>` with two new cases:
 - * Lambda for “`fun (x1 : t1, ..., xn : tn) → e`”, and
 - * Application for “`e (e1, ..., en)`”;
 - we also extend the data type `Pretype` with a new case called `TFun`, corresponding to the new function pretype “`(t1, ..., tn) → t`”:

```
and Pretype =
// ...
/// A function pretype, with argument pretypes and return pretype.
| TFun of args: List<PretypeNode>
        * ret: PretypeNode
```

- We extend `PrettyPrinter.fs` to support the new expressions `Lambda` and `Application`, and the new pretype `TFun`.
- We extend `Lexer.fsl` to support three new tokens:
 - `FUN` for the new keyword “`fun`”;
 - `RARROW` (right arrow) for the arrow “`->`” used to define lambda terms and function pretypes; and
 - `COMMA` for the symbol “`,`” used to separate arguments in function instances, applications, and pretypes.

Warning: If you have implemented `min` and `max` as part of *Project Idea: Extend Hygge0 and hygdec with New Arithmetic Operations*, then you should have already introduced a token corresponding to `COMMA`: you should reuse it (i.e. you should not have two tokens that match the same sequence of input characters).

- We extend `Parser.fsy` to recognise the desired sequences of tokens according to [Definition 21](#), and generate AST nodes for the new expressions. We proceed by adding:
 - a new rule under the `simpleExpr` category to generate `Lambda` instances;
 - a new rule under the `unaryExpr` category to generate `Application` instances;
 - a new rule under the `pretype` category to generate `TFun` pretype instances;
 - various auxiliary syntactic categories and rules to recognise the syntactic elements needed by the rules above. For instance:
 - * `parenTypesSeq` is a sequence of comma-separated pretypes, between parentheses (needed to parse function pretypes);
 - * `parenArgTypesSeq` is a sequence of comma-separated variables with type ascriptions, between parentheses (needed to parse the arguments of a lambda term);
 - finally, we also add a new rule to implement the syntactic sugar for simplified function definitions. To this purpose, we extend the `expr` syntactic category (because our simplified function definitions are just “let...” binders, so we want to give them the same precedence). The syntactic sugar rule looks as follows:

```

expr:
  // ...
  | FUN ident parenArgTypesSeq COLON pretype EQ simpleExpr SEMI expr {
    let (_, args) = List.unzip $3 // Extract argument pretypes
    mkNode(..., Expr.Let($2, mkPretypeNode(..., Pretype.TFun(args,
    ↪$5)),
                                mkNode(..., Lambda($3, $7)), $9))
  }

```

In other words, when the parser sees an expression like `fun f(x: int, y: int): int = x + 1`, it creates a `Let` expression instance that binds the variable `f` (with pretype `(int, int) -> int`) with the lambda term `fun (x: int, y: int) -> x + 1` (as expected by [Definition 21](#)).

- We extend the function `subst` in `ASTUtil.fs` to support the new expressions `Lambda` and `Application`, according to [Definition 22](#).
- We extend `Interpreter.fs` according to [Definition 23](#):
 - in the function `isValue`, we add a new case for `Lambda` (which is a value); and
 - in the function `reduce`:
 - * we add two new cases for `Lambda` and `Application`, and
 - * we adjust the case for `LetMut` to match the rule [R-LetM-Res2].
- We extend `Type.fs` by adding a new case to the data type `Type`, according to [Definition 24](#): the new case is called `TFun`. We also add a corresponding new case to the function `freeTypeVars` in the same file.

Note: Correspondingly, we also extend the pretty-printing function `formatType` in `PrettyPrinter.fs`, to display the function type we have just introduced.

- We extend `Typechecker.fs`:
 - we extend the type resolution function `resolvePretype` with a new case for function types, according to *Definition 25*;
 - we extend the function typer according to *Definition 26*, to support the new cases for the expressions `Lambda` and `Application`.
- As usual, we add new tests for all compiler phases.

6.6.2 Code Generation

This is certainly the trickiest part of the extension, because the *RISC-V calling convention* has many moving parts.

The extension of the function `doCodegen` (in the file `Typechecker.fs`) consists of the following main elements (detailed in the following sections):

- *Code Generation for Lambda Terms*
- *Code Generation for Named Functions*
- *Code Generation for Applications*

Code Generation for Lambda Terms

We add a new case to the function `doCodegen` to handle `Lambda` expressions, i.e. function instances. The key intuition is that, in order to turn a lambda term into an actual RISC-V function, according to our discussion on *Code Generation for a Function Instance*, we need to generate assembly code to:

1. place a RISC-V assembly **label to mark the memory address of the beginning of the function code**: we will use that label to call the function;
2. generate code for the **function prologue**, to save the callee-saved registers and update the frame pointer;
3. generate code for the **function body**;
4. generate code for the **function epilogue**, to restore the caller-saved registers and copy the function result onto the return register `a0`;
5. make sure that all the function code is placed **at the end of the assembly code generated by the compiler** (because the execution starts from the beginning of the code, and we only want the function to run when it is called/applied);
6. finally, place in the target register the memory address where the function is located (i.e. the label generated at point 1 above).

Note: As a consequence of points 1 and 5 above, the label and the assembly code of a function instance are globally visible in the generated RISC-V assembly – even when the original function instance was only visible in a limited scope (e.g. like the function `privateFun` in *Example 34*). To avoid potential clashes, the compiler must ensure that the label assigned to each function is unique across the generated assembly file.

The `doCodegen` case for `Lambda` looks as follows. (Note: the actual code on the `hyggec` Git repository contain many more comments, and you should read it.)

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | Lambda(args, body) ->
    let funLabel = Util.genSymbol "lambda" // Position of lambda term body
    let (argNames, _) = List.unzip args // Names of lambda term arguments

    // Pairs of arguments and types
    let argNamesTypes = List.map (fun a -> (a, body.Env.Vars[a])) argNames

    let bodyCode = compileFunction argNamesTypes body env // Body code

    let funCode = // Complete function code, with label placed before it
      Asm(RV.LABEL(funLabel), "Lambda function code")
      ++ bodyCode
      .TextToPostText // Move this code at the end of the text segment

    // Finally, load the function address (label) in the target register
    Asm(RV.LA(Reg.r(env.Target), funLabel), "Load lambda function address")
    ++ funCode
```

Steps 2, 3, and 4 are handled by the auxiliary function `compileFunction`: its code is not reported here, but it has plenty of comments: you should read it to see what it does.

Exercise 29 (Understanding Code Generation for Function Instances)

Write a file called e.g. `fun.hyg` containing a simple function instance, such as:

```
fun (x: int, y: int) -> x + y
```

Invoke `./hyggec compile fun.hyg` and observe the generated assembly. Using the comments in the generated assembly, identify the corresponding parts of the functions `doCodegen` (case for `Lambda`) and `compileFunction`.

Warning: The code of `compileFunction` is very limited, because:

- it only supports functions that take up to 8 arguments, passed via integer registers; and

- it only supports functions that return their value through an integer register; and
- in addition, it has the *overall limitations* that we discuss later.

Some of the missing features (e.g. taking more than 8 integer arguments, taking and returning float values) are part of the *Project Ideas* for this module.

Code Generation for Named Functions

The *Code Generation for Lambda Terms* (described above) assigns a random label to the mark the location of the function's compiled assembly code. This is not an issue for lambda terms, which are anonymous. However, Hygge programs may define functions by giving them a specific name, e.g. by using the syntactic sugar:

```
fun add(x: int, y: int): int = x + y;
add(1, 2)
```

which expands into:

```
let add: (int, int) -> int =
  fun (x: int, y: int) ->
    x + y;
add(1, 2)
```

The function `doCodeGen` includes a dedicated case that matches such “let...” bindings that directly use a lambda term to initialise a variable. In this case, `hyggec` performs code generation as follows:

1. it uses the name of the variable to generate the function label in the assembly code; and
2. when assigning storage information, it maps the variable directly to the assembly label of the function (instead, the standard code generation for “let...” would store the variable in a register, hence it would use one additional register).

The code for this case of `doCodeGen` looks as follows (and it is essentially a blend between the case for `Let` and the *Code Generation for Lambda Terms*):

```
let rec internal doCodeGen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | Let(name, _,
    {Node.Expr = Lambda(args, body);
     Node.Type = TFun(targs, _)}, scope) ->
    let funLabel = Util.genSymbol $"fun_{s{name}}" // Function label
    let (argNames, _) = List.unzip args // Names of lambda term arguments

    let argNamesTypes = List.zip argNames targs // Pairs of argument & type
```

(continues on next page)

(continued from previous page)

```

let bodyCode = compileFunction argNamesTypes body env // Compiled body

let funCode = // Compiled function code, with label placed in front
  (Asm(RV.LABEL(funLabel), $"Code for function '%s{name}'")
    ++ bodyCode).TextToPostText

// Storage info with name of compiled function pointing to 'funLabel'
let varStorage2 = env.VarStorage.Add(name, Storage.Label(funLabel))

// Compile the 'let...' scope with newly-defined function visible
(doCodegen {env with VarStorage = varStorage2} scope)
  ++ funCode

```

Exercise 30 (Understanding Code Generation for Named Functions)

Write a file called e.g. `fun-named.hyg` containing a simple named function instance, such as:

```

fun add(x: int, y: int): int = x + y;
()

```

Invoke `./hygcec compile fun-named.hyg` and observe the generated assembly. Notice the difference with [Exercise 29](#). Using the comments in the generated assembly, identify the corresponding parts of the functions `doCodegen` (case for `Let` with lambda terms) and `compileFunction`.

Code Generation for Applications

We add a new case to the function `doCodegen` to handle `Application` expressions, i.e. function calls. The key intuition is that, in order to turn an application into an actual RISC-V function call, according to our discussion on [Code Generation for a Function Call](#), we need to generate assembly code to:

1. compute the term being applied as a function (e.g. retrieve the memory address of the function being applied);
2. compute each argument of the function call;
3. perform the “before-call” procedures: save the caller-saved registers, and prepare the function arguments (by copying them onto the ‘a’ registers or on the stack);
4. perform the **function call**; and
5. perform the “after-call” procedures: copy the function call result from the return register `a0` onto the target register of the call.

The `doCodegen` case for `Application` looks as follows. (Note: the actual code on the `hygcec` Git repository contain many more details and comments, and you should read it!)

```

let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | Application(expr, args) ->
    let saveRegs = // ...Saved registers (except the target register)

    let appTermCode = // ...Code for expression being applied

    let argsCode = // ...Generate code to compute each application argument

    let argsLoadCode = // ...Copy each application arg into an 'a' register

    let callCode = // Code that performs the function call
      appTermCode
      ++ argsCode // Code to compute each argument of the function call
      .AddText(RV.COMMENT("Before call: save caller-saved registers"))
      ++ (saveRegisters saveRegs [])
      ++ argsLoadCode // Code to load arg values into arg registers
      .AddText(RV.JALR(Reg.ra, Imm12(0), Reg.r(env.Target)), "Call")

    let retCode = // Code that handles the function return value (if any)
      Asm(RV.MV(Reg.r(env.Target), Reg.a0),
        $"Copy function return value to target register")

    callCode // Put everything together, restore the caller-saved registers
      .AddText(RV.COMMENT("After function call"))
      ++ retCode
      .AddText(RV.COMMENT("Restore caller-saved registers"))
      ++ (restoreRegisters saveRegs [])

```

Exercise 31 (Understanding Code Generation for Function Application)

This is a follow-up to [Exercise 29](#) and [Exercise 30](#). Write a file called e.g. fun-app.hyg containing a simple function instance and application, such as:

```
(fun (x: int, y: int) -> x + y)(1, 2)
```

Invoke `./hygdec compile fun-app.hyg` and observe the generated assembly. Using the comments in the generated assembly, identify the corresponding parts of the functions `doCodegen` (case for `Application`).

Now try a similar experiment on a file with a corresponding named function definition and application, such as:

```
fun add(x: int, y: int): int = x + y;
add(1, 2)
```

Observe the differences with the assembly code generated in the previous case.

Warning: The code generation for Application expressions is very limited, because:

- it only supports functions that take up to 8 arguments, passed via integer registers; and
- it only supports functions that return their value through an integer register; and
- in addition, it has the *overall limitations* that we discuss later.

Some of the missing features (e.g. passing more than 8 integer arguments, passing and returning float values) are part of the *Project Ideas* for this module.

6.7 Limitations of the Current Specification and Code Generation

To conclude, we highlight that the treatment of functions presented in this module has relevant limitations in the support for **closures**, i.e. lambda terms that capture variables from their surrounding scope. Here is a simple program with a closure:

```
let x: int = 1;

fun addX(y: int): int = y + x; // x is captured from the surrounding scope

assert(addX(42) = 43)
```

In particular:

- the *Code Generation* does *not* support closures at all, and the code generation for examples like the one above is incorrect;
- the *Operational Semantics* and *Typing Rules* correctly support closures for immutable variables (like in the example above), but do not correctly support closures for mutable variables. As a consequence, a well-typed program that captures mutable variables may get stuck!

These limitations require further improvements of the Hygge language and hyggec compiler: we will address them later in the course.

6.8 References and Further Readings

The treatment of lambda terms, applications, and function types in Hygge is inspired by languages with functional programming support (like Java 8 or higher, Kotlin, F#, Scala, Haskell...), and its specification based on programming language theory concepts. To know more, you can refer to:

- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002. Available on DTU Findit³⁰. These chapters, in particular, may be useful:
 - Chapter 5 (The Untyped Lambda-Calculus)
 - Chapter 9 (Simply Typed Lambda-Calculus)

The RISC-V calling convention is documented here:

- <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc>

Note: For simplicity, in this module we have omitted a requirement of the RISC-V calling convention: the frame pointer address should always be aligned to 16 bytes (128 bits). This requirement is not enforced by RARS – but when targeting real RISC-V hardware, `hygdec` (or any other compiler) should keep track of the alignment of register `fp` and always increase/decrease it by multiples of 16 bytes.

6.9 Project Ideas

For your group project, you should implement at least 3 of the following project ideas:

- *Project Idea: Function Subtyping*
- *Project Idea: Recursive Functions*
- *Project Idea: Improved Implementation of the RISC-V Calling Convention: Pass and Return Floats via Registers*
- *Project Idea: Improved Implementation of the RISC-V Calling Convention: Pass more than 8 Integer (or Float) Arguments via The Stack*

6.9.1 Project Idea: Function Subtyping

The extensions described in this module does not change the subtyping judgement (*Definition 10*). As a consequence, whenever two function types are compared to see whether one is subtype of the other, the only applicable rule is [TSub-Refl], which requires the two types to be exactly equal. This leads to the restriction illustrated in *Example 40* below.

Example 40 (Consequences of Lack of Function Subtyping)

The following program does not type-check: you can see it by yourself, by saving this program in a file and running `./hygdec typecheck file.hyg`.

³⁰ <https://findit.dtu.dk/en/catalog/5c34980ed9001d2f3820637f>

```
type MyInt = int;

fun f(x: MyInt): MyInt = x + 1; // Type error!

let fAlias: (int) -> int = f; // Type error!

assert(fAlias(42) = f(42))
```

The reasons for the type errors reported by hyggec are the following:

- when defining `f`, the function should be of type `(MyInt) -> MyInt`, but its definition has type `(MyInt) -> int` (because the expression `x + 1` has type `int`, instead of `MyInt`);
- when defining `fAlias`, the initialisation value should be a function of type `(int) -> int`, but `f` has type `(MyInt) -> MyInt`.

These type errors certainly feel spurious: since `MyInt` is just an alias of `int`, those function types should be all subtypes of each other! And indeed, if we interpret the program above (using `./hyggec interpret file.hyg`), the program runs correctly and passes the assertion (in fact, there is a test case for the hyggec interpreter that corresponds to this example).

For this Project Idea, you should extend hyggec with **function subtyping**, by adding the following rule to [Definition 10](#):

$$\frac{\forall i \in 1..n : \Gamma \vdash T'_i \leq T_i \quad \Gamma \vdash T \leq T'}{\Gamma \vdash (T_1, \dots, T_n) \rightarrow T \leq (T'_1, \dots, T'_n) \rightarrow T'} \text{ [TSub-Fun]}$$

Recall that, by the Liskov Substitution Principle ([Definition 9](#)), an instance of a subtype should be safely usable whenever an instance of a larger type is required. Correspondingly rule [TSub-Fun] above says that a function type $(T_1, \dots, T_n) \rightarrow T$ is subtype of $(T'_1, \dots, T'_n) \rightarrow T'$ if:

- both function types expect n arguments;
- for all $i \in 1..n$, the argument type T'_i is subtype of the corresponding argument type T_i (in other words, the “smaller” function is more permissive in the types of arguments it accepts); and
- the return type T is subtype of the return type T' (in other words, the “smaller” function is more restrictive in the type of value it returns).

To extend hyggec with function subtyping, you will need to:

- extend the function `isSubtypeOf` in the file `Typechecker.fs`; and
- add some test cases that would *not* type-check without function subtyping (e.g. using [Example 40](#) as a starting point). Your tests should include functions that take other functions as arguments, and functions that return functions, as in [Example 34](#).

Note: If you choose this Project Idea, you will also need to add more test cases after we introduce structured data and its subtyping, in the next module.

6.9.2 Project Idea: Recursive Functions

The goal of this Project Idea is to allow a function to call itself recursively. To this end, you should:

1. extend the Hygge programming language with a “let rec...” expression, similar to F# (and similar to the existing “let...” for immutable variables); and
2. you should modify the syntactic sugar for `fun name(...)...` in `Parser.fsy` to expand into a “let rec...” expression (instead of the regular “let...”).

More in detail, you will need to implement the following specification.

First, you need to add a new expression to the Hygge syntax (notice that this introduces a new token for the symbol `rec`).

$$\begin{array}{l} \text{Expression } e ::= \dots \\ \quad | \quad \text{let rec } x : t = e; e' \end{array}$$

You will need to add a new `LetRec` expression in `AST.fs`, similar to `Let`. You will also need to update the rule that parses the syntactic sugar `fun name(...):... = ...` in `Parser.fsy`: instead of creating a `Let` AST node, the updated rule should create a `LetRec` AST node. This way, it becomes possible for a Hygge programmer to write a named function that calls itself recursively.

Then, you will need to implement the substitutions and the semantic rules for the new “let rec...” expression. For substitution, you need to add these new cases to [Definition 2](#):

$$\begin{array}{l} (\text{let rec } x : t = e_1; e_2) [x \mapsto e'] = \text{let rec } x : t = e_1; e_2 \\ (\text{let rec } y : t = e_1; e_2) [x \mapsto e'] = \text{let rec } y : t = e_1 [x \mapsto e']; e_2 [x \mapsto e'] \quad (\text{when } y \neq x) \end{array}$$

There is only one difference between the substitution above and the substitution for the regular “let x...” ([Definition 2](#)): in the first case above (i.e. when we substitute a variable x that has the same name of the variable bound by “let rec x...”), *the substitution has no effect (instead, in the regular “let x...” we propagate the substitution in the* is visible and bound in the initialisation expression, hence we “block” its substitution.

For the “let rec...” semantics, you need to add the following rule to [Definition 4](#):

$$\frac{v' = v [x \mapsto (\text{let rec } x : t = v; x)]}{\langle R \bullet \text{let rec } x : t = v; e \rangle \rightarrow \langle R \bullet e [x \mapsto v'] \rangle} \quad [\text{R-LetRec-Subst}]$$

The difference between rule `[R-Let-Subst]` (in [Definition 4](#)) and rule `[R-LetRec-Subst]` above is that:

- in rule `[R-Let-Subst]`, the variable being defined (x) is replaced by the initialisation value v in the scope of the “let...” expression;

- in rule [R-LetRec-Subst], instead, the variable being defined (x) is replaced by v' , which is obtained by taking the initialisation value v and replacing each occurrence of x within v with an instance of `let rec $x : t = v$; x` . Notice that this substitution can only have an effect when v is a lambda term that contains instances of x . (See [Example 41](#) below.)

Then, you will need to implement the following typing rule for the new “let rec...” expression:

$$\frac{\Gamma \vdash t \triangleright T \quad \Gamma' = \{\Gamma \text{ with Vars} + (x \mapsto T) \text{ and Mutables} \setminus \{x\}\} \quad \Gamma' \vdash e_1 : T \quad \Gamma' \vdash e_2 : T'}{\Gamma \vdash \text{let rec } x : t = e_1; e_2 : T'} \quad [\text{T-LetRec}]$$

The new rule [T-LetRec] above is similar to [T-Let2] in [Definition 16](#), except that:

- in [T-Let2], the initialisation expression e_1 is typed with the typing environment Γ . As a consequence, e_1 cannot refer to the same variable x being defined by the “let...” expression;
- in [T-RetLec] above, instead, the initialisation expression e_1 is typed with the extended typing environment Γ' , which includes the type of x (and the same Γ' is also used to type e_2). As a consequence, e_1 can be well-typed even if it recursively refers to x , i.e. the variable being defined by the “let rec...”.

Finally, you will need to extend `doCodegen` (in `RISCVCodegen.fs`) as follows.

1. You should add a new case for compiling the new `LetRec` expression. This should be similar to the `Let` code generation, except that the variable being defined should be made visible in `env.VarStorage` when compiling the initialisation expression.
2. You should also add a new special case for compiling functions with a given name (which are generated by the revised `fun name(...)...` syntactic sugar). This should be similar to the existing analogous special case for `Let`, but it should match a `LetRec` expression instead, and compile the function by making its name visible by the function body.

As usual, you should write tests for all the compiler phases you modified.

Example 41 (Reductions of a Recursive Function)

Consider the following Hygge program, where function f calls itself:

```
fun f(x) : int = f(x + 1);
f(0)
```

Intuitively, this expression should execute forever by calling $f(0)$, then $f(0 + 1)$, then $f(1 + 1)$...

According to this Project Idea, the expression above should desugar the function definition into a “let rec...” expression:

```
let rec f : (int) → int =
  fun (x : int) → f(x + 1);
  f(0)
```

If we reduce this expression in a runtime environment R , using the new rule [R-LetRec-Subst], we get:

$$v' = (\text{fun } (x : \text{int}) \rightarrow f(x + 1)) \left[f \mapsto \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f \right]$$

$$\left\langle R \bullet \left[\begin{array}{l} \text{let rec } f : (\text{int}) \rightarrow \text{int} = \\ \text{fun } (x : \text{int}) \rightarrow f(x + 1); \\ f(0) \end{array} \right] \right\rangle \rightarrow \left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (0) \right\rangle$$

[R-LetRec-Subst]

The next reduction performs the top-level function application:

$$\left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (0) \right\rangle \rightarrow \left\langle R \bullet \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (0 + 1) \right\rangle$$

[R-App-Res]

We now have another application, and we need to reduce the expression being applied into a value. To this end, we use rule [R-LetRec-Subst] again:

$$v' = (\text{fun } (x : \text{int}) \rightarrow f(x + 1)) \left[f \mapsto \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f \right]$$

$$\left\langle R \bullet \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (0 + 1) \right\rangle \rightarrow \left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (0 + 1) \right\rangle$$

[R-App-Eval-R1]

We now need to reduce the application argument into a value:

$$\frac{0 + 1 = 1}{\langle R \bullet 0 + 1 \rangle \rightarrow \langle R \bullet 1 \rangle} \text{ [R-Add-Res]}$$

$$\left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (0 + 1) \right\rangle \rightarrow \left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (1) \right\rangle$$

[R-App-Eval-R1]

And by performing the application, we get:

$$\left\langle R \bullet \left(\text{fun } (x : \text{int}) \rightarrow \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (x + 1) \right) (1) \right\rangle \rightarrow \left\langle R \bullet \left(\text{let rec } f : (\text{int}) \rightarrow \text{int} = \text{fun } (x : \text{int}) \rightarrow f(x + 1); \right)_f (1 + 1) \right\rangle$$

[R-App-Res]

Observe that this last reduction produced an expression that is very similar to the one we obtained with the first use of rule [R-App-Res] above — except that the first time we obtained an application of “let rec...” to $(0 + 1)$, while now we have obtained an application of the same “let rec...” to $(1 + 1)$. Therefore, we can continue reducing forever, and every time we will obtain an application of the same “let rec...” to a bigger value.

6.9.3 Project Idea: Improved Implementation of the RISC-V Calling Convention: Pass and Return Floats via Registers

The goal of this Project Idea is to extend hyggec to support functions that receive floating-point arguments, or return a floating-point value. To this end, you should follow *The RISC-V Calling Convention and Its Code Generation*, and improve the hyggec *Code Generation*. Note that your extension should support function that receive a mix of integer and float arguments, in any order, such as:

```
fun f(w: int, x: float, y: bool, z: string): float = {
  if y then println(z) else println(w+1);
  x + 42.0f
};
println(f(2, 1.0f, true, "Hello"));
println(f(2, 2.0f, false, "Hello"))
```

6.9.4 Project Idea: Improved Implementation of the RISC-V Calling Convention: Pass more than 8 Integer (or Float) Arguments via The Stack

The goal of this Project Idea is to extend hyggec to support functions that receive more than 8 integer arguments, by passing the arguments above the 8th on the stack. To this end, you should follow *The RISC-V Calling Convention and Its Code Generation*, and improve the hyggec *Code Generation*.

Hint: To implement this project idea, you will need to extend the type `Storage` in `RISCVCodeGen.fs` with a new case, describing a variable that can be found on the stack. The new case for `Storage` may look as follows:

```
type internal Storage =
  // ...
  /// This variable is stored on the stack, at the given offset (in bytes)
  /// from the memory address contained in the frame pointer (fp) register.
  | Frame of offset: int
```

Note: If you have also selected the *Project Idea on passing/returning floats*, keep in mind that what you should pass on the stack are:

- all integer arguments above the 8th *integer* argument, and
- all float arguments above the 8th *float* argument.

As a consequence, a call to the following function should *not* pass any argument via the stack:

```
fun f(x1: int, x2: int, x3: int, x4: int, x5: int,  
      y1: float, y2: float, y3: float, y4: float, y5: float): bool = {  
    // ...  
}  
//...
```

This is because, although `f` has 10 arguments in total, it takes less than 9 integer arguments, and less than 9 float arguments.

Module 7: Structured Data Types and the Heap

In this module we study how to extend Hygge with structured data types; more specifically, these lecture notes introduce a data type constructor called `struct`, inspired by the C programming language. A `struct` instance is a sequence of **fields**, each one having its own name and value. The `struct` data type constructor provides a blueprint for further extensions of Hygge. Notably, Hygge `structs` are modelled in a way that is very similar to Java objects: i.e. when a `struct` instance is created, the program only gets a **pointer** to the **memory heap** location where the actual data of the `struct` is written. As a consequence, `structs` are **passed and returned by reference** when invoking functions, and survive outside the scope where they are created.

7.1 Overall Objective

Our goal is to interpret, compile and run Hygge programs like the one shown in *Example 42* below.

Example 42 (A Hygge Program with Structures)

```
1 // Three type aliases for structure types
2 type Shape = struct { name: string;
3                       area: float };
4 type Circle = struct { name: string;
5                       area: float;
6                       radius: float };
7 type Square = struct { name: string;
8                       area: float;
9                       side: float };
10
11 // Function that takes a structure as argument
12 fun displayShape(shape: Shape): unit = {
13     print("Name: "); print(shape.name);
14     print("; area: "); println(shape.area)
15 };
```

(continues on next page)

(continued from previous page)

```
16
17 // Structure constructors
18 let c: Circle = struct { name = "Circle";
19                       area = 10.0f * 10.0f * 3.14f;
20                       radius = 10.0f };
21 let s: Square = struct { name = "Square";
22                       area = 2.0f * 2.0f;
23                       side = 2.0f };
24 let r: struct {name: string; area: float} =
25     struct { name = "Rectangle";
26             area = 3.0f * 4.0f;
27             width = 3.0f;
28             height = 4.0f };
29
30 // Function calls with structures as arguments. Note: the structures
31 // passed as arguments have more fields than required by the function.
32 displayShape(c);
33 displayShape(s);
34 displayShape(r);
35
36 // Assignment to structure fields
37 c.area <- s.area <- r.area <- 0.0f;
38 assert(c.area = s.area);
39 assert(s.area = r.area);
40 assert(r.area = 0.0f)
```

To introduce structured data types in the Hygge specification, and make it possible to write programs like the one in *Example 42*, we adopt a design that mixes ideas from Java, F# and TypeScript.

- We adopt a so-called **structural typing system** (inspired by [TypeScript](https://www.typescriptlang.org/docs/handbook/type-compatibility.html)³¹), where struct types are compared by their fields, and their names are just a convenience for programmers. For example: on lines 18, 21, and 24 of *Example 42*, the types of variables `c`, `s`, and `r` are not called “Circle”; still, on lines 32–34, `hyggec` allows those variables to be passed to function `displayShape`, which expects an argument of type `Circle`. This is allowed because the types of `c`, `s`, and `r` include all the fields required by `Circle` (this will be formalised in *Definition 32* later on).
- We **dynamically allocate structure instances**, by saving them in the **memory heap**; as a consequence, the creation of a structure just returns a **pointer** to a heap location, that can be used to access the structure fields. (The heap of typical RISC-V programs has been shown in *The RISC-V Memory Layout*.) These heap pointers appear in the program semantics, but they are *not* made directly accessible to Hygge programmers.
- To access a specific field of a structure instance, the semantics and code generation must use the structure pointer together with an **offset** which depends on the

³¹ <https://www.typescriptlang.org/docs/handbook/type-compatibility.html>

shape of the structure instance. Therefore, we will need to associate each structure pointer to some **information about the shape of the structure it points to**. We will keep this information in the runtime environment (for the interpreter) or in the structure types (for the compiled code).

This design implies that we also need some way to **deallocate unused structure instances from the heap** to support long-running programs that may create and discard large quantities of structures. We will mention this topic in the *References and Further Readings*, but we will not attempt the implementation of an actual garbage collector for *hygge*, since this would require a very substantial effort that is beyond the scope of this course.

Important: The extension described in this module is already implemented in the *upstream Git repository of hygge*: you should pull and merge the latest changes into your project compiler. The *Project Ideas* of this module further extend *Hygge* with support for other kinds structured data types.

7.2 Syntax

Definition 27 below extends the syntax of *Hygge* with structures and pointers applications, and with a new pretype that specifies the syntax of a new structure type.

Definition 27 (Syntax of Structures and Pointers)

We define the syntax of *Hygge0* with structures and pointers by extending *Definition 1* as follows.

Expression	$e ::= \dots$	
	$\text{struct } \{f_1 = e_1, \dots, f_n = e_n\}$	(Structure constructor, with $n \geq 1$)
	$e.f$	(Field selection)
Value	$v ::= \dots$	
	p	(Pointer (runtime value))
Pretype	$t ::= \dots$	
	$\text{struct } \{f_1 : t_1; \dots; f_n : t_n\}$	(Structure pretype, with $n \geq 1$)
Field	$f ::= z \mid \text{foo} \mid \text{a123} \mid \dots$	(Any non-reserved identifier)
Pointer	$p ::= 0x00000042 \mid 0x12abcdef \mid \dots$	(Memory address)

Definition 27 introduces the following syntactic elements:

- a **structure constructor** “ $\text{struct } \{f_1=e_1, \dots, f_n=e_n\}$ ” is used to instantiate a structure instance with n **fields** called f_1, \dots, f_n (with $i \geq 1$); each field f_i is

initialised by an expression e_i . Fields names have their own new syntactic category: they are syntactically similar to variables, in the sense that a field name can be any non-reserved identifier. Notice that the structure constructor is an expression, but it is *not* a value; in other words, structure constructors are not passed around directly (e.g. when calling a function), but they need to be reduced into values first (as we will see shortly);

- a **field selection** “ $e.f$ ” is used to access the field f of a target expression e (which, intuitively, is expected to be a structure instance);
- a **pointer** “ p ” denotes a memory address. We will see shortly that pointers are produced by structure constructors. Notice that pointers are **runtime values**: this means that they are part of the formal grammar of Hygge, but they are not supposed to be written directly by Hygge programmers, and should only be created during the execution of a program;
- a **structure pretype** “**struct** $\{f_1 : t_1, \dots, f_n : t_n\}$ ” describes the shape of a structure instance with n fields called f_1, \dots, f_n (with $i \geq 1$); each field f_i has pretype t_i .

7.3 Operational Semantics

Definition 28 formalises how substitution works for structure constructors, field selections, and pointers.

Definition 28 (Substitution for Structures Constructors and Pointers)

We extend *Definition 2* (substitution) with the following new cases:

$$\begin{aligned}(\text{struct } \{f_1 = e_1, \dots, f_n = e_n\}) [x \mapsto e'] &= \text{struct } \{f_1 = e_1 [x \mapsto e'], \dots, f_n = e_n [x \mapsto e']\} \\(e.f) [x \mapsto e'] &= (e [x \mapsto e']).f \\p [x \mapsto e'] &= p\end{aligned}$$

According to *Definition 28*, the substitution of variable x with expression e' works as follows:

- when performing the substitution on a structure constructor “**struct** $\{f_1=e_1, \dots, f_n=e_n\}$ ”, we simply propagate the substitution through each sub-expression e_i (for $i \in 1..n$);
- when performing the substitution on a field selection “ $e.f$ ”, we propagate the substitution through the target expression e ;
- finally, substitutions applied on a pointer p have no effect.

Definition 29 below formalises the semantics of structure constructors and field selections and assignments. To this purpose, we extend the runtime environment R with a simple model of a memory heap, as a mapping from memory addresses to values; we also

use R to save information on each known memory pointer p : this is needed to correctly access the contents of the memory block pointed by p .

Definition 29 (Semantics of Structures, Pointers, and the Heap)

We extend the definition of the **runtime environment** R in the *Structural Operational Semantics of Hygge0* by adding the following fields to the record R :

- R .Heap is a mapping from memory addresses to Hygge values, specifying which memory addresses are currently allocated in the heap, and what is the current value written in the corresponding memory location. Given a heap mapping h , we write $\text{maxAddr}(h)$ to retrieve the highest allocated address in h ; if h is empty, then $\text{maxAddr}(h)$ is $0x00000000$;
- R .PtrInfo is a mapping from memory addresses to lists of structure fields, specifying which structure fields are saved at the given address. We represent lists of structure fields as $[f_0; f_1; \dots; f_n]$.

Then, we define the **semantics of Hygge0 structures and pointers** by extending *Definition 4* to use the extended runtime environment R above, and by adding the following rules:

$$\begin{array}{c}
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{struct } \{f_1=v_1, \dots, f_{i-1}=v_{i-1}, f_i=e_i, \dots, f_n=e_n\} \rangle \rightarrow \langle R' \bullet \text{struct } \{f_1=v_1, \dots, f_{i-1}=v_{i-1}, f_i=e', \dots, f_n=e_n\} \rangle} \text{[R-Struct-F]} \\
 \\
 \frac{\begin{array}{l} h = R.\text{Heap} \quad h' = h + \{(p+i) \mapsto v_i\}_{i \in 0..n} \\ p = \text{maxAddr}(h) + 1 \quad i' = R.\text{PtrInfo} + (p \mapsto [f_0; \dots; f_n]) \end{array} \quad R' = \left\{ R \text{ with } \begin{array}{l} \text{Heap} = h' \\ \text{PtrInfo} = i' \end{array} \right\}}{\langle R \bullet \text{struct } \{f_0=v_0, \dots, f_n=v_n\} \rangle \rightarrow \langle R' \bullet p \rangle} \text{[R-Struct-Res]} \\
 \\
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e.f \rangle \rightarrow \langle R' \bullet e'.f \rangle} \text{[R-FieldSel-Eval]} \\
 \\
 \frac{R.\text{PtrInfo}(p) = [f_0; \dots; f_n] \quad \exists i \in 0..n : f = f_i \quad R.\text{Heap}(p+i) = v}{\langle R \bullet p.f \rangle \rightarrow \langle R' \bullet v \rangle} \text{[R-FieldSel-Res]} \\
 \\
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet e.f \leftarrow e_2 \rangle \rightarrow \langle R' \bullet e'.f \leftarrow e_2 \rangle} \text{[R-Assign-FieldSel-Target]} \\
 \\
 \frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet v.f \leftarrow e \rangle \rightarrow \langle R' \bullet v.f \leftarrow e' \rangle} \text{[R-Assign-FieldSel-Arg]} \\
 \\
 \frac{R.\text{PtrInfo}(p) = [f_0; \dots; f_n] \quad \exists i \in 0..n : f = f_i \quad R' = \{R \text{ with } \text{Heap} + ((p+i) \mapsto v)\}}{\langle R \bullet p.f \leftarrow v \rangle \rightarrow \langle R' \bullet v \rangle} \text{[R-Assign-FieldSel-Res]}
 \end{array}$$

The rules in *Definition 29* work as follows.

- By rule [R-Struct-F], we reduce a structure constructor “ $\text{struct } \{f_1=e_1, \dots, f_n=e_n\}$ ” by first reducing all its field initialisation expressions e_1, \dots, e_n , from left to right, until all of them become values.
- By rule [R-Struct-Res], a structure constructor “ $\text{struct } \{f_1=v_0, \dots, f_n=v_n\}$ ”

(where all fields are initialised by values) reduces by storing the structure data on the heap, and returning the memory address where such data is located. More in detail, the premises of the rule say that:

- h is the current heap, taken from the runtime environment R (i.e. $R.\text{Heap}$);
- p is the memory address of the first location after the maximum address currently used in h ; (e.g. if h assigns values to addresses between $0x00000001$ to $0x000000a8$, then p will be $0x000000a9$)
- h' is an updated heap that is equal to h , except that the memory locations in the range from $p + 0$ to $p + n$ are mapped to the values v_0, \dots, v_n that initialise the structure constructor fields. Notice that the memory addresses from $p + 0$ to $p + n$ were not used in the original heap h , but are now being used in h' ;
- i' is an updated pointer information mapping that is equal to $R.\text{PtrInfo}$, except that the pointer p is now mapped to the list of fields $[f_0; \dots; f_n]$ used in the structure constructor;
- R' is an updated runtime environment that is equal to R , except that $R'.\text{Heap}$ is h' and $R'.\text{PtrInfo}$ is i' , thus reflecting the fact that the structure instance is now allocated on the heap.

When all these premises hold, the reduction produces the updated runtime environment R' , together with the pointer p where the structure instance has been allocated. As a consequence, R' is “aware” of p , and it is possible to inspect and use the contents of the heap area pointed by p via $R'.\text{PtrInfo}$ and $R'.\text{Heap}$.

- By rule [R-FieldSel-Eval], we reduce a field selection expression “ $e.f$ ” by first reducing e , until it becomes a value;
- By rule [R-FieldSel-Res], a field selection expression “ $p.f$ ” (where p is a pointer value) expects that p is a known pointer in the runtime environment R , where a structure with a field called f is allocated. More in detail, the premises of the rule say that:
 - $R.\text{PtrInfo}$ maps the pointer p to a list of structure fields names $[f_0; \dots; f_n]$;
 - one of such fields names f_i (for some $i \in 1..n$) is equal to f (i.e. the field that is being accessed);
 - the heap $R.\text{Heap}$ maps the memory address $p + i$ to the value v .

When all these premises hold, the reduction produces an unchanged runtime environment R and the value v .

- Rule [R-Assign-FieldSel-Target] says that if we attempt to perform an assignment onto a structure field selected with “ $e.f$ ”, then we first need to reduce e , until it becomes a value.
- Rule [R-Assign-FieldSel-Arg] says that to perform an assignment “ $v.f \leftarrow e$ ” (where the target of the field selection is a value v), we first need to reduce the expression e (on the right-hand-side of the assignment) until it becomes a value.

- Rule [R-Assign-FieldSel-Res] says that an assignment “ $p.f \leftarrow v$ ” (where the field selection target is a pointer, and the right-hand-side of the assignment is a value) reduces by updating the structure data allocated on the heap. More in detail, the rule premises say that:
 - $R.PtrInfo$ maps the pointer p to a list of structure field names $[f_0; \dots; f_n]$;
 - one of such field names f_i (for some $i \in 1..n$) is equal to f (i.e. the field that is receiving the assignment);
 - R' is a runtime environment that is equal to R , except that in $R'.Heap$, the memory address $p + i$ (where the value of field f_i is stored) maps to the newly-assigned value v .

When all these premises hold, the reduction produces the updated runtime environment R' and the newly-assigned value v .

Example 43 (Reductions of Structure Construction and Assignment)

Consider the following expression, which creates a structure instance and updates one of its fields.

$$\text{let } s : t = \text{struct } \left\{ \begin{array}{l} \text{name} = \text{"Circle"}; \\ \text{area} = 42 \end{array} \right\};$$

$$s.\text{area} \leftarrow s.\text{area} + 2$$

Let us now consider the reductions of this expression, in a runtime environment R where:

- $R.Heap$ is empty, i.e. there is no allocated memory address on the heap;
- $R.PtrInfo$ is empty, i.e. we have no information about any known pointer.

In the first reduction below, the expression that initialises the “let...” expression is reduced, hence we perform the construction of a “struct” instance, via rule [R-Struct-Res] in [Definition 29](#). As a consequence, the reduction produces an updated runtime environment R' that is equal to R , except that:

- $R'.Heap$ is updated to make room for all the fields of the structure. In this case, we have two fields called *name* and *area*, carrying respectively a string and an integer, and each one of them is stored in a consecutive heap address. Consequently:
 - the heap address 0x0001 will point to the string “Circle”, and
 - the heap address 0x0002 will point to the integer 42;
- $R'.PtrInfo$ records the fact that the memory address 0x0001 is the beginning of a structure with two fields, called *name* and *area*.

The reduction yields the updated runtime environment R' and the memory address 0x0001 where the structure is saved.

$$\begin{array}{l}
 h = R.\text{Heap} = \emptyset \\
 \text{maxAddr}(h) + 1 = 0x0001
 \end{array}
 \quad
 \begin{array}{l}
 h' = \left\{ \begin{array}{l} 0x0001 \mapsto \text{"Circle"} \\ 0x0002 \mapsto 42 \end{array} \right\} \\
 i' = \{0x0001 \mapsto [\text{name}; \text{area}]\}
 \end{array}
 \quad
 R' = \left\{ R \text{ with } \begin{array}{l} \text{Heap} = h' \\ \text{PtrInfo} = i' \end{array} \right\}$$

$$\left\langle R \bullet \text{struct} \left\{ \begin{array}{l} \text{name} = \text{"Circle"}; \\ \text{area} = 42 \end{array} \right\} \right\rangle \rightarrow \langle R' \bullet 0x0001 \rangle$$

[R-Struct-Res]

$$\left\langle R \bullet \text{let } s : t = \text{struct} \left\{ \begin{array}{l} \text{name} = \text{"Circle"}; \\ \text{area} = 42 \end{array} \right\}; \right. \\
 \left. s.\text{area} \leftarrow s.\text{area} + 2 \right\rangle \rightarrow \left\langle R' \bullet \text{let } s : t = 0x0001; \right. \\
 \left. s.\text{area} \leftarrow s.\text{area} + 2 \right\rangle$$

[R-Let-Eval-Init]

In the second reduction below, a standard application of rule [R-Let-Subst] (from [Definition 4](#)) replaces each occurrence of the variable s with its initialisation value, i.e. the memory address $0x0001$ obtained in the previous reduction.

$$\left\langle R' \bullet \text{let } s : t = 0x0001; \right. \\
 \left. s.\text{area} \leftarrow s.\text{area} + 2 \right\rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 0x0001.\text{area} + 2 \rangle$$

[R-Let-Subst]

In the third reduction, the assignment “ $0x0001.\text{area} \leftarrow 0x0001.\text{area} + 2$ ” reduces the addition on its right-hand-side. This reduction, in turn, retrieves the value of the field access “ $0x0001.\text{area}$ ” from the runtime environment, via rule [R-FieldSel-Res] in [Definition 29](#). The premises of that rule inspect the runtime environment R' and find out that:

- according to $R'.\text{PtrInfo}$, the address $0x0001$ (on which a field selection is being attempted) points to a structure with a list of two fields, called name and area . We index the field names according to their position on the list, hence the field area has index 1;
- therefore, to access “ $0x0001.\text{area}$ ” we need to read the heap at the address $0x0002$ (i.e. the address $0x0001$ plus the offset 1 for the field area);
- the location $R'.\text{Heap}(0x0002)$ contains the value 42, which becomes the result of the reduction of “ $0x0001.\text{area}$ ”.

$$\begin{array}{l}
 R'.\text{PtrInfo}(0x0001) = [f_0; f_1] \\
 \text{where } f_0 = \text{name} \text{ and } f_1 = \text{area}
 \end{array}
 \quad
 R'.\text{Heap}(0x0002) = 42$$

$$\frac{\langle R' \bullet 0x0001.\text{area} \rangle \rightarrow \langle R' \bullet 42 \rangle}{\langle R' \bullet 0x0001.\text{area} + 2 \rangle \rightarrow \langle R' \bullet 42 + 2 \rangle}$$

[R-FieldSel-Res]

[R-Add-L]

$$\frac{\langle R' \bullet 0x0001.\text{area} \leftarrow 0x0001.\text{area} + 2 \rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 42 + 2 \rangle}{\langle R' \bullet 0x0001.\text{area} \leftarrow 0x0001.\text{area} + 2 \rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 42 + 2 \rangle}$$

[R-Assign-FieldSel-Arg]

The fourth reduction computes the addition $42 + 2$.

$$\frac{\langle R' \bullet 42 + 2 \rangle \rightarrow \langle R' \bullet 44 \rangle}{\langle R' \bullet 0x0001.\text{area} \leftarrow 42 + 2 \rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 44 \rangle}$$

[R-Add-Res]

$$\frac{\langle R' \bullet 0x0001.\text{area} \leftarrow 42 + 2 \rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 44 \rangle}{\langle R' \bullet 0x0001.\text{area} \leftarrow 42 + 2 \rangle \rightarrow \langle R' \bullet 0x0001.\text{area} \leftarrow 44 \rangle}$$

[R-Assign-FieldSel-Arg]

The fifth reduction performs the assignment of value 44 to the field selection “ $0x0001.\text{area}$ ”, via rule [R-Assign-FieldSel-Res] in [Definition 29](#). The rule premises inspect the runtime environment R' similarly to rule [R-FieldSel-Res] in the third reduction step above:

- the premises determine that “0x0001.*area*” is located at the heap address 0x0002, and
- they produce an updated runtime environment R'' that is equal to R' , except that in R'' .Heap the address 0x0002 points to the newly-assigned value 44.

$$\begin{array}{c}
 R'.\text{PtrInfo}(0x0001) = [f_0; f_1] \\
 \text{where } f_0 = \textit{name} \text{ and } f_1 = \textit{area} \quad R'' = \{R' \text{ with Heap} + (0x0002 \mapsto 44)\} \\
 \hline
 \langle R' \bullet 0x0001.\textit{area} \leftarrow 44 \rangle \rightarrow \langle R'' \bullet 44 \rangle \quad \text{[R-Assign-FieldSel-Res]}
 \end{array}$$

Therefore, the expression given at the beginning of this example terminates its reductions by reaching the value 44. Notice, however, that the runtime environment R'' is different from the initial runtime environment R , because we have:

$$R''.\text{PtrInfo} = \{0x0001 \mapsto [\textit{name}; \textit{area}]\} \quad \text{and} \quad R''.\text{Heap} = \left\{ \begin{array}{l} 0x0001 \mapsto \textit{Circle} \\ 0x0002 \mapsto 44 \end{array} \right\}$$

Exercise 32

Write the reductions of the following expression:

$$\begin{array}{l}
 \text{let } s : t = \text{struct} \left\{ \begin{array}{l} \textit{name} = \textit{Circle}; \\ \textit{area} = 40 + 2 \end{array} \right\}; \\
 \text{let } s2 : t = s; \\
 s2.\textit{area} \leftarrow s2.\textit{area} * 2
 \end{array}$$

7.4 Typing Rules

In order to type-check programs that use structures, we need to introduce a new structure type ([Definition 30](#)), a new rule for pretype resolution ([Definition 31](#)), a new subtyping rule ([Definition 32](#)), and some new typing rules ([Definition 33](#)).

Definition 30 (Structure Type)

We extend the Hygge0 typing system with a **structure type** by adding the following case to [Definition 5](#):

$$\begin{array}{l}
 \text{Type } T ::= \dots \\
 \quad | \quad \text{struct} \{f_1 : T_1; \dots; f_n : T_n\} \quad \left(\begin{array}{l} \text{Structure type, with } n \geq 1 \\ \text{and } f_1, \dots, f_n \text{ pairwise distinct} \end{array} \right)
 \end{array}$$

By [Definition 30](#), a structure type describes a structure instance where each field f_i (for $i \in 1..n$, with $n \geq 1$) has a type T_i . Note that the field names must be distinct from each other.

Example 44 (Structure Types)

The following type describes a structure with a field f of type `int`, and a field g of type `bool`.

$$\text{struct } \{f : \text{int}; g : \text{bool}\}$$

The following type describes a structure with a field f of type `int`, and a field g having a structure type, which in turn has a field a of type `float` and a field b of type `bool`.

$$\text{struct } \{f : \text{int}; g : \text{struct } \{a : \text{float}; b : \text{bool}\}\}$$

We also need a way to resolve a syntactic structure pretype (from [Definition 27](#)) into a valid structure type (from [Definition 30](#)): this is formalised in [Definition 31](#) below.

Definition 31 (Resolution of Structure Types)

We extend [Definition 7](#) (type resolution judgement) with this new rule:

$$\frac{f_1, \dots, f_n \text{ pairwise distinct} \quad \forall i \in 1..n : \Gamma \vdash t_i \triangleright T_i}{\Gamma \vdash \text{struct } \{f_1 : t_1; \dots f_n : t_n\} \triangleright \text{struct } \{f_1 : T_1; \dots, f_n : T_n\}} \text{ [TRes-Struct]}$$

According to [Definition 31](#), a function pretype is resolved by ensuring that field names are distinct from each other, and then by recursively resolving the type of each field.

We also extend [Definition 10](#) (subtyping) with a new rule for structure types, according to [Definition 32](#) below. This extension is not strictly necessary, but it adds great flexibility to the Hygge programming language, as we will see shortly. Notice that, without [Definition 32](#), the subtyping for structure types would only be allowed by rule [TSub-Refl] in [Definition 10](#), which only relates types that are exactly equal to each other.

Definition 32 (Subtyping for Structure Types)

We define the subtyping of Hygge0 with structure types by extending [Definition 10](#) with the following new rule:

$$\frac{m \geq n \quad \forall i \in 1..n : \Gamma \vdash T_i \leq T'_i}{\Gamma \vdash \text{struct } \{f_1 : T_1; \dots, f_m : T_m\} \leq \text{struct } \{f_1 : T'_1; \dots, f_n : T'_n\}} \text{ [TSub-Struct]}$$

According to rule [TSub-Struct] in [Definition 32](#), a structure type “ $\text{struct } \{f_1 : T_1, \dots, f_m : T_m\}$ ” is subtype of another structure type “ $\text{struct } \{f_1 : T'_1, \dots, f_n : T'_n\}$ ” when:

- the subtype contains *at least* all the structure fields that appear in the supertype, and such fields appear first and in the same order (this is enforced by the shape of the structure types and by the rule premise “ $m \geq n$ ”); and

- if a field f_i appears in the structure supertype with type T'_i , then T'_i is a supertype of the corresponding field type T_i in the structure subtype.

The rationale for rule [TSub-Struct] is based on *Definition 9* (Liskov Substitution Principle): an instance of a subtype should be safely usable whenever an instance of the supertype is required. See *Example 45* below.

Example 45

Consider the following structure type:

```
struct {f : int; g : bool}
```

Consider any well-typed Hygge program that uses an instance of such a structure type: intuitively, that program will only access the field f (using it as an int) and the field g (using it as a bool).

Therefore, that Hygge program will also work correctly if it operates on a structure instance of the following type:

```
struct {f : int; g : bool; h : string}
```

The reason is that the program will still be able to access the structure fields f and g , since they appear in the expected order and have the expected types; the program will simply ignore the additional field h . For this reason, *Definition 32* considers the second structure type as a subtype of the first.

In some sense, this is similar to **inheritance in object-oriented programming languages**: the second structure type above “derives” the first structure type (because it “implements” all its fields), hence the second structure type is a “subclass” of the first.

This idea is also visible in *Example 42*, where the function `displayShape` is called with arguments that have all the fields expected by the structure type `Shape`, plus other fields (that `displayShape` does not use). This is allowed because all the arguments passed to `displayShape` have a subtype of `Shape`, according to *Definition 32*.

We now have all the ingredients to define the typing rules for structure constructors and field accesses and assignments: they are formalised in *Definition 33* below.

Definition 33 (Typing Rules for Structures and Field Access and Assignment)

We define the typing rules of Hygge0 with structures and field assignments by extending *Definition 11* with the following rules (which use the structure type introduced

in *Definition 30* above):

$$\frac{f_1, \dots, f_n \text{ pairwise distinct} \quad \forall i \in 1..n : \Gamma \vdash e_i : T_i}{\Gamma \vdash \text{struct} \{f_1 = e_1, \dots, f_n = e_n\} : \text{struct} \{f_1 : T_1, \dots, f_n : T_n\}} \text{ [T-Struct]}$$

$$\frac{\Gamma \vdash e : \text{struct} \{f_1 : T_1, \dots, f_n : T_n\} \quad \exists i \in 1..n : f = f_i}{\Gamma \vdash e.f : T_i} \text{ [T-FieldSel]}$$

$$\frac{\Gamma \vdash e_1.f : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1.f \leftarrow e_2 : T} \text{ [T-Assign-FieldSel]}$$

The typing rules in *Definition 33* work as follows.

- By rule [T-Struct], a structure constructor “ $\text{struct} \{f_1=e_1, \dots, f_n=e_n\}$ ” has a structure type “ $\text{struct} \{f_1 : T_1, \dots, f_n : T_n\}$ ”, where (according to the rule premises) each field f_i has the type of the corresponding initialisation expression e_i . The first premise of the rule also requires that all field names f_i are distinct from each other.
- By rule [T-FieldSel], a field selection “ $e.f$ ” is well-typed if (according to the rule premises) e has a structure type, and there is a field f_i of that structure type which matches f (i.e. the field being accessed). If these premises hold, then the whole field access has type T_i (i.e. the type of the structure field f_i).
- By rule [T-Assign-FieldSel], an assignment to a field selection “ $e_1.f \leftarrow e_2$ ” is type-checked by ensuring that the field selection “ $e_1.f$ ” has a type T (first premise of the rule), and that such a type T is also the type of the assigned expression e_2 (second premise of the rule). If these premises hold, then the whole assignment has type T .

Example 46

Consider the following expression, which initialises variable s with a structure instance, and updates the value of the field $s.area$:

$$\begin{aligned} &\text{let } s : \text{struct} \{area : \text{int}\} = \\ &\quad \text{struct} \left\{ \begin{array}{l} area = 42; \\ name = \text{“Circle”} \end{array} \right\}; \\ &\quad s.area \leftarrow s.area + 2 \end{aligned}$$

We have the following typing derivation — where the typing environment Γ' is equal to Γ , except that we have $\Gamma'.\text{Vars}(s) = \text{struct} \{area : \text{int}\}$.

$$\frac{\frac{\frac{\Gamma \vdash \text{“int”} > \text{int}}{\Gamma \vdash \text{“struct} \{area : \text{int}\} > \text{struct} \{area : \text{int}\}} \text{ [TRes-Int]} \quad \frac{\frac{\Gamma \vdash 42 : \text{int}}{\Gamma \vdash \text{“42”} : \text{string}} \text{ [T-Int]} \quad \frac{\Gamma \vdash \text{“Circle”} : \text{string}}{\Gamma \vdash \text{“Circle”} : \text{string}} \text{ [T-String]} \quad \frac{\Gamma \vdash \text{“42”} : \text{string} \quad \Gamma \vdash \text{“Circle”} : \text{string}}{\Gamma \vdash \text{struct} \{area = 42; name = \text{“Circle”}\} : \text{struct} \{area : \text{int}; name : \text{string}\}} \text{ [T-Struct]}}{\Gamma \vdash \text{struct} \{area = 42; name = \text{“Circle”}\} : \text{struct} \{area : \text{int}; name : \text{string}\}} \text{ [T-Sub-Ref]} \quad \frac{\Gamma \vdash \text{int} \leq \text{int}}{\Gamma \vdash \text{struct} \{area : \text{int}; name : \text{string}\} \leq \text{struct} \{area : \text{int}\}} \text{ [TSub-Struct]} \quad \frac{\Gamma \vdash \text{int} \leq \text{int}}{\Gamma \vdash \text{struct} \{area : \text{int}; name : \text{string}\} \leq \text{struct} \{area : \text{int}\}} \text{ [T-Sub]} \quad \frac{\Gamma'(s) = \text{struct} \{area : \text{int}\}}{\Gamma' \vdash s : \text{struct} \{area : \text{int}\}} \text{ [T-Var]} \quad \frac{\Gamma'(s) = \text{struct} \{area : \text{int}\}}{\Gamma' \vdash s : \text{struct} \{area : \text{int}\}} \text{ [T-FieldSel]} \quad \frac{\Gamma' \vdash s : \text{struct} \{area : \text{int}\} \quad \Gamma' \vdash 2 : \text{int}}{\Gamma' \vdash s.area : \text{int}} \text{ [T-FieldSel]} \quad \frac{\Gamma' \vdash s.area : \text{int} \quad \Gamma' \vdash 2 : \text{int}}{\Gamma' \vdash s.area + 2 : \text{int}} \text{ [T-Add]} \quad \frac{\Gamma' \vdash s.area + 2 : \text{int}}{\Gamma' \vdash [s.area \leftarrow s.area + 2] : \text{int}} \text{ [T-Assign-FieldSel]} \quad \frac{\Gamma \vdash \text{struct} \{area : \text{int}\} > \text{struct} \{area : \text{int}\} \quad \Gamma \vdash \text{struct} \{area = 42; name = \text{“Circle”}\} : \text{struct} \{area : \text{int}; name : \text{string}\} \quad \Gamma' \vdash [s.area \leftarrow s.area + 2] : \text{int}}{\Gamma \vdash \text{let } s : \text{struct} \{area : \text{int}\} = \text{struct} \{area = 42; name = \text{“Circle”}\}; s.area \leftarrow s.area + 2 : \text{int}} \text{ [T-Let]}$$

Notice that the “let...” binder declares a variable s with type “ $\text{struct} \{area : \text{int}\}$ ”, but initialises s with a structure constructor that has an additional field “ $name$ ”: to satisfy

the premises of rule [T-Let], we need to show that such a structure constructor also has type “struct {area : int}” (i.e. without the additional field “name”). To this end, the derivation uses the subsumption rule [T-Sub] (from *Definition 11*) and the new subtyping rule [TSub-Struct] (from *Definition 32*); without rule [TSub-Struct], this program would not type-check.

7.5 Implementation

We now have a look at how `hyggec` is extended to implement structures and field accesses and assignments, according to the specification illustrated in the previous sections.

Tip: To see a summary of the changes described below, you can inspect the differences in the *hyggec Git repository* between the tags `functions` and `structures`.

7.5.1 Lexer, Parser, Interpreter, and Type Checking

These parts of the `hyggec` compiler are extended along the lines of *Example: Extending Hygge0 and hyggec with a Subtraction Operator*, except that, in order to implement *Definition 32*, we need to extend the function `isSubtypeOf`.

- We extend `AST.fs` in two ways, according to *Definition 27*:
 - we extend the data type `Expr<'E, 'T>` with three new cases:
 - * `Struct` for the structure constructor “struct { $f_1 = t_1, \dots, f_n = e_n$ }”;
 - * `FieldSelect` for “ $e.f$ ”; and
 - * `Pointer` for “ p ”;
 - we also extend the data type `Pretype` with a new case called `TStruct`, corresponding to the new structure pretype “struct { $f_1 : t_1, \dots, f_n : t_n$ }”:

```
and Pretype =
// ...
/// A structure pretype, with pretypes for each field.
| TStruct of fields: List<string * PretypeName>
```

- We extend `PrettyPrinter.fs` to support the new expressions and pretype.
- We extend `Lexer.fsl` to support two new tokens:
 - `STRUCT` for the new keyword “struct”, and
 - `DOT` for the symbol “.” used to access structure fields in the expression “ $e.f$ ”.
- We extend `Parser.fsy` to recognise the desired sequences of tokens according to *Definition 27*, and generate AST nodes for the new expressions. We proceed by adding:

- a new rule under the pretype category to generate TStruct pretype instances;
- two new rules under the primary category to generate Struct and FieldSelect instances;
- various auxiliary syntactic categories and rules to recognise the syntactic elements needed by the rules above. In particular:
 - * field matches a structure field;
 - * fieldInitSeq is a non-empty sequence of field assignments, separated by semicolons (needed to parse structure constructors);
 - * fieldTypeSeq is a sequence of fields with type annotations, separated by semicolons (needed to parse structure pretypes).

Note: We do *not* extend Parser.fs with new rules for parsing Pointer instances. The reason is that, as we mentioned when introducing the *syntax of structures*, we want to treat pointers as **runtime values** that are only produced by the semantics (while a program reduces) and cannot be written by Hygge programmers.

- We extend the function subst in ASTUtil.fs to support the new expressions Struct, FieldSelect and Pointer, according to *Definition 28*.
- We extend Interpreter.fs according to *Definition 29*:
 - we extend the definition of the record RuntimeEnv to include the new fields Heap and PtrInfo;
 - in the function isValue, we add a new case for Pointer; and
 - in the function reduce:
 - * we add a new case for Pointer (which does not reduce);
 - * we add a new case for Struct, with an auxiliary function called heapAlloc that helps allocating new values on top of the heap, as required by rule [R-Struct-Res];
 - * we add a new case for Fieldselect; and
 - * we add a new case to reduce an Assign expression when the left-hand-side of the assignment is a field selection, according rules [R-Assign-FieldSel-Target] and [R-Assign-FieldSel-Arg].
- We extend Type.fs by adding a new case to the data type Type, according to *Definition 30*: the new case is called TStruct. We also add a corresponding new case to the function freeTypeVars in the same file.

Note: Correspondingly, we also extend the pretty-printing function formatType in PrettyPrinter.fs, to display the stucture type we have just introduced.

- We extend `Typechecker.fs`:
 - we extend the type resolution function `resolvePretype` with a new case for structure types, according to *Definition 31*;
 - we extend the function `isSubtypeOf` with a new case for structure types, according to *Definition 32*;
 - we extend the function `typer` according to *Definition 33*, to support the new cases for the expressions `Struct` and `FieldSelect`, and type-check assignments to structure fields.
- As usual, we add new tests for all compiler phases.

7.5.2 Code Generation

We extend `RISCVCodegen.fs` in three ways:

- *Code Generation for Structure Constructors*
- *Code Generation for Field Selection*
- *Code Generation for Assignments to Structure Fields*

Code Generation for Structure Constructors

When compiling a structure constructor, we need to add a new case to the function `doCodegen` in `RISCVCodegen.fs` matching the expression `Struct`. We need to generate RISC-V assembly code that:

1. allocates enough space on the heap to contain all the structure fields, and
2. initialises each structure fields with the value produced by the corresponding field initialisation expression.

To allocate space on the heap, we use the RARS system call `Sbrk`: it extends the heap by allocating the amount of bytes specified in the register `a0`, and returns the address of the newly-allocated memory block in the register `a0` itself. In order to compute the required amount of bytes, we need to know the the size of the structure instance – which is quite straightforward: each type of Hygge value, once compiled, fits in a single 32-bit word, and therefore, we obtain the structure size by multiplying the number of structure fields by 4.

The resulting code looks as follows

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | Struct(fields) ->
    /// Assembly code for initialising each field of the struct
    let fieldsInitCode = // ...

    /// Assembly code that allocates space on the heap for the new
```

(continues on next page)

(continued from previous page)

```

// structure, through an 'Sbrk' system call. The size of the structure
// is computed by multiplying the number of fields by the word size (4)
let structAllocCode =
  (beforeSysCall [Reg.a0] [])
    .AddText([
      (RV.LI(Reg.a0, fields.Length * 4),
        "Amount of memory to allocate for a struct (in bytes)")
      (RV.LI(Reg.a7, 9), "RARS syscall: Sbrk")
      (RV.ECALL, "")
      (RV.MV(Reg.r(env.Target), Reg.a0),
        "Move syscall result (struct mem address) to target")
    ])
  ++ (afterSysCall [Reg.a0] [])

// Put everything together: allocate heap space, init all struct fields
structAllocCode ++ fieldsInitCode

```

Code Generation for Field Selection

The assembly code for field selections needs to establish what is the memory location of the selected structure field. To this end:

1. we compile the target expression of the field selection, which is expected to leave a memory address in the target register; and then
2. use the target expression type (which should be TStruct) to find out the offset of the selected field from that memory address.

We then use the RISC-V instruction `lw` (load word) to read a value from the memory location of the structure field, and write it in the target register.

```

let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | FieldSelect(target, field) ->
    /// Generated code for the target object whose field is being selected
    let selTargetCode = doCodegen env target
    /// Assembly code to access the struct field in memory
    let fieldAccessCode =
      match (expandType node.Env target.Type) with
      | TStruct(fields) ->
        let (fieldNames, fieldTypes) = List.unzip fields
        let offset = List.findIndex (fun f -> f = field) fieldNames
        match fieldTypes.[offset] with
        // ...
        | _ ->
          Asm(RV.LW(Reg.r(env.Target), Imm12(offset * 4),
            Reg.r(env.Target)),
            $"Retrieve value of struct field '%s{field}'")

```

(continues on next page)

(continued from previous page)

```
// Put everything together: compile the target, access the field
selTargetCode ++ fieldAccessCode
```

Code Generation for Assignments to Structure Fields

To support assignments to structure fields, we extend the code generation for the Assign expression. The new case generates RISC-V assembly code to:

- compute the target expression of the field selection (which should produce the memory address of a structure instance on the heap);
- compute the assigned value, and
- compute the memory address of the structure field that is receiving the assignment (as in the *Code Generation for Field Selection*).

Then, the generated code overwrites that memory address with the value being assigned.

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  // ...
  | Assign(lhs, rhs) ->
    match lhs.Expr with
    // ...
    | FieldSelect(target, field) ->
      let selTargetCode = // ...Code for target expression of selection
      let rhsCode = // ...Code for the right-hand-side of the assignment

      match target.Type with
      | TStruct(fields) ->
        let (fieldNames, _) = List.unzip fields // Struct field names
        // Offset of the selected struct field beginning of struct
        let offset = List.findIndex (fun f -> f = field) fieldNames
        let assignCode = // Assembly code to perform field assignment
        match rhs.Type with
        // ...
        | _ ->
          Asm([RV.SW(Reg.r(env.Target + 1u), Imm12(offset * 4),
            Reg.r(env.Target)),
            $"Assigning value to struct field '%s{field}''"),
            (RV.MV(Reg.r(env.Target), Reg.r(env.Target + 1u)),
            "Copying assigned value to target register")])

        // Put everything together
        rhsCode ++ selTargetCode ++ assignCode
```

7.6 References and Further Readings

The treatment of structure types in Hygge is inspired by languages with structural typing systems, such as F#, OCaml, Scala, TypeScript. To know more about the programming language concepts behind this, you can refer to:

- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002. [Available on DTU Findit³²](#).
 - Chapter 19 (Case Study: Featherweight Java) – in particular, section 19.3 (Nominal and Structural Type Systems)

Although this version of Hygge allocates structures on the heap, it does not provide any facility to deallocate them. Therefore, the heap usage of a program will tend to grow until the program terminates – and it is possible to write programs that allocate too many structures and are killed by the operating system (or by RARS) with an out-of-memory error. This issue can be addressed in two ways.

1. We could add a C-style expression like `free(e)` which removes from the heap the contents of `e` (which is supposed to reduce into a structure pointer). This is not entirely straightforward to implement – but most importantly, manual memory deallocation would introduce the typical problems of C programs, such as missing deallocations causing memory leaks, or duplicate deallocations that crash a program.
2. We could tweak the semantics of Hygge with the automatic removal of structures that are not pointed from anywhere within the program, nor the heap. Correspondingly, we could extend the `hyggec` code generation to include the assembly code of a **garbage collector** that is informed every time a new structure is allocated, and deallocates that structure when it becomes unreferenced. Writing (or finding) a garbage collector that can be compiled to run under RARS is not a trivial task. The [Boehm-Demers-Weiser garbage collector³³](#) is a popular option that can be integrated into C/C++ programs, and could also be used in the runtime system of a compiled programming language.

7.7 Project Ideas

For your group project, you should implement at least 3 of the following project ideas (listed in order of difficulty):

- *Project Idea: Extend Hygge with Reference Cells*
- *Project Idea: Extend Hygge with Tuples*
- *Project Idea: Mutable vs. Immutable Structure Fields*
- *Project Idea: Extend Hygge with Arrays*
- *Optional Challenge: Extend Hygge with Copying of Structures*

³² <https://findit.dtu.dk/en/catalog/5c34980ed9001d2f3820637f>

³³ <https://github.com/ivmai/bdwgc>

The last project idea includes an **optional challenge**: if you wish to take this challenge instead of some other project idea, please speak with the teacher.

Important: If, in the previous module, you chose *Project Idea: Function Subtyping*, please remember to add some test cases to show how function subtyping interplays with structure subtyping.

7.7.1 Project Idea: Extend Hygge with Reference Cells

The goal of this project idea is to extend Hygge and `hyggec` with a construct similar to [reference cells in the F# programming language](#)³⁴. Hygge programmers should be able to write, interpret, type-check, compile and run a program like the following:

```
let x: ref {int} = ref {40 + 2};
x.value <- x.value + 10;
assert(x.value = 52)
```

where `ref {int}` is the type of a reference to a value of type `int`, while the reference constructor `ref {40 + 2}` represents a reference to a location on the heap, where the result of the expression `40 + 2` is saved.

You should describe how you modify the Hygge language specification and the `hyggec` implementation to achieve this extension. As usual, you should also provide tests that leverage this extension.

Hint: You can approach this project idea by using F# reference cells as a blueprint:

- the type `ref {int}` is just syntactic sugar for a structure type `struct {value: int}`, and
 - the reference constructor `ref {42}` is just syntactic sugar for a structure constructor `struct {value = 42}`.
-

7.7.2 Project Idea: Extend Hygge with Tuples

The goal of this project idea is to extend Hygge and `hyggec` with tuples, i.e. sequences of elements of a fixed length determined *statically* (*not* at run-time), and with each element having its own predetermined type. Tuples can have any length (minimum one element). Hygge programmers should be able to write, interpret, type-check, compile and run a program like the following:

³⁴ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/reference-cells>

```
let t: tuple {string, int, bool} = tuple {"Hello", 40 + 2, true};
t._1 <- "Hej";
t._2 <- t._2 + 10;
assert(t._2 = 52)
```

where:

- `tuple {string, int, bool}` is the type of a tuple containing 3 values, such that the first has type `string`, the second has type `int`, and the third has type `bool`;
- `tuple {"Hello", 40 + 2, true}` is the constructor of a tuple containing 3 values, initialised as "Hello", 40 + 2, and true;
- the selectors `_1`, `_2`, etc. are used to access and assign a value to the corresponding element of the tuple. The tuple length determines which selectors are available: for instance, a tuple of length 12 has selectors ranging from `_1` to `_12`. (This is inspired by [tuples in the Scala 2 programming language](#)³⁵.)

You should describe how you modify the Hygge language specification and the `hyggec` implementation to achieve this extension. As usual, you should also provide tests that leverage this extension.

Hint: The hint given for the [project idea on reference cells](#) might be helpful...

7.7.3 Project Idea: Mutable vs. Immutable Structure Fields

The goal of this project idea is to allow Hygge and `hyggec` to distinguish between mutable and immutable structure fields. You can choose between two possible alternative designs:

1. all structure fields are mutable by default (as specified in this module) — but a programmer can optionally specify that some fields are immutable, e.g. by writing a structure type like: `struct { f: int; immutable g: string}`. Or,
2. all structure fields are *immutable* by default, but a programmer can optionally specify that some fields are mutable, e.g. by writing a structure type like: `struct { f: int; mutable g: string}`. This approach is similar to F#, but it will require the revision of several existing `hyggec` tests.

You should describe how you modify the Hygge language specification and the `hyggec` implementation to achieve this extension. As usual, you should also provide tests that leverage this extension.

Hint:

- You don't need to substantially change the Hygge semantics with respect to [Definition 28](#): since mutable/immutable structure fields are only specified in structure types, you can keep treating all fields as mutable in the interpreter.

³⁵ <https://docs.scala-lang.org/tour/tuples.html>

- You *do* need to revise the types and type checking:
 - you need to remember which fields of a structure type are mutable, and which fields are immutable;
 - if a program attempts to perform an assignment “ $e.f \leftarrow e'$ ” where f is an immutable field of the structure type of e , then the assignment should not type-check, and a corresponding error should be reported.
- Structure subtyping (*Definition 32*) should be extended to distinguish mutable and immutable fields. For maximum flexibility, you can extend subtyping as follows:
 - if a structure field f is immutable in the supertype, then f can be either mutable or immutable in the subtype;
 - if a structure field f is mutable in the supertype, then f must also be mutable in the subtype.

7.7.4 Project Idea: Extend Hygge with Arrays

The goal of this project idea is to extend Hygge and `hyggec` with arrays, i.e. sequences of values of a same type, with a fixed length determined at run-time (unlike tuples). Arrays can have any length (minimum one element). Hygge programmers should be able to write, interpret, type-check, compile and run a program like the following (but feel free to change the syntax as you like):

```

let n: int = readInt();
let arr: array {int} = array(n, 40 + 2);

let mutable i: int = 0;
while (i < arrayLength(arr)) do {
  arrayElem(arr, i) <- arrayElem(arr, i) + i;
  i <- i + 1
};

i <- 0;
while (i < arrayLength(arr)) do {
  assert(arrayElem(arr, i) = 42 + i)
  i <- i + 1
}

```

where:

- `array {int}` is the type of an array containing elements of type `int`;
- `array(n, 40 + 2)` is the constructor of an array containing n values, each one initialised with the result of the expression `40 + 2`;
- `arrayLength(arr)` is the size (number of elements) of the array `arr`;
- `arrayElem(arr, i)` is used to access and assign a value to element i of array `arr` (with elements numbered from 0). The program should get stuck if the index i is

not smaller than the size of the array.

You should describe how you modify the Hygge language specification and the hyggec implementation to achieve this extension. As usual, you should also provide tests that leverage this extension.

Hint: Intuitively, an array instance could be imagined as a structure with two fields:

- a field `length` with the size of the array; and
- a field `data` with a pointer to the memory location where the array elements are stored.

However, arrays are *not* just syntactic sugar for structures! The main difference is that the array size is dynamic, so the `data` field in the intuition above needs a dedicated treatment. Still, you can certainly reuse and adapt part of the specification and code for structures to design and implement arrays...

7.7.5 Optional Challenge: Extend Hygge with Copying of Structures

In the current specification of Hygge, and in hyggec, structures are always handled by reference. For example, consider the following program:

```
let s1: struct {f: int} = struct {f = 0};
let s2: struct {f: int} = s1;
s1.f <- 42;
assert(s1.f = 42);
assert(s2.f = 42)
```

The “let...” binder that introduces `s2` initialises it with the same heap address of the structure `s1`; as a consequence, any change to the fields of `s1` is reflected in `s2` — and *vice versa*.

The goal of this project idea is to extend Hygge and hyggec with an expression `copy(e)`, which takes an expression `e` (expected to be a structure) and returns a copy of `e` — i.e. a *new* structure that has the same type, fields, and values of `e`, but does *not* share any data with `e`.

As a consequence, Hygge programmers should be able to write, interpret, type-check, compile and run a program like the following:

```
let s1: struct {f: int} = struct {f = 0};
let s2: struct {f: int} = copy(s1);
s1.f <- 42;
assert(s1.f = 42);
assert(s2.f = 0)
```

You should describe how you modify the Hygge language specification and the hyggec implementation to achieve this extension. As usual, you should also provide tests that leverage this extension.

Note: The formal specification of the semantics of the `copy(e)` expression can be quite complicated, so you can omit it.

You can approach this project idea in two steps.

1. Extend Hygge and `hyggec` with **shallow copying**, i.e. copy the contents of structure fields, *without* recursively copying the structures pointed by those fields (if any). For example, shallow copying would allow to correctly execute both the example above, and the following program:

```
let s1: struct {f: struct {g: float}} = struct {f = struct {g = 0.0f}};
let s2: struct {f: struct {g: float}} = copy(s1);
s1.f.g <- 1.0f;
assert(s1.f.g = 1.0f);
assert(s2.f.g = 1.0f)
```

This is because shallow copying only copies the “first level” of the structure, and thus copies the structure pointer of `s1.f` into `s2.f`. Consequently, `s1.f` and `s2.f` point to the same structure, and any change to `s1.f.g` is visible in `s2.f.g` — and *vice versa*.

2. **Optional challenge:** implement **deep copying**, i.e. copy the contents of structure fields, and recursively copy any structure pointed by those fields. For example, deep copying would allow to correctly execute the following program:

```
let s1: struct {f: struct {g: float}} = struct {f = struct {g = 0.0f}};
let s2: struct {f: struct {g: float}} = copy(s1);
s1.f.g <- 1.0f;
assert(s1.f.g = 1.0f);
assert(s2.f.g = 0.0f)
```

This is because deep copying duplicates the structure pointed by `s1.f`, and initialises `s2.f` with a pointer to the structure copy. Consequently, `s1.f` and `s2.f` point to different structures that do not share any data, and any change to `s1.f.g` is not reflected in `s2.f.g` — and *vice versa*.

Hint: To understand how many fields of a structure should be copied by `copy(e)` (in a shallow or deep way), you can inspect either the runtime environment (when interpreting the program) or the structure type of `e` (during code generation).

8

Module 8: Lab Day

This module does not introduce new contents: **on 23 March, from 8:00 to 12:00, you can work on your project or past exercises. The teacher and TA will be present in the classroom, and you can request help or ask them questions about your project, or course topics, or technical issues.**

We can arrange **mini-sessions during the Lab Day** to address specific questions and topics requested by two or more students. To propose a question/topic for these mini-sessions, please use the **poll on the course website on DTU Learn**, under “Contents” → “Module 8”.

Module 9: Closures

In this module we study and address the *limitations* of the specification and implementation of functions presented in *Module 6: Functions and the RISC-V Calling Convention*. The key issue is that we do not (yet) have a proper treatment of **closures**, i.e. functions that capture (“close over”) variables which are defined (“bound”) in their surrounding scope. The solution to this issue is based on code rewriting in combination with *structured data types*.

9.1 Overall Objective

Our goal is to correctly interpret, compile and run Hygge programs like the one shown in *Example 47* and *Example 48* below.

Example 47 (A Function that Captures an Immutable Variable)

Consider the following program:

```
1 // Take x, return a function that adds x to its argument
2 fun makeAdder(x: int): (int) -> int =
3     fun (y: int) ->
4         x + y; // x is captured from the surrounding scope
5
6 let add1: (int) -> int = makeAdder(1);
7 let add2: (int) -> int = makeAdder(2);
8
9 // These assertions succeed in the interpreter, but fail in code generation!
10 assert(add1(40) = 41);
11 assert(add2(40) = 42)
```

Here the issue is that when we call `makeAdder`, the argument `x` is passed via register `a0`; however, `a0` is also assigned to the argument `y` of the function returned by `makeAdder`. Therefore, the compiled function ends up computing `y + y` (which is incorrect).

Example 48 (A Function that Captures a Mutable Variable)

Consider the following program:

```
// Return a function that counts how many times it is called
fun makeCounter(): () -> int = {
  let mutable x: int = 0;
  fun () ->
    x <- x + 1 // x is captured from the surrounding scope
};

// This is interpreted and compiled incorrectly!
let c1: () -> int = makeCounter();
let c2: () -> int = makeCounter();
assert(c1() = 1);
assert(c1() = 2);
assert(c2() = 1)
```

Here we have two issues:

- when `makeCounter` is interpreted, according to [Definition 15](#) it returns the lambda term `fun () -> x <- x + 1` — and when such a lambda term is called on the first assertion, the program gets stuck on the expression `assert((x <- x + 1) = 1)` (due to the unbound variable `x`); moreover,
- when `makeCounter` is compiled, it assigns register `t0` to variable `x` — but when the returned lambda term is called in the first assertion, then register `t0` is used to hold the variable `c1`. Therefore, the lambda term is reading the wrong register.

To address the issues highlighted in [Example 47](#) and [Example 48](#) above, we first discuss *what is a closure*. Then, we address the limitations of Hygge and hyggec in two steps of increasing difficulty:

- *Closures that Capture Immutable Variables* (which are already supported by the hyggec interpreter, but are not correctly supported by the code generation);
- *Closures that Capture Mutable Variables* (which are not correctly supported neither by the hyggec interpreter, nor the code generation).

We also address the special case of *Closures that Capture Top-Level Variables*, which roughly correspond to accessing global variables in a programming language like C.

9.2 What is a Closure?

The term **closure** was introduced in 1964 by Peter Landin to describe a lambda term that “closes over” variables that are defined in its surrounding scope. More precisely, we define what is a closure by using the notion of **free variable** (*Definition 34* below), and then formalising what it means for a Hygge expression to **capture a variable** (*Definition 35* below).

9.2.1 Free Variables

A **free variable** (a.k.a. **unbound variable**) is a variable that appears in a Hygge expression without being **bound** (i.e. defined) earlier, according to *Definition 34* below.

Definition 34 (Free Variables of a Hygge Expression)

The set of **free variables of a Hygge expression** e (written $\text{fv}(e)$) is defined as follows:

$$\begin{aligned}
 \text{fv}(x) &= \{x\} \\
 \text{fv}(v) &= \emptyset \quad (\text{when } v \text{ is not a lambda term}) \\
 \text{fv}(\text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e) &= \text{fv}(e) \setminus \{x_1, \dots, x_n\} \\
 \text{fv}(e_1 + e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
 \text{fv}(e_1 * e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
 \text{fv}(e_1; e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
 &\vdots \\
 \text{fv}(\text{struct } \{f_1=e_1; \dots; f_n=e_n\}) &= \text{fv}(e_1) \cup \dots \cup \text{fv}(e_n) \\
 \text{fv}(e(e_1, \dots, e_n)) &= \text{fv}(e) \cup \text{fv}(e_1) \cup \dots \cup \text{fv}(e_n) \\
 &\vdots \\
 \text{fv}(\text{let } x : t = e_1; e_2) &= \text{fv}(e_1) \cup (\text{fv}(e_2) \setminus \{x\}) \\
 \text{fv}(\text{let mutable } x : t = e_1; e_2) &= \text{fv}(e_1) \cup (\text{fv}(e_2) \setminus \{x\}) \\
 &\vdots \\
 \text{fv}(\text{type } x = t; e) &= \text{fv}(e)
 \end{aligned}$$

The intuition behind *Definition 34* is that a variable x is free in a Hygge expression e when x occurs in e , but there is at least one occurrences of x at least one occurrence of x such that:

1. x is not in the scope of any “let $x : t = \dots$ ” nor “let mutable $x : t = \dots$ ” binder in e , and
2. x is not in the scope of any lambda term that has x among its arguments.

Consequently, $\text{fv}(e)$ is computed as follows:

1. take all the sub-expressions e_1, e_2, \dots of e ;
2. for each sub-expression, recursively compute the sets of free variables $\text{fv}(e_1), \text{fv}(e_2), \dots$;

3. if e itself is binding some variable x , then subtract x from the free variables of the scope where x is defined (as in the cases for “let...”, “let mutable...”, and lambda terms);
4. finally, compute the union of the remaining sets of free variables.

This recursive computation of free variables terminates with the base cases where either:

- e is a “simple” value v that is *not* a lambda term: in this case, the set of free variables is the empty set \emptyset ; or
- e is a variable x : in this case, the set of free variables is the singleton set $\{x\}$.

Example 49 (Free Variables of a Hygge Expression)

Consider the following Hygge expression:

$$x + y$$

The free variables in the expression above are:

$$\text{fv}(x + y) = \text{fv}(x) \cup \text{fv}(y) = \{x\} \cup \{y\} = \{x, y\}$$

Now consider the expression:

$$\text{let } x : \text{int} = 3; x + y$$

The free variables in the expression above are:

$$\begin{aligned} \text{fv}(\text{let } x : \text{int} = z; x + y) &= \text{fv}(z) \cup (\text{fv}(x + y) \setminus \{x\}) \\ &= \{z\} \cup ((\text{fv}(x) \cup \text{fv}(y)) \setminus \{x\}) \\ &= \{z\} \cup (\{x, y\} \setminus \{x\}) \\ &= \{z\} \cup \{y\} \\ &= \{z, y\} \end{aligned}$$

Note: There is a strong connection between:

- the free variables of an expression e (according to [Definition 34](#)), and
- the substitution of a variable x in an expression e (according to [Definition 2](#) and its extensions in [Definition 14](#), [Definition 18](#), [Definition 22](#), and [Definition 28](#)).

In fact, a substitution $e[x \mapsto e']$ can have an effect (i.e. return an expression that is different from e) *only if* x is a free variable of e , i.e. only if $x \in \text{fv}(e)$. If $x \notin \text{fv}(e)$ (e.g. because x only appears under a “let x...” binder in e , or because x does not occur in e at all), then the substitution $e[x \mapsto e']$ should return e without any change. It is very important to preserve this property whenever the Hygge (or any other programming language) is extended with new expressions.

Important: When we talk about the “free variables” and “captured variables” of an expression e , we are *only* referring to the variables of e that can be substituted by values

in the Hygge semantics (e.g. by a “let...” binder or a function call/application). We are *not* referring to type variables introduced by “type x...”; in fact, the case for $\text{fv}(\text{type } x = t; e)$ in [Definition 34](#) simply ignores the type variable x and computes $\text{fv}(e)$.

Exercise 33 (Computing the Free Variables of a Hygge Expression)

Compute the free variables of the following expressions, according to [Definition 34](#):

- $\text{let } x : \text{int} = 3; x + 4$
 - $\text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y$
 - $\text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + z$
 - $\text{let } f : t = \text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y; f(2, z)$
-

Exercise 34 (Defining the Free Variables of Hygge Expressions)

[Definition 34](#) is incomplete. Provide a definition of the missing cases: you should define one new case for each form of expression e that is omitted in [Definition 34](#), such as $<$, $=$, $\text{if } \dots \text{ then } \dots \text{ else } \dots$, $\text{print}(\dots)$. Write some examples showing how the updated definition of free variables works.

9.2.2 Captured Variables and Closures

A **captured variable** (a.k.a. **closed-over variable**) is a variable that appears free inside a Hygge value – and more specifically, inside a lambda term (since lambda terms are the only kind of values that can contain variables). This is formalised in [Definition 35](#) below.

Definition 35 (Captured Variables and Closures)

The set of **variables captured (or closed over)** by an expression e (written $\text{cv}(e)$) is

defined as follows:

$$\begin{aligned}
 \text{cv}(x) &= \emptyset \\
 \text{cv}(v) &= \text{fv}(v) \\
 \text{cv}(e_1 + e_2) &= \text{cv}(e_1) \cup \text{cv}(e_2) \\
 \text{cv}(e_1 * e_2) &= \text{cv}(e_1) \cup \text{cv}(e_2) \\
 \text{cv}(e_1; e_2) &= \text{cv}(e_1) \cup \text{cv}(e_2) \\
 &\vdots \\
 \text{cv}(\text{struct } \{f_1=e_1; \dots; f_n=e_n\}) &= \text{cv}(e_1) \cup \dots \cup \text{cv}(e_n) \\
 \text{cv}(e(e_1, \dots, e_n)) &= \text{cv}(e) \cup \text{cv}(e_1) \cup \dots \cup \text{cv}(e_n) \\
 &\vdots \\
 \text{cv}(\text{let } x : t = e_1; e_2) &= \text{cv}(e_1) \cup (\text{cv}(e_2) \setminus \{x\}) \\
 \text{cv}(\text{let mutable } x : t = e_1; e_2) &= \text{cv}(e_1) \cup (\text{cv}(e_2) \setminus \{x\}) \\
 &\vdots \\
 \text{cv}(\text{type } x = t; e) &= \text{cv}(e)
 \end{aligned}$$

We say that a **variable** x is **captured (or “closed over”)** in an expression e if $x \in \text{cv}(e)$.

When $\text{cv}(v) \neq \emptyset$, then we say that v is a **closure**.

The crucial part of [Definition 35](#) above is the case “ $\text{cv}(v) = \text{fv}(v)$ ”, which says that a variable is captured if it appears free inside a value v . There is only one kind of values that (according to [Definition 34](#)) can contain free variables: this kind of values are lambda terms. Therefore, [Definition 35](#) says that a lambda term $v = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ captures (or “closes over”) a variable x when x appears in v as a free variable. This means that:

- x is *not* one of the arguments x_1, \dots, x_n of v , and
- there is some occurrence of x in the lambda term body e that is *not* bound by any “let x...” or “let mutable x...”, *nor* by the arguments of any lambda term inside e .

When a value v captures at least one variable (i.e. when $\text{cv}(v) \neq \emptyset$), then we say that v is a closure; as discussed above, this can only happen if v is a lambda term.

More generally, according to [Definition 35](#), a variable x is captured in an expression e if e contains (as a sub-expression) a lambda term that captures x .

Now, consider a program containing a lambda term e that captures variable x : if such a program type-checks, then any free occurrence of x in e must be a reference to a variable that is defined in the scope *surrounding* e . This means that, when interpreting e (or generating code for e), we must make sure that the correct value for the captured x (or the correct storage location for x) is used. In this Module we explore how to do it.

Example 50 (Captured Variables of a Hygge Expression)

Consider the following Hygge expression:

$$x + y$$

The captured variables in the expression above are:

$$\text{cv}(x + y) = \text{cv}(x) \cup \text{cv}(y) = \emptyset \cup \emptyset = \emptyset$$

Now consider the expression:

$$\text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y + z$$

The captured variables in the expression above are:

$$\begin{aligned} \text{cv}(\text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y + z) &= \text{fv}(\text{fun } (x : \text{int}, y : \text{int}) \rightarrow x + y + z) \\ &= \text{fv}(x + y + z) \setminus \{x, y\} \\ &= (\text{fv}(x) \cup \text{fv}(y) \cup \text{fv}(z)) \setminus \{x, y\} \\ &= \{x, y, z\} \setminus \{x, y\} \\ &= \{z\} \end{aligned}$$

Exercise 35 (Computing the Captured Variables of a Hygge Expression)

Compute the captured variables of the following expressions, according to [Definition 35](#):

- `let x : int = 3; x + 4`
- `fun (x : int, y : int) → x + y`
- `fun (x : int, y : int) → x + z`
- `let f : t = fun (x : int, y : int) → x + z; f (2, w)`

Exercise 36 (Defining the Captured Variables of Hygge Expressions)

[Definition 35](#) is incomplete. Provide a definition of the missing cases: you should define one new case for each form of expression e that is omitted in [Definition 34](#), such as `<`, `=`, `if ... then ... else ...`, `print(...)`. Write some examples showing how the updated definition of captured variables works.

9.3 Closures that Capture Immutable Variables

This section addresses the situation illustrated in [Example 47](#) above: we have a lambda term $v = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ that captures an *immutable* variable x which is defined in a surrounding “let...” binder (by [Definition 36](#)). Observe that:

- the captured variable x may *not* be in the scope of the lambda term v when v is executed. Indeed, in [Example 47](#), the lambda term that captures x is executed *outside* the scope of the function `makeAdder` which defines x as one of its arguments. Moreover,
- the closure v may be instantiated multiple times, and the captured variable x may have a different value in each instance. Indeed, in [Example 47](#), the lambda term that captures x is instantiated with $x = 1$ in `add1`, and $x = 2$ in `add2`.

The two points above suggest that:

1. we need to store the value of x in a dedicated location that is never used for other purposes. Clearly, such a dedicated location cannot be a register; and
2. we may need to allocate and keep available multiple values of x , depending on how many times `makeAdder` is invoked and x is closed over. Therefore, the allocation of x must be performed *dynamically*, as needed, when the program runs.

To achieve all of this we apply a code transformation called **closure conversion**, explained below.

9.3.1 Closure Conversion of a Lambda Term

Intuitively, when generating code for a lambda term v , we perform its closure conversion as follows.

1. We save all variables captured by v in a structure representing the **closure environment**, which we will often call env below. Just like all Hygge structures, env is dynamically allocated on the **memory heap**.
2. We rewrite the lambda term v into a lambda term v' that, when applied, performs the same computations of v — except that:
 - v' is a “plain” function that does *not* capture any variable. Instead,
 - v' takes the closure environment structure env as an additional argument, and uses env 's fields instead of the variables captured by v .
3. We keep the rewritten v' and its closure environment env “together” and make sure that, whenever the program being compiled tries to apply the original lambda term v , the generated code applies the plain function v' instead, passing the closure environment env as an additional argument.

This intuition is illustrated in [Example 51](#) below; then, we explore closure conversion in full detail.

Example 51 (Closure Conversion: an Intuition)

Consider again the following Hygge program, taken from [Example 47](#):

```
1 // Take x, return a function that adds x to its argument
2 fun makeAdder(x: int): (int) -> int =
3     fun (y: int) ->
4         x + y; // x is captured from the surrounding scope
5
6 let add1: (int) -> int = makeAdder(1);
7 let add2: (int) -> int = makeAdder(2);
8
9 // These assertions succeed in the interpreter, but fail in code generation!
10 assert(add1(40) = 41);
11 assert(add2(40) = 42)
```


Our goal is to closure-convert the lambda term defined on lines 3 and 4, which captures the variable x from the surrounding scope. The converted code looks as follows: (Note: this is just pseudo-code)

```

1 // Take x, return a function that adds x to its argument
2 fun makeAdder(x: int): (int) -> int = {
3     // Capture environment, with a field for each captured variable
4     let env: struct {x: int} = struct {x = x};
5     // Rewritten lambda term, taking 'env' as an additional argument
6     let v' = fun (env: struct {x: int}, y: int) ->
7         env.x + y; // The captured x is replaced by env.x
8     // We return a pointer to a closure structure containing v' and env
9     struct {f = v'; env = env}
10 }
11
12 let add1: (int) -> int = makeAdder(1); // Points to a struct with f and env
13 let add2: (int) -> int = makeAdder(2); // Points to a struct with f and env
14
15 // The application 'add1(40)' is compiled as: (add1.f)(add1.env, 40)
16 assert(add1(40) = 41);
17 // The application 'add2(40)' is compiled as: (add2.f)(add2.env, 40)
18 assert(add2(40) = 42)

```

We now explore in more detail how closure conversion works. Suppose that we are generating code for the following lambda term v , having type T_v :

$$\begin{aligned}
 v &= \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e \\
 T_v &= (T_1, \dots, T_n) \rightarrow T
 \end{aligned}$$

To closure-convert v above, we proceed as follows.

Important: Recall that, when we generate code for a lambda term v , we *produce the memory address where the lambda term's code is stored* (in the Text segment of the generated assembly program).

1. We compute the set C containing the variables captured by v , i.e. $C = \text{cv}(v)$.

Note: This description of closure conversion also works if C is empty, i.e. the lambda term v does not capture any variable. In this case, in the next steps we will create an empty env structure that will be unused.

2. Suppose that the set of captured variables is $C = \{y_1, \dots, y_m\}$, and that their types are respectively T'_1, \dots, T'_m . We perform the **closure conversion** of v , as follows.
 - We define the **closure environment** structure instance env , having the structure type T_{env} :

$$\begin{aligned}
 env &= \text{struct } \{y_1 = y_1; \dots; y_m = y_m\} \\
 T_{env} &= \text{struct } \{y_1 : T'_1; \dots; y_m : T'_m\}
 \end{aligned}$$

In other words, env is a structure where each field has the same name of a captured variable y_i ; moreover, each field $env.y_i$ is initialised with the value of the corresponding variable y_i .

- We rewrite the lambda term v as v' below, with type $T_{v'}$:

$$\begin{aligned} v' &= \text{fun } (env : t_{env}, x_1 : t_1, \dots, x_n : t_n) \rightarrow e [y_1 \mapsto env.y_1] \dots [y_m \mapsto env.y_m] \\ T_{v'} &= (T_{env}, T_1, \dots, T_n) \rightarrow T \end{aligned}$$

(Above, t_{env} is a pretype corresponding to type T_{env} .) In other words:

- we add an argument called env to the rewritten lambda term v' , such that env has type T_{env} above (hence, env is a structure with one field for each captured variable y_i , for $i \in 1..m$); and
- we rewrite the body e of the lambda term v such that, in the rewritten v' , all references to any captured variable y_i are replaced by a corresponding structure field selection $env.y_i$.

As a consequence, the rewritten v' is similar to v , except that v' is a “plain” function that does not capture any variable; in fact, v' only accesses its arguments (including its additional argument env).

- We create the following structure instance $clos$, having type T_{clos} :

$$\begin{aligned} clos &= \text{struct } \{f = v'; env = env\} \\ T_{clos} &= \text{struct } \{f : T_{v'}; env : T_{env}\} \end{aligned}$$

The structure $clos$ is the actual representation of an instance of the lambda term v : in fact, $clos$ “keeps together” a pointer to the “plain” function v' , and the closure environment env created when v is instantiated.

3. We complete the compilation of the lambda term v by producing the memory address of the structure $clos$ (instead of the memory address of the original v).

9.3.2 Applying a Closure-Converted Lambda Term

When we closure-convert each lambda term v in a Hygge program (as described *above*), we also need to revise the code generation for application expressions. This is because:

- before introducing closure conversion, *the code generation for a lambda term yielded the memory address of the lambda term code*. Therefore, to “call” a function we just needed to jump to that memory address;
- however, after introducing closure conversion, the code generation for a lambda term v yields the memory address of a closure structure $clos$ having the two fields $clos.f$ and $clos.env$, where:
 - the first field $clos.f$ is the memory address of the “plain” function v' that implements v . When called, $clos.f$ expects to receive the structure $clos.env$ as first argument, followed by any other argument expected by v ;

- the second field `clos.env` is the memory address of a structure containing the **closure environment**, i.e. a copy of all variables captured by v , as they were when v was instantiated.

Consequently, suppose that we want to apply a lambda term v to some arguments e_1, \dots, e_n , i.e.:

$$v(e_1, \dots, e_n)$$

To perform this application, we now need to “apply” the closure structure $clos$ (corresponding to v) to the arguments e_1, \dots, e_n . Hence, the code generation for application expressions must be revised to perform the following application:

$$clos.f(clos.env, e_1, \dots, e_n)$$

i.e. we need to apply the “plain” function $clos.f$ to the closure environment $clos.env$ followed by the arguments e_1, \dots, e_n .

9.4 Closures that Capture Mutable Variables

This section addresses the situation illustrated in [Example 47](#): we have a lambda term $v = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ that captures a *mutable* variable x which is defined by a surrounding “let mutable...” binder. This situation is, to some extent, similar to [Closures that Capture Immutable Variables](#): indeed, the same closure conversion solution can address the simple case shown in [Example 47](#). Unfortunately, this approach does *not* work correctly in more complex cases, like [Example 52](#) below.

Example 52 (Two Functions that Capture a Mutable Variable)

Consider the following program:

```

1  type Counters = struct {f1: () -> int; f2: () -> int};
2
3  // Return a structure with two functions that share a counter, counting how
4  // many times they have been called.
5  fun makeCounters(): Counters = {
6      let mutable i: int = 0;
7      // The lambda terms below capture i twice
8      struct { f1 = fun () -> i <- i + 1;
9                f2 = fun () -> i <- i + 1 } : Counters
10 };
11
12 let c1: Counters = makeCounters();
13 assert(c1.f1() = 1); // c1.f1 and c2.f2 should share the same counter
14 assert(c1.f2() = 2);
15
16 let c2: Counters = makeCounters();
17 assert(c2.f2() = 1); // c2.f1 and c2.f2 should share another counter
18 assert(c2.f1() = 2)

```

Note: The type ascription “... : Counters” on line 9 can be omitted if `hyggec` is extended with *function subtyping*.

If we simply apply a *closure conversion* to the example above, then the assertions on lines 14 and 18 fail. The reason is that a simple closure conversion would rewrite the lambda terms assigned to the structure fields `f1` and `f2` as follows:

```
// ...
fun makeCounters(): Counters = {
  let mutable i: int = 0;

  // The lambda terms below do not capture any variable
  struct { f1 = struct { f = fun (env: ...) -> env.i <- env.i + 1;
                       env = struct { i = i } } }

          f2 = struct { f = fun (env: ...) -> env.i <- env.i + 1;
                       env = struct { i = i } } } : Counters
};
// ...
```

Therefore, each lambda term assigned to the structure fields `f1` and `f2` would be transformed into a closure structure with its own environment, containing its own copy of the captured variable `i`. As a consequence, the function application on lines 14 and 18 will return 1, because calling `c1.f1` would not change the counter used by `c1.f2` — and similarly, calling `c2.f2` would not change the counter used by `c2.f1`.

To correctly support *Example 52*, we need to add another transformation step: we **rewrite captured mutable variables by moving them into the memory heap**. This technique is also *adopted by F#*³⁶. More in detail, we follow these steps.

1. Whenever we encounter a binder “let mutable $x : t = e; e'$ ”, we check whether the variable x is captured somewhere in the scope e' — i.e. we check whether $x \in \text{cv}(e')$ (according to *Definition 35*).
2. If x is captured in e' , then we replace the “let mutable $x...$ ” binder with the following *immutable* binder:

$$\text{let } x : \text{struct } \{ \text{value} : t \} = \text{struct } \{ \text{value} = e \}; (e' [x \mapsto x.\text{value}])$$

In other words:

- we define x as a (heap-allocated) structure with a unique field called *value*, and
- we rewrite the scope of e' by replacing each occurrence of x with the structure field selection $x.\text{value}$.

³⁶ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/values/#mutable-variables>

3. After this rewriting, the updated scope of the “let x..” binder (i.e. the expression $e' [x \mapsto x.value]$) is still capturing x — but now, x is an *immutable* variable (pointing to a mutable structure on the heap).
4. We recursively repeat the above rewriting on a whole Hygge program, turning any captured mutable variable into an immutable variable pointing to a heap structure.
5. As a consequence, all lambda terms are rewritten to only capture immutable variables: hence, we can compile them by applying a normal *closure conversion*.

Example 53 (Rewriting Captured Mutable Variables)

Consider again the program in *Example 52*. The function `makeCounter` can be rewritten as follows, to move its captured mutable variable `i` into the heap:

```
// ...
fun makeCounters(): Counters = {
  let i: struct {value: int} = struct {value = 0};
  // The lambda terms below capture i twice
  struct { f1 = fun () -> i.value <- i.value + 1;
           f2 = fun () -> i.value <- i.value + 1 } : Counters
};
// ...
```

After this rewriting, the closure conversion discussed in *Closures that Capture Immutable Variables* will assign to `f1` and `f2` their own capture environments, each one with its own copy of `i` — with the following resulting code:

```
// ...
fun makeCounters(): Counters = {
  let i: struct {value: int} = struct {value = 0};

  // The lambda terms below do not capture any variable
  struct { f1 = struct { f = fun (env: ...) ->
                        env.i.value <- env.i.value + 1;
                        env = struct { i = i } }

           f2 = struct { f = fun (env: ...) ->
                        env.i.value <- env.i.value + 1;
                        env = struct { i = i } } } : Counters
};
// ...
```

Since `i` is now a pointer to a structure on the heap, the fields `env.i` in the two closure environments point to the same structure on the heap; therefore, the updates to `i.value` will be applied to the same memory location, and will be visible across the two closures.

Note: This rewriting strategy turns a captured mutable variable into a *reference cell*.

9.5 Closures that Capture Top-Level Variables

This section addresses the special case of variable capture illustrated in *Example 54* below: we have a lambda term $v = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e'$ that captures a (mutable or immutable) variable x which is defined by a *top-level* “let” binder.

Example 54 (A Function that Captures a Top-Level Variable)

Consider the following program, which captures a (mutable or immutable) variable x :

```
let x: int = 1; // This example can be adapted with x mutable

// Take an argument y, return x + y + 1 (using x above)
fun addXPlusOne(y: int): int = {
  let y1: int = y + 1;
  x + y1 // x is captured from the surrounding scope
};

// This is compiled incorrectly!
assert(addXPlusOne(40) = 42)
```

This program is *not* compiled correctly: when the generated assembly code runs, the assertion fails. The reason is that the variable x is assigned to register $t0$ — but when the function `addXPlusOne` is called, then $t0$ is overwritten by the content of variable $y1$. Therefore, the compiled lambda term returned by `addXPlusOne` ends up computing $y1 + y1$ (which is incorrect).

We can fix the issue outlined in *Example 54* by applying the code transformations for *immutable* and *mutable* variables discussed in the previous sections (depending on whether the captured variable x is mutable or immutable).

However, *Example 54* also allows for an optimisation, based on the fact that *the lambda term that captures x never exits the scope of x* ; this happens because x is a “**top-level**” binder in Hygge, i.e. it is akin to a global variable in C. The idea of “top-level” binder is made precise in *Definition 36* below.

Definition 36 (Top-Level and Local “let...” Binders and Variables)

Consider an Hygge program e . Take any binder e' that occurs as a subterm of the program e , and such that:

- $e' = \text{let } x : t = \dots$, or
- $e' = \text{let mutable } x : t = \dots$

We say that the binder e' and its variable x are *top-level* in the program e when:

- e' is *not* inside the initialisation expression of another “let” binder, and
- e' is *not* inside the body of a function.

We say that the binder e' and its variable x are *local* when they are *not* top-level in the program e .

Note: When a variable x is “top-level” according to [Definition 36](#), it is intuitively similar to a “global” variable in programming languages like C; however, a top-level variable is *not* necessarily visible in a whole Hygge program, because its scope may be limited. For instance, consider:

```
{
  let x: int = 40;
  let f: () -> int = fun() -> 2 + x;
  assert(f() = 42)
}
println("Hello")
```

In the program above, the variable x is “top-level” according to [Definition 36](#) (and is also captured by the lambda term f); however, the scope of x is limited by the curly brackets, hence x is not visible in the last line of the program.

Now, let us consider again [Example 54](#) and the lambda term $v = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e$ that captures a top-level variable x : the key observation is that, since x is top-level, the scope of x always “surrounds” the lambda term v , whenever v is called (i.e. applied to some values); moreover, x is *not* dynamically instantiated: unlike the previous examples, we know *at compile time* that at most one instance of the top-level variable x may exist.

Therefore, we can avoid the issues described in [Example 54](#) without resorting to a full-fledged closure conversion of x , and without using the heap for storing x :

1. we can allocate the memory address for x *statically* during code generation, in the Data segment of the generated assembly program; and
2. during code generation for closures that capture *immutable* or *mutable* variables (as described in the previous sections), we can simply disregard any top-level variable x , and focus on *local* variables only.

Remark 2

The approach of storing a variables in a Data segment memory location is already adopted in the [Code Generation for Named Functions](#). Besides using less registers, this code generation strategy allows a function to call another function defined in its surrounding scope, without running into the capturing issues highlighted in [Example 54](#).

9.6 Implementation

This section discusses a hyggec extension that introduces new functions to compute the free and captured variables of an AST node. These functions are a stepping stone towards implementing closures — but to see what else is required, please see the *Project Ideas*.

Tip: To see a summary of the changes described below, you can inspect the differences in the *hyggec Git repository* between the tags `structures` and `free-captured-vars`.

9.6.1 Free Variables of an AST Node

The function `freeVars` in the file `ASTUtil.fs` is an implementation of *Definition 34*. As usual, it is a pattern matching over all possible Hygge expressions, and its code looks as follows.

```
// Compute the set of free variables in the given AST node.
let rec freeVars (node: Node<'E, 'T>): Set<string> =
  match node.Expr with
  | UnitVal
  | IntVal(_)
  | BoolVal(_)
  | FloatVal(_)
  | StringVal(_)
  | Pointer(_) -> Set[]
  | Var(name) -> Set[name]
  | Add(lhs, rhs)
  | Mult(lhs, rhs) ->
    Set.union (freeVars lhs) (freeVars rhs)
  // ...
  | Let(name, _, init, scope)
  | LetMut(name, _, init, scope) ->
    // All the free variables in the 'let' initialisation, together with all
    // free variables in the scope --- minus the newly-bound variable
    Set.union (freeVars init) (Set.remove name (freeVars scope))
  // ...
```

9.6.2 Captured Variables of an AST Node

The function `capturedVars` in the file `ASTUtil.fs` is an implementation of *Definition 35*. As usual, it is a pattern matching over all possible Hygge expressions, and its code looks as follows.

```
// Compute the set of captured variables in the given AST node.
let rec capturedVars (node: Node<'E, 'T>): Set<string> =
  match node.Expr with
```

(continues on next page)

(continued from previous page)

```

| UnitVal
| IntVal(_)
| BoolVal(_)
| FloatVal(_)
| StringVal(_)
| Pointer(_)
| Lambda(_, _) ->
    // All free variables of a value are considered as captured
    freeVars node
| Var(_) -> Set[]
| Add(lhs, rhs)
| Mult(lhs, rhs) ->
    Set.union (capturedVars lhs) (capturedVars rhs)
//...
| Let(name, _, init, scope)
| LetMut(name, _, init, scope) ->
    // All the captured variables in the 'let' initialisation, together with
    // all captured variables in the scope --- minus the newly-bound var
    Set.union (capturedVars init) (Set.remove name (capturedVars scope))
// ...

```

9.7 References and Further Readings

This is the seminal paper that introduces the notion of closure, and a transformation that is very similar to the closure conversion illustrated in this module:

- Peter Landin. *The Mechanical Evaluation of Expressions*. The Computer Journal, Volume 6, Issue 4, January 1964. Available on DTU Findit³⁷.

50 years later, Java 8 introduced lambda expressions, which are essentially syntactic sugar for anonymous classes that can capture local variables. Notably, lambda expressions in Java can only capture local *immutable* variables (which are called “effectively final” in the Java documentation); capturing mutable variables causes a compilation error. For more details:

- <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

³⁷ <https://findit.dtu.dk/en/catalog/5cb987c5d9001d01a410601f>

9.8 Project Ideas

These project ideas extend Hygge and hyggec with consistent support for closures. The meaning of “consistent” is: if a form of closure is not correctly supported, then it is rejected by the type checker.

Therefore, you can choose **one among the following 3 combinations of Project Ideas** (which are presented in order of increasing difficulty).

1. **C-style closures (i.e. almost no closures).** This combination of Project Ideas is the most restrictive: it only supports the simplest form of closure (similar to using “global variables” in C), and rejects any form of local variable closure by issuing type checking errors.
 - *Project Idea: Code Generation for Top-Level Closures*
 - *Project Idea: Forbid Closures of Local Variables*
2. **Java-style closures (i.e. only immutable closures).** This combination of Project Ideas is similar to Java, where lambda expressions can only capture immutable variables (as discussed in the *References and Further Readings*) and capturing mutable variables causes a type checking error.
 - *Project Idea: Code Generation for Closures of Immutable Variables*
 - *Project Idea: Forbid Closures of Mutable Variables*
 - **Optional challenge:** when a lambda term captures a *top-level* immutable variable x , avoid copying x onto the heap during code generation, by adapting and integrating *Project Idea: Code Generation for Top-Level Closures*
3. **F#-style closures.** This combination of Project Ideas is the most advanced, and similar to the features of most functional programming languages (including F#): type checking is unchanged, but the interpreter and code generation are improved to deliver full support for closures.
 - *Project Idea: Code Generation for Closures of Immutable Variables*
 - *Project Idea: Support Closures of Mutable Variables*
 - **Optional challenge:** when a lambda term captures a *top-level* variable x , avoid moving x onto the heap during code generation, by adapting and integrating *Project Idea: Code Generation for Top-Level Closures*

Note: To implement the Project Ideas above, you should *not* change the syntax of Hygge nor the lexer/parser of hyggec.

Important: To implement the Project Ideas above, you will also need to extend the definitions of `freeVars` and `capturedVars` in `ASTUtil.fs` (discussed in the *Implementation* section above): you will need to add new cases to cover any expression that you added

to `hyggec` in one of the previous Project Ideas. This corresponds to extending *Definition 34* (free variables) and *Definition 35* (captured variables).

All cases should be straightforward. If you have chosen the *Project Idea on recursive functions*, then you will need to add the following case to *Definition 34*: (notice the slight difference with respect to the existing case for “let $x : t = \dots$ ”)

$$\text{fv}(\text{let rec } x : t = e_1; e_2) = (\text{fv}(e_1) \cup \text{fv}(e_2)) \setminus \{x\}$$

9.8.1 Project Idea: Code Generation for Top-Level Closures

The goal of this Project Idea is to implement the limited form of closure described in *Closures that Capture Top-Level Variables*, without supporting closures that capture local variables.

You should extend the code generation environment `CodegenEnv` (in the file `RISCVCodegen.fs`) with a new field that tracks whether code generation is taking place in a “top-level” part of the input program. This new field could be called e.g. `env.AtTopLevel` and have type `boolean`, and it should be updated during code generation:

- it should be set to `true` at the beginning of the code generation, and
- it should be turned to `false` when entering the initialisation expression of a `let` binder, or the body of a lambda term, according to *Definition 36*.

When `env.AtTopLevel` is `true`, and you are generating code for a “let $x \dots$ ” or “let mutable $x \dots$ ” binder, you can:

- assign to the variable x a Data segment memory location marked by a unique assembly label called e.g. `label_var_x`; and
- in the code generation environment `env`, let `env.VarStorage` map the variable x to `Storage.Label("label_var_x")`. This way, any attempt to access x will look into the corresponding memory address marked by the label.

You will also need to update `doCodegen` in `RISCVCodegen.fs`, as follows:

- you will need to extend the code generation case for `Var(name)`, since it assumes that, if a variable is stored in a data segment label, then it represents the address of a function. To support other types of values, you will need to add a new case similar to the following:

```
let rec internal doCodegen (env: CodegenEnv) (node: TypedAST): Asm =
  match node.Expr with
  // ...
  | Var(name) ->
    // To compile a variable, we inspect its type and where it is stored
    match node.Type with
    // ...
    | _ -> // Default case for variables holding integer-like values
```

(continues on next page)

(continued from previous page)

```

match (env.VarStorage.TryFind name) with
// ...
| Some(Storage.Label(lab)) ->
    match (expandType node.Env node.Type) with // <-- New_
↳from here
    | TFun(_,_) ->
        Asm(RV.LA(Reg.r(env.Target), lab), $"Load variable '%s
↳{name}''")
    | _ ->
        Asm([(RV.LA(Reg.r(env.Target), lab),
            $"Load address of variable '%s{name}''")
            (RV.LW(Reg.r(env.Target), Imm12(0), Reg.r(env.
↳Target))),
            $"Load value of variable '%s{name}''")])

```

- you will also need to extend the code generation case for `Assign(lhs, rhs)`, since it does not currently support target variables that are stored in a memory address.

You should describe how you modify the `hygdec` compiler to achieve this extension. As usual, you should also provide tests that leverage this extension: you can use [Example 54](#) as a starting point.

9.8.2 Project Idea: Forbid Closures of Local Variables

The goal of this project idea is to reject any Hygge program that cannot be correctly compiled after implementing [Project Idea: Code Generation for Top-Level Closures](#). Therefore, the goal is to detect captured *local* variables at compile-time. More specifically, the goal is to make the Hygge typing system and the `hygdec` type checking more strict, by rejecting closures of (mutable or immutable) *local* variables, according to [Definition 36](#).

To this end, you should extend the typing environment Γ with a new field, called $\Gamma.\text{AtTopLevel}$, which is a boolean that:

- is true when a type checking starts, and typing rules are applied in the top-level part of a program, and
- becomes false when entering the initialisation expression of a “let $x : t = \dots$ ” or “let mutable $x : t = \dots$ ” binder, or the body of a lambda term, according to [Definition 36](#).

Then, you should implement the typing rules presented in [Definition 37](#) below.

Definition 37 (Typing Rules for Rejecting Closures of Local Variables)

We define the following typing rules, that replace rules [T-MLet] and [T-Let2] in [Definition 16](#), and rule [T-Fun] in [Definition 26](#).

$$\frac{\Gamma \vdash t \triangleright T \quad \{\Gamma \text{ with } \text{AtTopLevel}=\text{false}\} \vdash e_1 : T \quad \left\{ \Gamma \text{ with } \begin{array}{l} \text{Vars} + (x \mapsto T) \\ \text{Mutables} \cup \{x\} \end{array} \right\} \vdash e_2 : T' \quad \begin{array}{l} \Gamma.\text{AtTopLevel} \\ \text{or } x \notin \text{cv}(e_2) \end{array}}{\Gamma \vdash \text{let mutable } x : t = e_1; e_2 : T'} \quad [\text{T-MLet2}]$$

$$\frac{\Gamma \vdash t \triangleright T \quad \{\Gamma \text{ with } \text{AtTopLevel}=\text{false}\} \vdash e_1 : T \quad \left\{ \Gamma \text{ with } \begin{array}{l} \text{Vars} + (x \mapsto T) \\ \text{Mutables} \setminus \{x\} \end{array} \right\} \vdash e_2 : T' \quad \begin{array}{l} \Gamma.\text{AtTopLevel} \\ \text{or } x \notin \text{cv}(e_2) \end{array}}{\Gamma \vdash \text{let } x : t = e_1; e_2 : T'} \quad [\text{T-Let3}]$$

$$\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e : T \quad \left\{ \Gamma \text{ with } \begin{array}{l} \text{Vars} + (x \mapsto T) \\ \text{Mutables} \setminus \{x\} \end{array} \right\} \vdash e_2 : T'}{\Gamma \vdash \text{let } x : t = \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e; e_2 : T'} \quad [\text{T-LetFun}]$$

$$\frac{x_1, \dots, x_n \text{ pairwise distinct} \quad \forall i \in 1..n : \Gamma \vdash t_i \triangleright T_i \quad \left\{ \Gamma \text{ with } \begin{array}{l} \text{Vars} + \{x_i \mapsto T_i\}_{i \in 1..n} \\ \text{AtTopLevel} = \text{false} \end{array} \right\} \vdash e : T}{\Gamma \vdash \text{fun } (x_1 : t_1, \dots, x_n : t_n) \rightarrow e : (T_1, \dots, T_n) \rightarrow T} \quad [\text{T-Fun2}]$$

The only differences between the rules in [Definition 37](#) and their original versions are:

- the new rules [T-MLet2], [T-Let3] and [T-Fun2] set $\Gamma.\text{AtTopLevel}$ to false when entering the initialisation expression of a “let $x : t = \dots$ ” or “let mutable $x : t = \dots$ ” binder, or the body of a lambda term, according to [Definition 36](#);
- rules [T-MLet2] and [T-Let3] do not allow capturing a local variable;
- the rule [T-LetFun] introduces a useful special case: if a “let” binder introduces a variable as a “named function” (i.e. a variable that is initialised with a lambda term) then that variable can be captured in the body of the “let...”, even when the variable is local. With this rule we can type-check named functions, according to [Remark 2](#) above. (Also notice that rule [T-LetFun] corresponds to [T-Let2] in [Definition 16](#), except that [T-LetFun] is restricted to lambda terms).

You should describe how you modify the hygge compiler to achieve this extension. As usual, you should also provide tests that leverage this extension: you can use [Example 54](#) as a starting point.

9.8.3 Project Idea: Code Generation for Closures of Immutable Variables

The goal of this Project Idea is to support *Closures that Capture Immutable Variables*. To this end, you should modify hygge to implement closure conversion. This can be addressed in various ways; the most immediate solution is to change `RISCVCodegen.fs` as follows:

- when generating assembly code for a lambda term v , you should first rewrite the lambda term to obtain its *closure conversion*, ensuring that the resulting code produces a pointer to the closure structure for v ;
- when generating assembly code for `Application(expr, args)`, you should con-

sider that the memory address produced by `expr` is now the address of a closure structure, as discussed in *Applying a Closure-Converted Lambda Term*.

You should describe how you modify the `hygdec` compiler to achieve this extension. As usual, you should also provide tests that leverage this extension: you can use *Example 47* as a reference.

9.8.4 Project Idea: Forbid Closures of Mutable Variables

The goal of this project idea is to reject any Hygge program that cannot be correctly compiled after implementing *Project Idea: Code Generation for Closures of Immutable Variables*. More specifically, the goal is to make the Hygge typing system and the `hygdec` type checking more strict, by rejecting closures of *mutable* variables.

You should implement the typing rules presented in *Definition 38* below.

Definition 38 (Typing Rules for Rejecting Closures of Mutable Variables)

We define the following typing rule, that replaces rule [T-MLet] in *Definition 16*.

$$\frac{\Gamma \vdash t \triangleright T \quad \Gamma \vdash e_1 : T \quad \left\{ \Gamma \text{ with } \begin{array}{l} \text{Vars} + (x \mapsto T) \\ \text{Mutables} \cup \{x\} \end{array} \right\} \vdash e_2 : T' \quad x \notin \text{cv}(e_2)}{\Gamma \vdash \text{let mutable } x : t = e_1; e_2 : T'} \quad [\text{T-MLet2}]$$

The only difference between the new rule [T-MLet2] and the original rule [T-MLet] in *Definition 16* is that [T-MLet2] does not allow capturing the mutable variable being defined.

You should describe how you modify the `hygdec` compiler to achieve this extension. As usual, you should also provide tests that leverage this extension.

9.8.5 Project Idea: Support Closures of Mutable Variables

The goal of this project idea is to extend Hygge and `hygdec` to support *Closures that Capture Mutable Variables*. This involves two steps:

- *Step 1: Updating the Hygge Semantics and the hygdec Interpreter*, and
- *Step 2: Updating the hygdec Code Generation*.

For both steps, you should describe how you modify the `hygdec` compiler to achieve this extension. As usual, you should also provide tests that leverage this extension, using *Example 48* and *Example 52* as a reference.

Step 1: Updating the Hygge Semantics and the hyggec Interpreter

You need to revise the semantics of Hygge, as illustrated in [Definition 39](#) below.

Definition 39 (Semantics with Heap Promotion of Captured Mutable Variables)

We define the following semantic rules for Hygge expressions, that replace rules [R-LetM-Eval-Init] and [R-LetM-Eval-Scope] in [Definition 15](#):

$$\frac{x \notin \text{cv}(e_2) \quad \langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{let mutable } x : t = e; e_2 \rangle \rightarrow \langle R' \bullet \text{let mutable } x : t = e'; e_2 \rangle} \text{ [R-LetM-Eval-Init2]}$$

$$\frac{\begin{array}{l} x \notin \text{cv}(e) \\ R' = \{R \text{ with Mutables} + (x \mapsto v)\} \end{array} \quad \langle R' \bullet e \rangle \rightarrow \langle R'' \bullet e' \rangle \quad \begin{array}{l} R''.\text{Mutables}(x) = v' \\ R.\text{Mutables}(x) = v_? \\ R''' = \{R'' \text{ with Mutables}(x) = v_?\} \end{array}}{\langle R \bullet \text{let mutable } x : t = v; e \rangle \rightarrow \langle R''' \bullet \text{let mutable } x : t = v'; e' \rangle} \text{ [R-LetM-Eval-Scope2]}$$

$$\frac{x \in \text{cv}(e_2)}{\langle R \bullet \text{let mutable } x : t = e; e_2 \rangle \rightarrow \langle R \bullet \text{let } x : \text{struct } \{ \text{value} : t \} = \text{struct } \{ \text{value} = e \}; e_2 [x \mapsto x.\text{value}] \rangle} \text{ [R-LetM-ToStruct]}$$

In [Definition 39](#) above, the rules [R-LetM-Eval-Init2] and [R-LetM-Eval-Scope2] are identical to the corresponding rules in [Definition 15](#) – except that the new rules have an additional premise (on the left) requiring that the declared mutable variable x is *not* captured in the scope of the “let” binder. If that premise is false, then we can apply the new rule [R-LetM-ToStruct], which rewrites the “let mutable...” binder into an immutable “let” binder that initialises x as a heap-allocated structure, as described in [Closures that Capture Mutable Variables](#).

Step 2: Updating the hyggec Code Generation

You should update the hyggec code generation to support the correct compilation of [Closures that Capture Mutable Variables](#). To achieve this, you should:

1. rewrite “let mutable...” binders of captured variables according to [Definition 39](#) above, and then
2. further transform the resulting “let” binder by applying a [closure conversion](#).

Hint: You can achieve the overall code transformation by combining the interpreter code you wrote for [Step 1: Updating the Hygge Semantics and the hyggec Interpreter](#) with your work on [Project Idea: Code Generation for Closures of Immutable Variables...](#)

10

Module 10: Discriminated Unions and Recursive Types

In this module we study how to extend Hygge with **discriminated union types** (a.k.a. **sum types**) and a **pattern matching** expression, with a design inspired by the [corresponding features of the F# programming language](#)³⁸. This allows Hygge programmers to create values that can have one among different types, so they can write e.g. a function returning either an integer result, or an error string (which can be distinguished by pattern matching).

We also discuss what is needed to extend Hygge with **recursive types**: we will see that the combination of structures, discriminated unions, and recursive types enables the definition and handling of complex data structures like lists or trees.

10.1 Discriminated Union Types and Pattern Matching

We now explore the *Syntax*, *Operational Semantics*, *Typing Rules*, and *Implementation* of discriminated union types and pattern matching. But first, let us discuss the *Overall Objective*.

10.1.1 Overall Objective

By extending Hygge and hyggec with support for discriminated union types and pattern matching, our goal is to interpret, compile and run Hygge programs like the one shown in *Example 55* below.

Example 55 (A Hygge Program with Union Types and Pattern Matching)

```
1 // An optional integer value.  
2 type OptionalInt = union {
```

(continues on next page)

³⁸ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions>

(continued from previous page)

```
3   Some: int;
4   None: unit
5 };
6
7 fun displayOption(o: OptionalInt): unit =
8   match o with {
9     Some{x} -> println(x);
10    None{_] -> println("None")
11  };
12
13 displayOption(Some{42});
14 displayOption(None{()});
15
16 // A shape type.
17 type Shape = union {
18   Circle: struct { radius: float };
19   Rectangle: struct { width: float; height: float };
20   Square: struct { side: float }
21 };
22
23 // Return the area of a shape.
24 fun area(s: Shape): float =
25   match s with {
26     Circle{c} -> c.radius * c.radius * 3.14f;
27     Rectangle{r} -> r.width * r.height;
28     Square{s} -> s.side * s.side
29   };
30
31 assert(area(Circle{struct {radius = 2.0f}}) = 12.56f);
32 assert(area(Rectangle{struct {width = 2.0f; height = 3.0f}}) = 6.0f);
33 assert(area(Square{struct {side = 5.0f}}) = 25.0f)
```

10.1.2 Syntax

We extend the Hygge syntax as specified in *Definition 40* below.

Definition 40 (Syntax of Discriminated Unions)

We define the syntax of Hygge0 with discriminated unions by extending *Definition 27* (syntax of Hygge with structures) as follows.

Expression	$e ::= \dots$	
	$l\{e\}$	(Discriminated union constructor)
	$\text{match } e \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots; l_n\{x_n\} \rightarrow e_n\}$	(Pattern matching, with $n \geq 1$)
Pretype	$t ::= \dots$	
	$\text{union } \{l_1 : t_1; \dots; l_n : t_n\}$	(Discriminated union pretype, with $n \geq 1$)
Union label	$l ::= z \mid \text{foo} \mid \text{a123} \mid \dots$	(Any non-reserved identifier)

The intuition behind *Definition 40* above is that a discriminated union pretype t describes a value that can have one between several possible pretypes t_1, \dots, t_n :

- each possible case of a union pretype t consists of a **label** l_i and a type t_i , for some $i \in 1..n$;
- to construct an instance of a discriminated union, a programmer specifies a label l and an expression e (between curly brackets);
- to use a discriminated union instance, a programmer needs to use **pattern matching** to inspect v 's label and retrieve its corresponding value.

Note: In *Definition 40* above we do not introduce the syntax of new values: this is because a discriminated union constructor returns a **pointer** to a heap location, as we will see later in the *Operational Semantics*.

This is the reason why *Definition 40* above extends the syntax of Hygge structures (*Definition 27*): our specification of discriminated unions requires the pointers introduced there. Similarly, the *Operational Semantics* below will use the heap introduced with *Hygge structures*.

10.1.3 Operational Semantics

Definition 41 formalises how substitution works for discriminated union constructors and pattern matching expressions.

Definition 41 (Substitution for Discriminated Unions and Pattern Matching)

We extend *Definition 28* (substitution for Hygge with structures) with the following new cases:

$$(l\{e\})[x \mapsto e'] = l\{e[x \mapsto e']\}$$

$$\left(\begin{array}{l} \text{match } e \text{ with } \{ \\ l_1\{x_1\} \rightarrow e_1; \\ \dots; \\ l_n\{x_n\} \rightarrow e_n \\ \} \end{array} \right) [x \mapsto e'] = \begin{array}{l} \text{match } e[x \mapsto e'] \text{ with } \{ \\ l_1\{x_1\} \rightarrow e'_1; \\ \dots; \\ l_n\{x_n\} \rightarrow e'_n \\ \} \end{array} \quad \text{where } \forall i \in 1..n : e'_i = \begin{cases} e_i[x \mapsto e'] & \text{if } x_i \neq x \\ e_i & \text{if } x_i = x \end{cases}$$

Notice that in [Definition 41](#) above, a substitution of a variable x is propagated through a pattern matching case $l\{x'\} \rightarrow e$ *only if the matching variable x' is different from x* . This is because pattern matching acts as a **binder** for the matched variable x' , hence x' is considered a newly-defined variable that must be treated as distinct from any other occurrences of x' in the surrounding scope. This corresponds to the treatment of *Free and Captured Variables* of pattern matching expressions (presented later).

Definition 42 (Semantics of Discriminated Unions and Pattern Matching)

We define the **semantics of Hygge discriminated unions and pattern matching** by adding the following rules to [Definition 29](#) (semantics of Hygge with structures):

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet l\{e\} \rangle \rightarrow \langle R' \bullet l\{e'\} \rangle} \text{ [R-UnionCons-Eval]}$$

$$\frac{\begin{array}{l} h = R.\text{Heap} \\ p = \text{maxAddr}(h) + 1 \end{array} \quad h' = h + \left\{ \begin{array}{l} p \mapsto \text{"1"} \\ (p+1) \mapsto v \end{array} \right\} \quad R' = \{R \text{ with Heap} = h'\}}{\langle R \bullet l\{v\} \rangle \rightarrow \langle R' \bullet p \rangle} \text{ [R-UnionCons-Res]}$$

$$\frac{\langle R \bullet e \rangle \rightarrow \langle R' \bullet e' \rangle}{\langle R \bullet \text{match } e \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots\} \rangle \rightarrow \langle R' \bullet \text{match } e' \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots\} \rangle} \text{ [R-Match-Eval]}$$

$$\frac{\exists k \in 1..n \quad R.\text{Heap}(p) = \text{"}l_k\text{"} \quad R.\text{Heap}(p+1) = v}{\langle R \bullet \text{match } p \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots; l_n\{x_n\} \rightarrow e_n\} \rangle \rightarrow \langle R \bullet e_k[x_k \mapsto v] \rangle} \text{ [R-Match-Res]}$$

The rules in [Definition 42](#) above work as follows.

- By rule [R-UnionCons-Eval], we reduce a union construction expression “ $l\{e\}$ ” by first reducing e , until it becomes a value.
- By rule [R-UnionCons-Res], a union constructor “ $l\{v\}$ ” (with a label followed by a value) reduces by storing the label l and value v on the heap, and returning the memory address where their data is located. (This is similar to rule [R-Struct-Res] in [Definition 29](#).) More in detail, the premises of rule [R-UnionCons-Res] say that:
 - h is the current heap, taken from the runtime environment R (i.e. $R.\text{Heap}$);
 - p is the memory address of the first location after the maximum address currently used in h ; (e.g. if h assigns values to addresses between $0x00000001$ to $0x000000a8$, then p will be $0x000000a9$)

- h' is an updated heap that is equal to h , except that:
 - * the memory location p contains the label l (here represented as a string "l");
 - * the memory location $p + 1$ contains the value v .

Notice that the memory addresses p and $p + 1$ were not used in the original heap h , but are now being used in h' ;

- R' is an updated runtime environment that is equal to R , except that $R'.\text{Heap}$ is h' .
- By rule [R-Match-Eval], we reduce a pattern matching expression “match e with $\{\dots\}$ ” by first reducing e , until it becomes a value.
 - By rule [R-Match-Res], we reduce a pattern matching expression “match p with $\{\dots\}$ ” (where p is a memory pointer) as follows:
 1. we check whether the heap location p contains a label “ l_k ”, for some $k \in i..n$ (i.e. the label stored on the heap must be equal to one of those appearing in the pattern matching cases);
 2. if so, we take the value v stored at the heap location $p + 1$; and
 3. we compute the pattern matching reduction by taking the the match continuation expression e_k and substituting each occurrence of the matched variable x_k with the value v .

Example 56 (Reductions of Structure Construction and Assignment)

Consider the following expression, which constructs a discriminated union instance and updates one of its fields.

```
let x : t = Some {42};
match x with {
  None{x} → ()
  Some{y} → println(y)
}
```

In the first reduction below, the discriminated union constructor “Some {42}” reduces by storing Some and 42 on consecutive heap addresses, and returning the base address 0x0001.

$$\frac{h = R.\text{Heap} = \emptyset \quad \max\text{Addr}(h) + 1 = 0x0001 \quad h' = \left\{ \begin{array}{l} 0x0001 \mapsto \text{"Some"} \\ 0x0002 \mapsto 42 \end{array} \right\} \quad R' = \{R \text{ with Heap} = h'\}}{\langle R \bullet \text{Some} \{42\} \rangle \rightarrow \langle R' \bullet 0x0001 \rangle} \quad [\text{R-UnionCons-Res}]$$

$$\langle R \bullet \left(\begin{array}{l} \text{let } x : t = \text{Some} \{42\}; \\ \text{match } x \text{ with } \{ \\ \quad \text{None}\{x\} \rightarrow () \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right) \rangle \rightarrow \langle R' \bullet \left(\begin{array}{l} \text{let } x : t = 0x0001; \\ \text{match } x \text{ with } \{ \\ \quad \text{None}\{x\} \rightarrow () \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right) \rangle \quad [\text{R-Let-Eval-Init}]$$

In the second reduction below, we substitute each occurrence of x with the value (pointer) $0x0001$.

$$\frac{}{\langle R' \bullet \left\langle \begin{array}{l} \text{let } x : t = 0x0001; \\ \text{match } x \text{ with } \{ \\ \quad \text{None}\{x\} \rightarrow () \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right\rangle \rightarrow \langle R' \bullet \left\langle \begin{array}{l} \text{match } 0x0001 \text{ with } \{ \\ \quad \text{None}\{x\} \rightarrow () \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right\rangle} \text{[R-Let-Subst]}$$

In the third reduction, rule [R-Match-Res] checks that the heap location $0x0001$ contains "Some", which corresponds to the second pattern matching case $\text{Some}\{y\} \rightarrow \dots$; therefore, the rule takes the corresponding continuation expression $\text{println}(y)$, and substitutes each occurrence of the matched variable y with the value 42 (found on the heap in the location that follows $0x0001$).

$$\frac{R'.\text{Heap}(0x0001) = \text{"Some"} \quad R'.\text{Heap}(0x0002) = 42}{\langle R' \bullet \left\langle \begin{array}{l} \text{match } 0x0001 \text{ with } \{ \\ \quad \text{None}\{x\} \rightarrow () \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right\rangle \rightarrow \langle R' \bullet \text{println}(42) \rangle} \text{[R-Match-Res]}$$

Then, the expression $\text{println}(42)$ continues reducing according to the usual semantics.

10.1.4 Free and Captured Variables

As mentioned when discussing variable substitution (*Definition 41* above), the cases of the pattern matching expression *bind* the matched variable. Correspondingly, the definition of free and captured variables (*Definition 43* and *Definition 44* below) exclude that matched variable from their results.

Definition 43 (Free Variables of Union Constructors and Pattern Matching)

We extend *Definition 34* (free variables) with the following new cases:

$$\begin{aligned} \text{fv}(l\{e\}) &= \text{fv}(e) \\ \text{fv}(\text{match } e \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots; l_n\{x_n\} \rightarrow e_n\}) &= \text{fv}(e) \cup \bigcup_{i \in 1..n} (\text{fv}(e_i) \setminus \{x_i\}) \end{aligned}$$

Example 57

Consider the following Hygge expression:

```

let x : t = Some {42};
match x with {
  None{y} → z + 1
  Some{y} → println(y)
}
    
```

By *Definition 43*, the free variables of this expression are:

$$\begin{aligned}
 \text{fv} \left(\begin{array}{l} \text{let } x : t = \text{Some } \{42\}; \\ \text{match } x \text{ with } \{ \\ \quad \text{None}\{y\} \rightarrow z + 1 \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right) &= \text{fv}(\text{Some } \{42\}) \cup \left(\text{fv} \left(\begin{array}{l} \text{match } x \text{ with } \{ \\ \quad \text{None}\{y\} \rightarrow z + 1 \\ \quad \text{Some}\{y\} \rightarrow \text{println}(y) \\ \} \end{array} \right) \setminus \{x\} \right) \\
 &= \emptyset \cup \left(\left(\text{fv}(x) \cup \left((\text{fv}(z + 1) \setminus \{y\}) \cup (\text{fv}(\text{println}(y)) \setminus \{y\}) \right) \right) \setminus \{x\} \right) \\
 &= \left(\text{fv}(x) \cup \left((\{z\} \setminus \{y\}) \cup (\{y\} \setminus \{y\}) \right) \right) \setminus \{x\} \\
 &= (\{x\} \cup (\{z\} \cup \emptyset)) \setminus \{x\} \\
 &= \{x, z\} \setminus \{x\} \\
 &= \{z\}
 \end{aligned}$$

Definition 44 (Captured Variables of Union Constructors and Pattern Matching)

We extend *Definition 35* (captured variables) with the following new cases:

$$\begin{aligned}
 \text{cv}(l\{e\}) &= \text{cv}(e) \\
 \text{cv}(\text{match } e \text{ with } \{l_1\{x_1\} \rightarrow e_1; \dots; l_n\{x_n\} \rightarrow e_n\}) &= \bigcup_{i \in 1..n} (\text{cv}(e_i) \setminus \{x_i\})
 \end{aligned}$$

10.1.5 Typing Rules

In order to type-check programs that use discriminated unions and pattern matching, we need to introduce a new discriminated union type (*Definition 45*), a new rule for pretype resolution (*Definition 46*), a new subtyping rule (*Definition 47*), and some new typing rules (*Definition 48*).

Definition 45 (Discriminated Union Type)

We extend the Hygge typing system with a **discriminated union type** by adding the following case to *Definition 30*:

$$\begin{array}{l}
 \text{Type } T ::= \dots \\
 \quad | \quad \text{union } \{l_1 : T_1; \dots; l_n : T_n\} \quad \left(\begin{array}{l} \text{Discriminated union type, with } n \geq 1 \\ \text{and } l_1, \dots, l_n \text{ pairwise distinct} \end{array} \right)
 \end{array}$$

By [Definition 45](#) above, a discriminated union type describes various cases where a label l_i (for $i \in 1..n$, with $n \geq 1$) tags a type T_i . Note that the labels names must be distinct from each other.

Example 58 (Discriminated Union Types)

The following type describes a union type with two cases: a with type `int`, and b with type `bool`.

$$\text{struct } \{a : \text{int}; b : \text{bool}\}$$

The following type describes a union type with two cases: c with type `int`, and d with structure type, which in turn has a field a of type `float` and a field b of type `bool`.

$$\text{struct } \{c : \text{int}; d : \text{struct } \{a : \text{float}; b : \text{bool}\}\}$$

We also need a way to resolve a syntactic union pretype (from [Definition 40](#)) into a valid union type (from [Definition 45](#)): this is formalised in [Definition 46](#) below.

Definition 46 (Resolution of Discriminated Union Types)

We extend [Definition 31](#) (type resolution judgement) with this new rule:

$$\frac{l_1, \dots, l_n \text{ pairwise distinct} \quad \forall i \in 1..n : \Gamma \vdash t_i \triangleright T_i}{\Gamma \vdash \text{union } \{l_1 : t_1, \dots, l_n : t_n\} \triangleright \text{union } \{l_1 : T_1, \dots, l_n : T_n\}} \text{ [TRes-Union]}$$

According to rule [TRes-Union] in [Definition 46](#) above, we resolve a discriminated union pretype by ensuring that all labels l_i are distinct from each other, and all types T_i used in the union can be resolved.

We also introduce a new subtyping rule for discriminated union types, according to [Definition 47](#) below. This extension adds flexibility to the typing system, and is also necessary to type-check union constructors, as we will see later in [Example 60](#). Notice that, without [Definition 47](#), the subtyping for union types would only be allowed by rule [TSub-Refl] in [Definition 10](#), which only relates types that are exactly equal to each other.

Definition 47 (Subtyping for Discriminated Union Types)

We define the subtyping of Hygge with discriminated union types by extending [Definition 32](#) with the following new rule:

$$\frac{\forall i \in 1..m : \exists j \in 1..n : l_i = l'_j \text{ and } \Gamma \vdash T_i \leq T'_j}{\Gamma \vdash \text{union } \{l_1 : T_1; \dots, l_m : T_m\} \leq \text{union } \{l'_1 : T'_1; \dots, l'_n : T'_n\}} \text{ [TSub-Union]}$$

According to rule [TSub-Union] in [Definition 47](#) above:

- for each label l_i in the subtype, there must be an equal label l'_j in the supertype (but the supertype can have more labels that do not appear in the subtype); and
- when two labels l_i and l'_j are equal, then the type T_i must be subtype of T'_j .

Important: According to *Definition 47*, the order of the labels in the union subtype and supertype is not important.

This is unlike subtyping for structures (*Definition 32*), where the order of the fields in the subtype must match the order of fields in the supertype.

Example 59

Consider the following union type:

$$\text{union } \{A : \text{int}; B : \text{bool}\}$$

Consider any well-typed Hygge program that uses an instance of such a union type: intuitively, that program can only access the content that instance via pattern matching, obtaining either case A (with an integer) or case B (with a boolean).

Therefore, that Hygge program will also work correctly if it operates on an instance of the following union type:

$$\text{union } \{A : \text{int}\}$$

The reason is that the program will still support case A, while case B will not be used. For this reason, *Definition 47* considers the second union type as a subtype of the first.

Definition 48 (Typing Rules for Union Constructors and Pattern Matching)

We define the **typing rules of Hygge with discriminated union constructors and pattern matching** by extending *Definition 33* with the following rules (which use the union type introduced in *Definition 45* above):

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash l \{e\} : \text{union } \{l : T\}} \text{ [T-UnionCons]}$$

$$\frac{\Gamma \vdash e : \text{union } \{l'_1 : T_1; \dots; l'_m : T_m\} \quad \begin{array}{l} l_1 \dots, l_n \text{ pairwise distinct} \\ \forall i \in 1..n : \exists j \in 1..m : l_i = l'_j \text{ and } \{\Gamma \text{ with Env} + (x_i \mapsto T_j)\} \vdash e_i : T \end{array}}{\Gamma \vdash \text{match } e \text{ with } \{l_1 \{x_1\} \rightarrow e_1; \dots; l_n \{x_n\} \rightarrow e_n\} : T} \text{ [T-Match]}$$

The typing rules in *Definition 48* above work as follows.

- By rule [T-UnionCons], a discriminated union constructor $l \{e\}$ has a union type with just one case: the label l , and the type T that type-checks the expression e .
- By rule [T-Match], a pattern matching expression $\text{match } e \text{ with } \{\dots\}$ has type T only if:

1. the labels l_1, \dots, l_n used in the pattern matching expression are distinct from each other;
2. the expression e type-checks, and it has a union type with labels l'_i, \dots, l'_m ;
3. each label l_i used in the pattern matching expression is equal to some label l'_j in the union type of e ;
4. each continuation expression e_i of the pattern matching cases can be type-checked by extending the typing environment Γ with a new entry, where the matched variable x_i has the corresponding type T_j ;
5. under such extended Γ , each continuation expression e_i has the same type T (which is also the type of the whole pattern matching expression).

Example 60 (Typing Derivation of Unions and Pattern Matching)

Consider the following Hygge expression:

```

let x : t = Some {42};
match x with {
  None{x}  → ()
  Some{y}  → print(y)
}

```

Using the typing rules in [Definition 48](#), the expression above can be type-checked with the following typing derivation — where:

- the typing environment Γ' is equal to Γ , except that we have $\Gamma(x) = \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\}$;
- the typing environment Γ'' is equal to Γ' , except that we have $\Gamma''(x) = \text{unit}$;
- the typing environment Γ''' is equal to Γ' , except that we have $\Gamma'''(y) = \text{int}$.

$$\begin{array}{c}
\frac{\Gamma \vdash \text{"int"} \triangleright \text{int} \quad [\text{TRes-Int}] \quad \Gamma \vdash \text{"unit"} \triangleright \text{unit} \quad [\text{TRes-Unit}]}{\Gamma \vdash \text{"union"} \left\{ \begin{array}{l} \text{Some} : \text{int}; \\ \text{None} : \text{unit} \end{array} \right\} \triangleright \text{union} \left\{ \begin{array}{l} \text{Some} : \text{int}; \\ \text{None} : \text{unit} \end{array} \right\} \quad [\text{TRes-Union}]} \\
\frac{\Gamma \vdash 42 : \text{int} \quad [\text{TVal-Int}] \quad \Gamma \vdash \text{Some}\{42\} : \text{union}\{\text{Some} : \text{int}\} \quad [\text{TUnionCons}]}{\Gamma \vdash \text{Some}\{42\} : \text{union}\{\text{Some} : \text{int}\}} \\
\frac{\Gamma \vdash \text{int} \leq \text{int} \quad [\text{TSub-Ref}] \quad \Gamma \vdash \text{union}\{\text{Some} : \text{int}\} \leq \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\} \quad [\text{TSub-Union}]}{\Gamma \vdash \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\} \quad [\text{TSub}]} \\
\frac{\Gamma'(x) = \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\} \quad [\text{TVar}]}{\Gamma' \vdash x : \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\}} \text{weccan} \\
\frac{}{\Gamma'' \vdash () : \text{unit}} \quad [\text{TVal-Unit}] \quad \frac{\Gamma''(y) = \text{int} \quad [\text{T-Var}]}{\Gamma'' \vdash y : \text{int}} \quad [\text{T-Print}] \\
\frac{}{\Gamma''' \vdash \text{print}(y) : \text{unit}} \quad [\text{T-Match}]}
{\Gamma \vdash \text{let } x : \text{union}\{\text{Some} : \text{int}; \text{None} : \text{unit}\} = \text{Some}\{42\}; \\ \text{match } x \text{ with } \{ \\ \text{None}\{x\} \rightarrow () \\ \text{Some}\{y\} \rightarrow \text{print}(y) \\ \} : \text{unit}} \quad [\text{T-Let}]
\end{array}$$

In the derivation above, we can see that:

- we use the new subtyping rule [\[TSub-Union\]](#) ([Definition 47](#)) to initialise variable x (whose union type has the two case labels “Some” and “None”) with the union constructor $\text{Some}\{42\}$ (whose union type only has the label “Some”);
- to type-check the pattern matching on x , we type-check the continuation expression $\text{print}(y)$ with a typing environment Γ''' , where the variable y (bound in the pattern matching case for the label “Some”) has type int — which in turn is taken from the case “Some” in the union type of x .

10.1.6 Implementation

We now have a look at how `hygdec` is extended to implement discriminated unions and pattern matching, according to the specification illustrated in the previous sections.

Tip: To see a summary of the changes described below, you can inspect the differences in the *hygdec Git repository* between the tags `free-captured-vars` and `union-rec-types`.

Important: Code generation for discriminated unions and pattern matching is not implemented! This is one of the *Project Ideas* for this Module.

10.1.7 Lexer, Parser, Interpreter, and Type Checking

These parts of the `hygdec` compiler are extended along the lines of *Example: Extending Hygge0 and hygdec with a Subtraction Operator*.

- We extend `AST.fs` in two ways, according to *Definition 40*:
 - we extend the data type `Expr<'E, 'T>` with two new cases:
 - * `UnionCons` for the discriminated union constructor “`l{e}`”;
 - * `Match` for the pattern matching expression “`match e with {l{x} → e'; ...}`”;
 - we also extend the data type `Pretype` with a new case called `TUnion`, corresponding to the new union pretype “`union {l1 : t1, ..., ln : tn}`”:

```

and Pretype =
// ...
/// Discriminated union type. Each case consists of a name and a
↳pretype.
| TUnion of cases: List<string * PretypeNode>
```

- We extend `PrettyPrinter.fs` to support the new expressions and pretype.
- We extend `Lexer.fsl` to support three new tokens:
 - `UNION` for the new keyword “`union`”, and
 - `MATCH` for the new keyword “`match`”, and
 - `WITH` for the new keyword “`with`”.
- We extend `Parser.fsy` to recognise the desired sequences of tokens according to *Definition 40*, and generate AST nodes for the new expressions. We proceed by adding:

- a new rule under the pretype category to generate TUnion pretype instances (this rule reuses the fieldTypeSeq category to parse the union type cases, since they are syntactically similar to structure fields);
- a new rule under the primary category to generate UnionCons instances;
- a new rule under the simpleExpr category to generate Match instances
- a new auxiliary parsing rule matchCases that recognises a non-empty sequence of pattern matching cases “ $l\{x\} \rightarrow e$ ”.
- We extend ASTUtil.fs as follows:
 - we extend the function subst in to support the new expressions UnionCons and Match, according to *Definition 41*;
 - we extend the function freeVars according to *Definition 43*;
 - we extend the function capturedVars according to *Definition 44*.
- We extend Interpreter.fs according to *Definition 42*. In the function reduce:
 - we add a new case for UnionCons, and
 - we add a new case for Match.
- We extend Type.fs by adding a new case to the data type Type, according to *Definition 45*: the new case is called TUnion. We also add a corresponding new case to the function freeTypeVars in the same file.

Note: Correspondingly, we also extend the pretty-printing function formatType in PrettyPrinter.fs, to display the structure type we have just introduced.

- We extend Typechecker.fs:
 - we extend the type resolution function resolvePretype with a new case for discriminated union types, according to *Definition 46*;
 - we extend the function isSubtypeOf with a new case for union types, according to *Definition 47*;
 - we extend the function typer according to *Definition 48*, to support the new cases for the expressions UnionCons and Match.
- As usual, we add new tests for all compiler phases.

10.2 Recursive Types

We now study how to extend Hygge with **recursive type definitions**, allowing us to define data structures like lists and trees. We discuss the *Overall Objective*, and then the changes that are necessary for *extensions to hyggec* and *Extending Subtyping to Support Recursive Types*. Then we briefly discuss their *implementation*.

10.2.1 Overall Objective

Our goal is to fully support Hygge programs like the one illustrated in *Example 61* below.

Example 61 (A Hygge Program with Recursive Types)

```

1 // A tree type, where nodes can have zero, one, or two children.
2 type Tree = union {
3   Leaf: int;
4   Node1: struct {value: int; child: Tree};
5   Node2: struct {value: int; child1: Tree; child2: Tree}
6 };
7
8 // A (non-empty) list, where each node can have zero or one child.
9 type List = union {
10  Leaf: int;
11  Node1: struct {value: int; child: List}
12 };
13
14 // Check whether the given tree only has one element.
15 fun hasSize1(t: Tree): bool =
16   match t with {
17     Leaf{ } -> true;
18     Node1{ } -> false;
19     Node2{ } -> false
20   };
21
22 // Here, 't' can be given type Tree or List (both type-check)
23 let t: Tree = Node1{struct{value = 1;
24                       child = Node1{struct{value = 2;
25                                       child = Leaf{3}}}}};
26
27 // If 't' above has type List, we get a stack overflow during type checking!
28 assert(not hasSize1(t))

```

The program above can be interpreted, type-checked, compiled and executed correctly. However, if we modify the type annotation on line 23 from `Tree` to `List`, then the `hyggec` compiler crashes during type-checking: the function `isSubtypeOf` enters in an infinite loop that causes a stack overflow. (Try it!)

To support a program like the one in *Example 61*, we need two changes to Hygge and hyggec:

1. *Extending Type Definitions to Support Recursive Types* (NOTE: this is already implemented in the hyggec version presented in this module); and
2. *Extending Subtyping to Support Recursive Types* (which is **not** implemented, and is one of the *Project Ideas* for this Module).

10.2.2 Extending Type Definitions to Support Recursive Types

According to the typing rule [T-Type] of Hygge0 (*Definition 8*), a type definition type $x = t$; e does not allow the defined type t to mention the type variable x : this prevents definition of recursive types like `Tree` and `List` in *Example 61* above. To solve this issue, we extend the typing system as shown in *Definition 49* below.

Definition 49 (Type System with Recursive Type Definitions)

We define the Hygge typing system with recursive type definitions by replacing rule [T-Type] in *Definition 8* with the following rule:

$$\frac{\begin{array}{c} x \notin \{\text{bool}, \text{int}, \text{float}, \text{string}, \text{unit}\} \quad x \notin \Gamma.\text{TypeVars} \\ \{\Gamma \text{ with TypeVars} + (x \mapsto \text{unit})\} \vdash t \triangleright T \quad \{\Gamma \text{ with TypeVars} + (x \mapsto T)\} \vdash e : T' \quad x \notin T' \end{array}}{\Gamma \vdash \text{type } x = t; e : T'} \quad [\text{T-Type-Rec}]$$

There is only one difference between rule [T-Type-Rec] in *Definition 49* and [T-Type] in *Definition 8*: in order to resolve the pretype t , the premise of rule [T-Type-Rec] uses a typing environment extended by assigning a “dummy” type to x (in this case `unit` – but any other type would work). This way, if t resolves to type T , then T can refer to the type variable x .

Exercise 37

To see why rule [T-Type-Rec] in *Definition 49* resolves pretypes in an extended environment with a “dummy” entry, consider the following recursive pretype:

$$\text{union } \{a : \text{int}; b : \text{List}\}$$

Try to resolve the pretype above into an actual type, using *Definition 46* with the following typing environments:

- first, by using $\Gamma.\text{TypeVars} = \emptyset$ (i.e. no type variables are defined);
 - then, by using $\Gamma.\text{TypeVars} = \{\text{List} \mapsto \text{unit}\}$ (i.e. a “dummy” entry for the type variable `List` is defined).
-

10.2.3 Extending Subtyping to Support Recursive Types

Before extending the subtyping of Hygge, we illustrate its limitations in [Example 62](#) below.

Example 62 (Why We Need to Extend Subtyping to Handle Recursive Types)

Consider again the program in [Example 61](#), changing the type annotation of `t` on line 23 from `Tree` to `List`. In this case, on line 28, the application of function `hasSize1` needs to check whether `List` (the type of the argument `t`) is subtype of the expected argument type `Tree`. This creates the derivation below, where we use the typing environment:

$$\Gamma.\text{TypeVars} = \left\{ \begin{array}{l} \text{Tree} \mapsto \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{Tree} \end{array} \right\}; \\ \text{Node2} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child1} : \text{Tree}; \\ \text{child2} : \text{Tree} \end{array} \right\} \end{array} \right\} \\ \text{List} \mapsto \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

and the subtyping rules in [Definition 10](#), [Definition 32](#), and [Definition 47](#).

$$\frac{\frac{\frac{\Gamma \vdash \text{int} \leq \text{int} \quad [\text{TSub-Refl}]}{\Gamma \vdash \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \leq \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{Tree} \end{array} \right\}} \quad [\text{TSub-Struct}]}{\Gamma \vdash \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\} \leq \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{Tree} \end{array} \right\}; \\ \text{Node2} : \dots \end{array} \right\}} \quad [\text{TSub-Union}]}{\Gamma \vdash \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\} \leq \text{Tree}} \quad [\text{TSub-Var-R}]}{\Gamma \vdash \text{List} \leq \text{Tree}} \quad [\text{TSub-Var-L}]$$

Observe that, going bottom-up towards the the top-right branch, the derivation reaches a point where it is trying to prove that `List` is subtype of `Tree` – which is the starting point of the derivation! Therefore, we can extend this derivation infinitely, by repeating the rules above. The `hyggec` function `isSubtypingOf` tries to do precisely this, and thus, it enters in an infinite loop.

Intuitively, an infinite derivation like the one shown in [Example 62](#) suggests that, indeed, `List` should be considered a subtype `Tree`. However, until now we have only considered *finite* derivations – and the implementation of `isSubtypingOf` reflects this assumption and does not handle recursive types correctly.

To resolve the issue illustrated in [Example 62](#) above, we need to detect whether a subtyping derivation is infinite — and if this happens, we want to conclude that the subtyping judgement holds. We obtain this by revising the subtyping judgement as formalised in [Definition 50](#) below.

Definition 50 (Recursive Subtyping in Hygge)

To extend Hygge with support for **recursive subtyping**, we replace the [Definition 10](#) (subtyping judgement) by introducing a new subtyping judgement which has the same form:

$$\Gamma \vdash T \leq T'$$

However, the new subtyping judgement is defined with just one rule:

$$\frac{\emptyset, \Gamma \vdash T \leq T'}{\Gamma \vdash T \leq T'} \text{ [TSub]}$$

In rule [TSub] above, the premise uses the **subtyping judgement with assumptions**, which has the following form:

$$A, \Gamma \vdash T \leq T'$$

where A is a **set of subtyping assumptions**, i.e. a set of pairs of types (T, T') , meaning “we assume that T is subtype of T' ”.

The subtyping judgement with assumptions is defined by the following rules:

$$\begin{array}{c} \frac{}{A, \Gamma \vdash T \leq T} \text{ [TSubA-Refl]} \qquad \frac{(T, T') \in A}{A, \Gamma \vdash T \leq T'} \text{ [TSubA-Assume]} \\ \\ \frac{A \cup \{(x, T)\}, \Gamma \vdash \Gamma.\text{TypeVars}(x) \leq T}{A, \Gamma \vdash x \leq T} \text{ [TSubA-Var-L]} \qquad \frac{A \cup \{(T, x)\}, \Gamma \vdash T \leq \Gamma.\text{TypeVars}(x)}{A, \Gamma \vdash T \leq x} \text{ [TSubA-Var-R]} \\ \\ \frac{m \geq n \quad \forall i \in 1..n : A, \Gamma \vdash T_i \leq T'_i}{A, \Gamma \vdash \text{struct } \{f_1 : T_1; \dots, f_m : T_m\} \leq \text{struct } \{f_1 : T'_1; \dots, f_n : T'_n\}} \text{ [TSubA-Struct]} \\ \\ \frac{\forall i \in 1..m : \exists j \in 1..n : l_i = l'_j \text{ and } A, \Gamma \vdash T_i \leq T'_j}{A, \Gamma \vdash \text{union } \{l_1 : T_1; \dots, l_m : T_m\} \leq \text{union } \{l'_1 : T'_1; \dots, l'_n : T'_n\}} \text{ [TSubA-Union]} \end{array}$$

Observe that in [Definition 50](#) above, the subtyping rules with assumptions [TSubA-Struct] and [TSubA-Union] are identical (respectively) to the subtyping rules [TSub-Struct] (from [Definition 32](#)) and [TSub-Union] (from [Definition 47](#)) — except that the new rules recursively propagate the set of assumptions A between the conclusion and premises of the rule. Such assumptions are pairs of types (T, T') and represent the “belief” that T is subtype of T' . The subtyping assumptions are used as follows:

- new assumptions are introduced in the premises of rule [TSubA-Var-L] and [TSubA-Var-R]. When we build a derivation bottom-up, these assumptions “remember” that the derivation is trying to prove that T is a subtyping of T' (where at least one between T and T' is a type variable);

- the assumptions are then used in rule [TSubA-Assume]: to prove that T is subtype of T' , we can use rule [TSubA-Assume] if the pair (T, T') is in the set of assumptions A .

The effect of these new subtyping rules with assumptions can be seen in *Example 63* below.

Example 63 (Recursive Subtyping in Action)

Let us revise *Example 62* using the new subtyping rules with assumptions in *Definition 50*. We use the same typing environment Γ ; for brevity, let us define the following alias:

$$T = \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\}$$

Then, we prove that *List* is subtype of *Tree* with the following derivation:

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\text{[TSubA-Refl] } \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}, \Gamma \vdash \text{int} \leq \text{int}}{\text{[TSubA-Ref] } \frac{\frac{\text{[TSubA-Assume] } (List, Tree) \in \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}}{\frac{\text{[TSubA-Struct] } \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}, \Gamma \vdash List \leq Tree}}{\frac{\text{[TSubA-Union] } \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}, \Gamma \vdash \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \leq \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{Tree} \end{array} \right\}}{\frac{\text{[TSubA-Var-R] } \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}, \Gamma \vdash \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\} \leq \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{Tree} \end{array} \right\}; \\ \text{Node2} : \dots \end{array} \right\}}{\frac{\text{[TSubA-Var-L] } \left\{ \begin{array}{l} (\text{List}, \text{Tree}) \\ (T, \text{Tree}) \end{array} \right\}, \Gamma \vdash \text{union} \left\{ \begin{array}{l} \text{Leaf} : \text{int}; \\ \text{Node1} : \text{struct} \left\{ \begin{array}{l} \text{value} : \text{int}; \\ \text{child} : \text{List} \end{array} \right\} \end{array} \right\} \leq Tree}}{\frac{\text{[TSub] } \emptyset, \Gamma \vdash List \leq Tree}}{\Gamma \vdash List \leq Tree}} \text{[TSub]}$$

Observe that, going bottom-up, the derivation propagates the assumption $(List, Tree)$ from its root across all branches — and when the derivation reaches the top-right branch, rule [TSubA-Assume] terminates the derivation (instead of continuing infinitely). As a consequence, we are now able to prove that the recursive type *List* is subtype of the recursive type *Tree*, using a finite derivation.

10.2.4 Implementation of Recursive Types and Subtyping

The improved typing rule presented in *Definition 49* is already implemented in the function typer (case for expression `Type(...)`). To see what has been changed, you can inspect the differences in the *hygge* Git repository between the tags `free-captured-vals` and `union-rec-types`.

Instead, the recursive subtyping presented in *Definition 50* is **not** yet implemented: it is one of the *Project Ideas* for this Module.

10.3 References and Further Readings

The revised subtyping judgement with support for *Recursive Types*, and the resulting function to check whether T is subtype of T' , are based on the following seminal paper.

- Roberto M. Amadio and Luca Cardelli. 1993. *Subtyping Recursive Types*. ACM Transactions on Programming Languages and Systems, 15, 4 (September 1993). <https://doi.org/10.1145/155183.155231>

The subtyping algorithm that derives from the treatment above has exponential complexity w.r.t. the size of the types being checked. However, the algorithm can be optimised to achieve quadratic complexity (at the price of a slight complication). This topic is addressed in the following reference, together with a comprehensive treatment of recursive types and coinduction.

- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002. Available on DTU Findit³⁹. These chapters, in particular, may be useful:
 - Chapter 20 (Recursive Types)
 - Chapter 21 (Metatheory of Recursive Types) – in particular, Sections 21.9 and 21.10 discuss how to optimise the recursive subtyping algorithm

10.4 Project Ideas

For your group project, you should implement at least 2 of the following project ideas (listed in order of increasing difficulty).

- *Project Idea: Exhaustive Pattern Matching*
- *Project Idea: Implement Code Generation for Union Type Constructors and Pattern Matching*
- *Project Idea: Implement Recursive Subtyping*
- *Project Idea: Better Inference of Pattern Matching Result Type*

10.4.1 Project Idea: Exhaustive Pattern Matching

The goal of this project idea is to make the type checking of pattern matching expressions more strict, by requiring them to cover *all* possible labels allowed for the variable being matched.

For example, consider the following Hygge program:

```
1 // An optional integer value.  
2 type OptionalInt = union {  
3   Some: int;
```

(continues on next page)

³⁹ <https://findit.dtu.dk/en/catalog/5c34980ed9001d2f3820637f>

(continued from previous page)

```

4   None: unit
5   };
6
7   fun displayOption(o: OptionalInt): unit =
8     match o with {
9       // Some{x} -> println(x);
10      None{ _ } -> println("None")
11    };
12
13  displayOption(Some{42})

```

This program type-checks – but when interpreted, it gets stuck on the assertion on line 13: this happens because the pattern matching on lines 8–11 does not handle the case where `o` is `None`.

If you choose this project idea, you will need to:

1. explain how to adjust the typing rule [T-Match] in *Definition 48* to require that *all* labels in the type of the matched expression e are covered in the pattern matching cases;
2. correspondingly extend the case for `Match(...)` of the function typer (in the file `Typechecker.fs`), reporting an error if any pattern matching case is missing; and
3. as usual, include tests showcasing your extension to Hygge and hyggec – including the example program above.

Note: If you choose this Project Idea, you might need to revise some existing hyggec tests, that might not type-check anymore because they include a non-exhaustive pattern matching expression.

10.4.2 Project Idea: Implement Code Generation for Union Type Constructors and Pattern Matching

The code generation for discriminated union types constructors and pattern matching is not implemented. The goal of this Project Idea is to implement it, and cover (at least) code generation for all well-typed tests that are supported by the hyggec interpreter, adding more tests if necessary.

Note: If a pattern matching expression is applied to a union instance $l\{v\}$, but the label l is not covered in any of the pattern matching cases, then generated code should cause a run-time failure (similar to an assertion violation). To avoid this situation, you might consider to also choose the *Project Idea on exhaustive pattern matching*.

Hint:

- Since union type instances are stored on the heap, you may find some inspiration in the cases `Struct` and `FieldAccess` of the function `doCodegen` (in the file `RISCVCodegen.fs`).
- The semantic rules in [Definition 42](#) store a discriminated union instance $l\{v\}$ on the heap by saving a string "l" to represent the label l . This is a handy abstraction for the interpreter – but it may be quite cumbersome to implement as-is for code generation, and also inefficient: when performing a pattern matching, to see whether a label on the heap matches a label in a pattern matching case, the generated code would need to compare two strings, character by character.

A simpler and more efficient approach is to:

1. use a distinct integer value to represent each union label l appearing in a Hygge program;
2. save that integer value on the heap (when union instances are created) to represent the label l ; and
3. when performing pattern matching against a label l , check whether the integer value found on the heap is equal to the integer value that represents a label l .

To achieve this, the function `genSymbolId` (in the file `Util.fs`) may be useful...

10.4.3 Project Idea: Implement Recursive Subtyping

The goal of this project idea is to update the function `isSubtypeOf` (in the file `Typechecker.fs`) to implement the new subtyping judgement in [Definition 50](#), using subtyping assumptions.

If you choose this project idea, you should explain how you designed and implemented the revised `isSubtypeOf` function, and include some tests to showcase your extension (in particular, you should be able to type-check the program in [Example 62](#) when the type ascription on line 23 is changed to `List`).

Note: If you choose this project idea, together with *better type inference for pattern matching* and *recursive functions*, you should be able to type-check the complex Hygge example shown at the very beginning of this course, in *A Taste of Hygge*.

10.4.4 Project Idea: Better Inference of Pattern Matching Result Type

The current implementation of type checking for pattern matching expressions follows *Definition 48* in a rather restrictive way. For example, consider the following Hygge program:

```

1 // An optional integer value.
2 type OptionalInt = union {
3   Some: int;
4   None: unit
5 };
6
7 fun maybeIncrement(o: OptionalInt): OptionalInt =
8   match o with {
9     Some{x} -> Some{x + 1};
10    None{ _ } -> None{()}
11  };
12
13 let x: OptionalInt = maybeIncrement(Some{41});
14
15 assert(match x with {
16   Some{y} -> y = 42;
17   None{ _ } -> false
18 })

```

The program can be interpreted correctly, but type checking fails with the following error:

```

(10:20-10:27): pattern match result type mismatch: expected union {Some: int},
↳ found union {None: unit}

```

This is because the type T of the continuation expression of the *first* pattern matching case is used as result type for the whole pattern matching expression — and if some other case has a type T' that is *not* a subtype of T , then type checking fails. Therefore, in the example above:

- the continuation expression of the first pattern matching case (line 9) has type $T = \text{union } \{Some : \text{int}\}$;
- the continuation expression of the second pattern matching case (line 10) has type $T' = \text{union } \{None : \text{unit}\}$;
- T' is not subtype of T , hence the type checking error.

A better approach would be to:

- compute the types T_1, \dots, T_n of all continuation expression of all pattern matching cases;
- compute (if possible) a type T that is the “most precise” supertype of all types T_1, \dots, T_n ; and

- use T as result type of the whole pattern matching expression.

Such a common supertype T is called the **least upper bound (LUB)** of T_1, \dots, T_n .

To compute the LUB of two types T and T' , you can define a recursive function that works as follows:

- if T is subtype of T' , their LUB is T' ;
- if T' is subtype of T , their LUB is T ;
- if T and T' are both union types, their LUB is a union type T'' that contains:
 - all the case labels that appear in T or T' ; moreover,
 - if a case label l appears in both T (as “ $l : T_1$ ”) and T' (as “ $l : T_2$ ”), then the label l must also appear in T'' – and its type should be the LUB between T_1 and T_2 ;
- if T and T' are both structure types, then... (What is their LUB? What conditions should T and T' satisfy for their LUB to exist?)

To compute the LUB of three or more types T_1, T_2, T_3, \dots , you can simply compute $\text{lub}(T_1, \text{lub}(T_2, \text{lub}(T_3, \dots)))$.

If the LUB of two types cannot be computed (e.g. the LUB of int and string does not exist), then the compiler should report an error.

If you choose this project idea, you should explain how you designed and implemented the LUB function, and include some tests to showcase your extension (including the example above, and ideally also covering structure types).

Note: The same LUB-based improvement can be applied to the inference of the result type of “if” expressions. You could even consider starting with “if” expressions (which are simpler) and then extending your improvement to pattern matching.

Note: If you choose this project idea, together with *implementing recursive subtyping* and *recursive functions*, you should be able to type-check the complex Hygge example shown at the very beginning of this course, in *A Taste of Hygge*.

11

Module 11: Intermediate Representations and Register Allocation

In this module we discuss the concept of **intermediate representation** in a compiler, focusing on a specific example called **Administrative Normal Form (ANF)**. We then explore its application in the `hyggec` compiler, with the objective of using ANF to improve the **code generation and register allocation strategy** of `hyggec`.

11.1 Overall Objective

Our goal is to compile and run Hygge programs like the one shown in [Example 64](#) below.

Example 64 (A Hygge Program Using (Too) Many Registers)

Consider the following Hygge program:

```
1 let res: int = 1 + (2 + (3 + (4 + (5 + (6 + (7 + (8
2           + (9 + (10 + (11 + (12 + (13 + (14
3           + (15 + (16 + (17 + (18 + 19)))))))))))));
4 assert(res = 190)
```

If we try to compile this program using `./hyggec compile ...`, the compiler crashes with the following error:

```
Unhandled exception. System.Exception: BUG: invalid generic register number 18
at RISC.V.Reg.r(UInt32 n) in ../src/RISC.V.fs:line 88
```

Note: The program shown in [Example 64](#) above is a solution to Exercise ??.

The reason for the crash shown in [Example 64](#) above is that, when generating assembly for a Hygge expression e , the *Code Generation Strategy* of `hyggec` tends to use a new

register for each sub-expression of e (by incrementing the current `env.Target register`): this is done to hold intermediate results that are needed to compute the final result of e . Depending on how such sub-expressions are arranged, `hygdec` may need more registers than the ones that are available.

More specifically, the API `Reg.r(n)` in the file `RISCV.fs` gives access to a generic *integer register* n selected between `t0..t6` and `s1..s11` — which means that n must be a number between 0 and 17. However, the code generation for *Example 64* tries to use:

- register `Reg.r(0)` to hold the result of the sub-expression 1;
- register `Reg.r(1)` to hold the result of the sub-expression 2;
- ...
- register `Reg.r(17)` to hold the result of the sub-expression 18;
- register `Reg.r(18)` to hold the result of the sub-expression 19 — and this causes the crash.

Solving this issue is not straightforward:

- we need to evaluate the 18 sub-expressions above *exactly* in that order, to respect the *Formal Semantics of Hygge0*. Therefore, in *Example 64* we necessarily have to produce 18 intermediate results before we can compute their final sum and initialise the variable `res`;
- even if we find an ad-hoc solution for this example, we can find many other examples of Hygge expressions that need more registers than those available.

Therefore, to improve the `hygdec` code generation we need a general **register allocation** solution that can use (and re-use) a limited number of registers in a more sophisticated way. This task is quite challenging: the program in *Example 64* does not give many hints on how we could do that. Therefore, we address the challenge by taking an intermediate step: before attempting code generation and register allocation, we translate Hygge programs into an equivalent **intermediate representation** that is closer to the generated assembly code, and makes register allocation simpler to handle.

11.2 What is an Intermediate Representation (IR)?

Generally speaking, an **intermediate representation (IR)** is any internal data structure used by a compiler to represent the code being compiled. Under this very broad definition, `hygdec` already uses *three* intermediate representations of the input source code:

- the `UntypedAST` data structure produced after parsing (defined `AST.fs` and produced by `Parser.fsy`);
- the `TypedAST` data structure produced after type checking (defined and produced in `Typechecker.fs`);
- the `Asm` data structure produced by code generation (defined in `RISCV.fs` and produced in `RISCVCodegen.fs`).

Intermediate representations are designed as stepping stones towards specific purposes in the compilation process: for example, in the case of `hyggec`, the construction of `Asm` instances (via code generation) is made possible by the type information available in `TypedAST` instances (which in turn are produced by type checking). Our purpose in this Module is to improve the register allocation strategy of `hyggec` – and unfortunately, the intermediate representation `TypedAST` seems too high-level to help us: we need an intermediate representation that is closer to the output language.

11.3 Administrative Normal Form (ANF)

Introducing an IR in a compiler requires significant effort, and poses a problem: since generating the IR requires a translation from some other IR, how do we know that the translation is correct? Or more accurately: how can we show that the behaviour of the IR coincides with the behaviour of the parsed program? To spot possible mistakes in the translation, we should:

- formalise the syntax and semantics of the IR,
- write an interpreter for the IR, and
- given a Hygge expression e , check whether interpreting e and interpreting its IR translation yield the same result. This would require writing many new tests specifically for the IR.

In this module we explore a form of intermediate representation called **Administrative Normal Form (or A-Normal Form, or ANF)**. ANF has two features that make it very appealing for `hyggec`:

1. it is close enough to RISC-V assembly to simplify the problem of register allocation; and
2. unlike other alternative IRs in compiler literature, ANF does *not* require us to introduce a new language, nor new data structures in the `hyggec` compiler. In fact, *a Hygge expression translated in ANF is still a Hygge expression*, that respects the Hygge syntax plus some additional constraints (described in [Definition 51](#) below). Therefore:
 - a Hygge expression in ANF can be represented as an `UntypedAST` or `TypedAST` in the `hyggec` compiler, and
 - we can test the correctness of the ANF translation by simply taking the existing tests for the `hyggec` interpreter, translating them into ANF, and running the ANF version with the existing interpreter, without any change. If some test fails after its ANF translation, then the ANF translation is buggy and needs to be fixed.

Definition 51 (Administrative Normal Form (ANF))

We say that a Hygge expression e is in **Administrative Normal Form (ANF)** when:

1. e follows the so-called **Barendregt convention**, i.e. all the variables bound in e must have a unique name;
2. moreover, e must be either:
 - a variable, or
 - a “let” expression “let $x : t = e_i; e_s$ ” or “let mutable $x : t = e_i; e_s$ ” where:
 - the initialisation expression e_i is either:
 - * a variable y ;
 - * a value v that is *not* a lambda term;
 - * a readInt() or readFloat() expression;
 - * a lambda term “fun $(x_1 : t_1, \dots, x_n : t_n) \rightarrow e_b$ ” where the body e_b is in ANF;
 - * an arithmetic expression “ $y + z$ ” or “ $y * z$ ” where the left-hand-side and right-hand-side are both variables;
 - * a logical expression “ y or z ” or “ y and z ” where the left-hand-side and right-hand-side are both variables;
 - * a relational expression “ $y = z$ ” or “ $y < z$ ” where the left-hand-side and right-hand-side are both variables;
 - * a logical negation “not y ” where the argument is a variable;
 - * an assertion assert(y) where the argument is a variable;
 - * a type ascription “ $y : t$ ” where the type-annotated expression is a variable;
 - * a type declaration “type $y = t; e_s$ ” where the scope expression e_s is in ANF;
 - * a structure constructor “struct $\{f_1 = y_1; \dots; f_n : y_n\}$ ” where all field initialisation expressions are all variables y_1, \dots, y_n ;
 - * a union constructor “ $l \{y\}$ ” where the initialisation expression is a variable y ;
 - * a conditional expression “if y then e_t else e_f ” where:
 - the condition expression is a variable y ;
 - the “then” branch expression e_t is in ANF;
 - the “else” branch expression e_f is in ANF;
 - * a loop “while e_c do e_b ” where:
 - the loop condition e_c is in ANF;
 - the loop body e_b is in ANF;
 - * a pattern matching “match y with $\{l_1 \{z_1\} \rightarrow e_1; \dots; l_n \{z_n\} \rightarrow e_n\}$ ” where:

- the matched expression is a variable y ;
- each continuation expression e_1, \dots, e_n is in ANF;
- the scope expression e_s is in ANF.

Example 65 (A Hygge Expression and Its ANF Equivalent)

Consider this simple Hygge program:

```
let x: int = 2 + 3;
assert(x = 5)
```

This program is *not* in ANF according to [Definition 51](#), because:

- the initialisation expression of the “let x...” is *not* in ANF: it is an addition whose two operands are not variables; and
- the expression in the scope of the “let...” is *not* in ANF: it is an assertion whose argument is not a variable.

However, the following Hygge program is in ANF, and is also equivalent to the program above — in the sense that it performs the computations, in the same order:

```
let y1: int = 2;
let y2: int = 3;
let x: int = y1 + y2;
let y3: int = 5;
let y4: bool = x = y3;
let y5: unit = assert(y4)
y5
```

Observe that in the ANF program in [Example 65](#), each sub-expression (i.e. the operands of the addition, the argument of the assertion) is first assigned to a dedicated variable, and then used to compute other expressions. In other words, each intermediate result needed to compute other results is explicitly stored in a variable. This pattern makes the ANF program very reminiscent of assembly code, as shown in [Example 66](#) below.

Example 66 (From Hygge to ANF to Assembly)

Consider this Hygge program:

```
let x: int = 1 + 2 + 3;
x
```

The following program in ANF is equivalent to the one above, in the sense that it performs the same computations and in the same order. The comments show a corresponding assembly version of the ANF program, where we use registers instead of variables.

```
let y0: int = 1;      // li t0, 1
let y1: int = 2;      // li t1, 2
let y2: int = y0 + y1; // add t2, t0, t1
let y3: int = 3;      // li t3, 3
let x: int = y2 + y3; // add t4, t2, t3
x
```

11.4 Transformation of a Hygge Expression into ANF

To transform a Hygge expression e into an equivalent ANF expression e' , according to [Definition 51](#), we can follow a procedure based on two functions:

- `toANFDefs(...)`
- `toANF(..., ...)`

The function `toANFDefs(...)`:

- takes one argument e , which is a Hygge expression, and
- returns a pair consisting of a variable y and a **list of ANF definitions** L , where:
 - the variable y (which is unique and may be autogenerated) represents the result of the expression e after its ANF transformation, and
 - the list L contains a sequence of triplets (z, m, e_z) , each one representing an ANF definition:
 - * z is a unique variable name (which may be autogenerated);
 - * m_z is a boolean saying whether the variable z is mutable or not; and
 - * e_z is the expression (in ANF) that initialises z .

The last element of L must be a triplet (y, m, e_y) – i.e. L must end with the name of the variable y returned by `toANFDefs(e)`, and its initialisation expression e_y (in ANF).

The idea is that when `toANFDefs(e)` returns the pair (y, L) , then y can be used to get the same result of the original expression e – provided that all the variables in L are initialised beforehand, in the order in which they appear in L and using their corresponding initialisation expressions (which are in ANF). For more details, see [Converting a Hygge Expression into a List of ANF Definitions](#).

Note: From now on, for brevity, when writing ANF definition triplets (z, m_z, e_z) we will often omit the element m_z when its value is false – which means that the triplet describes the definition of an immutable variable z .

Instead, the function `toANF(..., ...)`:

- takes a pair of arguments: a variable y and a list of ANF definitions L ending with the initialisation of y (as returned by `toANFDefs(...)` above), and
- returns an expression in ANF — which is a sequence of “let” binders that perform the variable definitions and initialisations described in L , and terminate with the variable y .

The idea is that `toANF(y, L)` constructs a Hygge expression in ANF that performs the initialisations and computations in L (which initialise y), and then returns the value of y . For more details, see *Converting a List of ANF Definitions into a Hygge Expression in ANF*.

Using the two functions above, we can transform a Hygge expression e in ANF by simply executing:

$$\text{toANF}(\text{toANFDefs}(e))$$

11.4.1 Converting a Hygge Expression into a List of ANF Definitions

When `toANFDefs(e)` is invoked, it proceeds as follows:

1. it chooses a (possibly autogenerated) variable y to represent the result of e ;
2. it recursively calls itself on each sub-expression of e , getting corresponding variables and lists of ANF definitions;
3. it assembles such variables and lists of ANF definitions according to the semantics of e (which establishes the evaluation order) and the requirements of *Definition 51* (which establishes what e should look like in ANF); and
4. returns the variable y together with the corresponding list of ANF definitions, ending with the triplet (y, m_y, e_y) (where e_y is the ANF expression based on e that initialises y).

More in detail, `toANFDefs(e)` processes e as follows.

- If e is just a variable y , then `toANFDefs(y)` just returns y and an empty list of ANF definitions.
- If e is just a value v that is *not* a lambda term, then `toANFDefs(v)` returns the following pair:
 - an autogenerated variable y (representing the value v , and)
 - a list of ANF definitions with just one element (y, false, v) , meaning that y is initialised by the value v .

Example 67 (From Integer Value to List of ANF Definitions)

Consider the following Hygge expression:

The result of $\text{toANFDefs}(42)$ is the following pair containing a unique variable y_0 capturing the value, and a list of ANF definitions that just initialises y_0 (which is immutable) with 42:

$$(y_0, [(y_0, \text{false}, 42)])$$

- If e is an addition “ $e_1 + e_2$ ”, then $\text{toANFDefs}(e_1 + e_2)$ must:
 - generate a unique variable y representing the result of the addition;
 - perform a recursive call on e_1 – thus getting the unique variable z_1 and its list of ANF definitions L_1 ;
 - perform a recursive call on e_2 – thus getting the unique variable z_2 and its list of ANF definitions L_2 ;
 - return the following pair:
 - * the autogenerated variable y with the result of the addition, and
 - * a list of ANF definitions consisting of L_1 , followed by L_2 , followed by the pair $(y, z_1 + z_2)$. This means that the variable y is initialised by the addition $z_1 + z_2$ – which requires that all the variables in L_1 and L_2 are initialised first.
-

Example 68 (From Addition to List of ANF Definitions)

Consider the following Hygge expression:

$$1 + 2$$

The result of $\text{toANFDefs}(1 + 2)$ is the following pair containing a unique variable capturing the result of the addition, and a list of ANF definitions introducing other (immutable) variables for the sub-expressions 1 and 2:

$$\left(y_2, \left[\begin{array}{l} (y_0, \text{false}, 1), \\ (y_1, \text{false}, 2), \\ (y_2, \text{false}, y_0 + y_1) \end{array} \right] \right)$$

- If e is a “let” binding “let $y : t = e_i; e_s$ ” or “let mutable $y : t = e_i; e_s$ ”, then $\text{toANFDefs}(e)$ must:
 - compute a unique variable y' ;
 - substitute y with y' in both e_i and e_s , getting the new initialisation expression e'_i and the new scope expression e'_s . This is necessary to ensure that all bound variables are unique, as required by [Definition 51](#);
 - perform a recursive call on the initialisation expression e'_i – thus getting the unique variable z_i and its list of ANF definitions L_i ;

- perform a recursive call on the scope expression e'_s – thus getting the unique variable z_s and its list of ANF definitions L_s ;
- return the following pair:
 - * the unique variable z_s corresponding to the result of the “let” scope, and
 - * a list of ANF definitions constructed by concatenating:
 - L_i , i.e. the ANF definitions yielded by the initialisation expression;
 - the triplet $(y', m_{y'}, z_i)$, which initialises y' (used in the “let” scope) with z_i (the result of the initialisation expression in ANF) – and where $m_{y'}$ is:
 - extbullet true if e is a “let mutable...” binding, or
 - extbullet false otherwise;
 - L_s , i.e. the ANF definitions yielded by the scope expression (which ends by initialising the returned variable z_s).

Example 69 (From “Let...” Expression to List of ANF Definitions)

Consider the following Hygge expression:

$$\begin{array}{l} \text{let } x : t = 1 + 2; \\ x + 3 \end{array}$$

The result of $\text{toANFDefs}(\dots)$ for the expression above is the following pair containing a unique variable capturing the result of the “let” expression, and a list of ANF definitions: (for brevity, here we omit the second element of each ANF definition triplet, since it is always false because each variable is immutable)

$$\left(y_5, \begin{array}{l} [(y_0, 1), \\ (y_1, 2), \\ (y_3, y_0 + y_1), \\ (x, y_3), \\ (y_4, 3), \\ (y_5, x + y_4)] \end{array} \right)$$

-
- If e is a if-then-else expression “if e_c then e_t else e_f ”, then $\text{toANFDefs}(e)$ must ensure that only one of the ANF transformations of e_t and e_f is executed, depending on whether the ANF transformation of e_c returns true or false. Therefore, $\text{toANFDefs}(e)$ must:
 - generate a unique variable y representing the result of the if-then-else expression;
 - perform a recursive call on the condition expression e_c – thus getting the unique variable z_c and its list of ANF definitions L_c ;
 - perform a recursive call on the “then” branch expression e_t – thus getting the unique variable z_t and its list of ANF definitions L_t ;

- perform a recursive call on the “else” branch expression e_f – thus getting the unique variable z_f and its list of ANF definitions L_f ;
- turn the lists of ANF definitions L_t and L_f into corresponding Hygge expressions e'_t and e'_f (both in ANF) by invoking $\text{toANF}(z_t, L_t)$ and $\text{toANF}(z_f, L_f)$ (described *in the section below*);
- return the following pair:
 - * the autogenerated variable y with the result of the if-then-else expression, and
 - * a list of ANF definitions constructed by concatenating:
 - L_c , i.e. the ANF definitions yielded by the condition expression e_c ; and
 - the pair $(y, \text{if } z_c \text{ then } e'_t \text{ else } e'_f)$, which initialises y with the result of the if-then-else expression in ANF.

Example 70 (From “If-Then-Else” Expression to List of ANF Definitions)

Consider the following Hygge expression:

$$\text{if } 2 < 3 \text{ then } 1 + 2 \text{ else } 3 * 4$$

The results of $\text{toANFDefs}(1 + 2)$ and $\text{toANFDefs}(3 * 4)$ (for the “then” and “else” branches) give, respectively, the following pairs containing a unique variable capturing the result of the expression, and a list of ANF definitions (where all defined variables are immutable):

$$\left(y_3, \begin{bmatrix} (y_0, 1), \\ (y_1, 2), \\ (y_2, y_0 + y_1) \end{bmatrix} \right) \quad \left(z_2, \begin{bmatrix} (z_0, 3), \\ (z_1, 4), \\ (z_2, z_0 * z_1) \end{bmatrix} \right)$$

If we invoke $\text{toANF}(\dots, \dots)$ (described *in the section below*) on the two pairs above, we get the corresponding Hygge expressions in ANF:

$$\begin{array}{ll} \text{let } y_0 : t = 1; & \text{let } z_0 : t = 3; \\ \text{let } y_1 : t = 2; & \text{let } z_1 : t = 4; \\ \text{let } y_2 : t = y_0 + y_1; & \text{let } z_2 : t = z_0 * z_1; \\ y_2 & z_2 \end{array}$$

The result of $\text{toANFDefs}(\dots)$ for the expression above is the following pair containing a unique variable capturing the result of the “if-then-else” expression, and a list of ANF definitions:

$$\left(w_3, \begin{bmatrix} (w_0, 2), \\ (w_1, 3), \\ (w_2, w_0 < w_1) \\ \left(w_3, \text{if } w_2 \text{ then } \begin{pmatrix} \text{let } y_0 : t = 1; \\ \text{let } y_1 : t = 2; \\ \text{let } y_2 : t = y_0 + y_1; \\ y_2 \end{pmatrix} \text{ else } \begin{pmatrix} \text{let } z_0 : t = 3; \\ \text{let } z_1 : t = 4; \\ \text{let } z_2 : t = z_0 * z_1; \\ z_2 \end{pmatrix} \right) \end{bmatrix} \right)$$

Important: The function $\text{toANFDefs}(e)$ contains one case for each possible Hygge expression e . The complete implementation of the function is available in the *hygge* Git repository, in the file `ANF.fs`. For more details, see also *Implementation: ANF Transformation and Register Allocation in hygge*.

11.4.2 Converting a List of ANF Definitions into a Hygge Expression in ANF

The purpose of the function $\text{toANF}(x, L)$ is to take a list of ANF definitions (ending with the definition of variable x) and produce a corresponding Hygge expression in ANF, consisting of a series of nested “let...” expressions that exactly match the definitions in L . The idea is that, for each ANF definition triplet (y, m_y, e_y) in the list L :

- if m_y is false (i.e. y is immutable), we produce a binder “let $y : t = e_y; \dots$ ”;
- otherwise, when m_y is true (i.e. y is mutable), we produce a binder “let mutable $y : t = e_y; \dots$ ”.

You can find a hint of this behaviour in *Example 70* above (for the “then” and “else” branches of the if-then-else); let us now see a few more examples.

Example 71 (From List of ANF Definitions to Expression in ANF (1))

Consider the pair of variable and ANF definitions in *Example 67*, obtained from the expression 42 by executing $\text{toANFDefs}(42)$:

$$(y_0, L) \quad \text{where } L = [(y_0, \text{false}, 42)]$$

By executing $\text{toANF}(y_0, L)$, we get the expression:

$$\begin{aligned} &\text{let } y_0 : t = 42; \\ &y_0 \end{aligned}$$

which is the ANF translation of the original expression 42.

Example 72 (From List of ANF Definitions to Expression in ANF (2))

Consider the pair of variable and ANF definitions in *Example 68*, obtained from the expression $1 + 2$ by executing $\text{toANFDefs}(1 + 2)$:

$$(y_2, L) \quad \text{where } L = \left[\begin{array}{l} (y_0, 1), \\ (y_1, 2), \\ (y_2, y_0 + y_1) \end{array} \right]$$

By executing $\text{toANF}(y_2, L)$, we get the expression:

$$\begin{aligned} &\text{let } y_0 : t = 1; \\ &\text{let } y_1 : t = 2; \\ &\text{let } y_2 : t = y_0 + y_1; \\ &y_2 \end{aligned}$$

which is the ANF translation of the original expression $1 + 2$.

Example 73 (From List of ANF Definitions to Expression in ANF (3))

Consider the pair of variable name y_5 and ANF definitions L obtained in [Example 69](#) from a “let...” expression. If we execute $\text{toANF}(y_5, L)$, we get the following expression:

```
let  $y_0 : t = 1$ ;  
let  $y_1 : t = 2$ ;  
let  $y_3 : t = y_0 + y_1$ ;  
let  $x : t = y_3$ ;  
let  $y_4 : t = 3$ ;  
let  $y_5 : t = x + y_4$ ;  
 $y_5$ 
```

which is the ANF translation of the “let...” expression in [Example 69](#).

Important: The implementation of the function $\text{toANF}(x, L)$ is available in the [hyggec Git repository](#), in the file `ANF.fs`. For more details, see also [Implementation: ANF Transformation and Register Allocation in hyggec](#).

Exercise 38

Convert the following Hygge expressions into ANF, according to [Definition 51](#). Ideally, you should proceed in two phases, as described above:

- first, generate the *list of ANF definitions*;
- then, generate the *corresponding ANF expression*.

You can use `hyggec` to check your answers or simply *experiment with ANF transformations*.

- $2 * 3$
 - $1 + 2 * 3$
 - let $x : t = 1$; 2
 - let $x : t = 1 * 2 * 3$; x
-

11.5 ANF-Based Linear Register Allocation

We now address the opening problem for this module: how to compile arbitrarily-complex Hygge expressions (requiring any number of intermediate results) by only using a limited number of registers. We explore the following topics:

- *Recognising and Discarding Unused Intermediate Results*
- *Why Discarding Unused Variables is Not Enough*
- *ANF-Based Code Generation with Register Allocation*

11.5.1 Recognising and Discarding Unused Intermediate Results

When a Hygge expression is converted to ANF, each one of its sub-expressions and intermediate computations becomes associated to a dedicated variable. Therefore:

- we need to maintain the results of a computation available in a register only as long as the corresponding variable z is referenced somewhere in the program;
- to understand whether a variable z (and the value it holds) is referenced somewhere in the program, we simply check whether z appears as a free variable in the scope of a “let” binder.

The idea is illustrated in *Example 74* below.

Example 74 (A Hygge Program and its Intermediate Computations)

Consider the following Hygge program:

```
1 let res: int = ((1 + 2) + 3) + 4;
2 assert(res = 10)
```

When translated into ANF, the program becomes:

```
1 let y0: int = 1;
2 let y1: int = 2;
3 let y2: int = y0 + y1; // y0 and y1 are now unused
4 let y3: int = 3;
5 let y4: int = y2 + y3; // y2 and y3 are now unused
6 let y5: int = 4;
7 let y6: int = y4 + y5; // y4 and y5 are now unused
8 let res: int = y6; // y6 is now unused
9 let y7: int = 10;
10 let y8: bool = res = y7; // y7 is now unused
11 let y9: unit = assert(y8); // y8 is now unused
12 y9
```

The comments in the program above mark the points where variables become unused (i.e. are not in the free variables of the code that follows), and therefore, their associated register could be reused to hold some other variable. For example:

- y3 can reuse the register of y0;
- y4 can reuse the register of y1;
- y5 can reuse the register of y2;
- ...

By discarding variables as soon as they become unused, and reusing their registers, the program above can be compiled to just use 3 registers.

11.5.2 Why Discarding Unused Variables is Not Enough

The approach illustrated in *the previous section* is very helpful to reduce the number of registers in use; however, it is not enough to address the *opening example of this module*. To see why, let us examine a simplified scenario where we only have 3 registers available, and consider the Hygge programs in *Example 75* below.

Example 75 (A Hygge Program using (Too) Many Registers (Simplified))

Consider the following Hygge program:

```
1 let res: int = 1 + (2 + (3 + 4));
2 assert(res = 10)
```

When translated into ANF, the program becomes:

```
1 let y0: int = 1;
2 let y1: int = 2;
3 let y2: int = 3;
4 let y3: int = 4;
5 let y4: int = y2 + y3; // y2 and y3 are now unused
6 let y5: int = y1 + y4; // y1 and y4 are now unused
7 let y6: int = y0 + y5; // y0 and y5 are now unused
8 let res: int = y6; // y6 is now unused
9 let y7: int = 10;
10 let y8: bool = res = y7; // y7 is now unused
11 let y9: unit = assert(y8); // y8 is now unused
12 y9
```

Observe that the ANF program above needs to maintain the 4 variables y0...y3 available at the same time (whereas the ANF program in *Example 74* can discard y0 and y1 and y3).

Now, suppose that our target architecture only has 3 available registers (numbered from 0 to 2). In this situation, if we try to compile this ANF program using `./hyggec compile ...`, the compiler would crash with the following error:

```
Unhandled exception. System.Exception: BUG: invalid generic register number 4
at RISC.V.Reg.r(UInt32 n) in .../src/RISC.V.fs:line 88
```

The reason for the crash is that the *Code Generation Strategy* of hyggec tends to use a new register for each sub-expression; therefore, when compiling the program above, it tries to use:

- register `Reg.r(0)` to hold variable `y0` (i.e. the result of the sub-expression 1);
 - register `Reg.r(1)` to hold variable `y1` (i.e. the result of the sub-expression 2);
 - register `Reg.r(2)` to hold variable `y2` (i.e. the result of the sub-expression 3);
 - register `Reg.r(3)` to hold variable `y3` (i.e. the result of the sub-expression 4) — and this causes the crash.
-

11.5.3 ANF-Based Code Generation with Register Allocation

To compile Hygge expressions that use more intermediate results (and ANF variables) than available registers, we need a more sophisticated approach:

- we leverage the fact that expressions are in ANF, and
- we reuse registers even when their value is still needed by the Hygge expression, by generating assembly code to:
 - **spill variables**, i.e. copy the value of a variable onto the stack, to reuse its register for another variable; and
 - **load variables**, i.e. assign a register to a previously-spilled variable (possibly after spilling another variable) and restore its value from the stack.

To achieve this, we need a **code generation environment** with the following information:

- a **target variable name** z that will contain the result of the expression being compiled.
- the **known variables** x_1, \dots, x_n ;
- each known variable might have up to *two* storage locations:
 - a **stack location** (as an offset from the frame pointer register `fp`) that never changes during the lifetime of a variable;
 - if possible, a **register** that may change during the lifetime of the variable;
- a subset of **needed variables** that should never be discarded (even if they may seem unused); and

Now, suppose that we are compiling a Hygge expression e , using the code generation environment outlined above. We proceed as described in the following subsections.

Generating Code for a “Let” Binder or Variable in ANF

According to *Definition 51*, when we start compiling a Hygge program in ANF we can expect to find either a “let” binder, or a variable.

When we generate code for a “let” binder “let $x : t = e_i; e_s$ ” or “let mutable $x : t = e_i; e_s$ ”:

1. we discard all known variables that are not “needed” in the code generation environment, nor free (i.e. not used) in e_i nor e_s (as described in the *previous section*). This helps freeing up registers and stack locations (if possible);
2. we assign both a register and a stack location to the newly-bound variable x
 - if all registers are being used, we generate code to spill one of the other known variables x_1, \dots, x_n onto its assigned stack location, and assign its register to x ;
 - if no stack locations are available, we **allocate a new stack location by decreasing the stack pointer register sp** (recall that *the RISC-V stack grows downwards*);
3. we compile the initialisation expression e_i , by targeting the variable x . Since e_i is in ANF, according to *Definition 51*, it can only have a few specific shapes (discussed *below*);
4. we discard all known variables that are not “needed” in the code generation environment, nor free (i.e. not used) in e_s ;
5. we generate code for the scope expression e_s .

When we generate code for a variable x , we proceed as follows:

1. if x does not have a register assigned to it (because it has been spilled onto the stack before), we generate code to load x onto a register;
 - if all registers are being used, we generate code to spill one of the other known variables x_1, \dots, x_n onto its assigned stack location, and assign its register to x ;
2. we check where the target variable z is stored:
 - if z has a register assigned to it, we copy the value of x onto that register;
 - otherwise, z has been spilled onto the stack, so we just copy the value of x onto the stack location assigned to z .

Generating Code for the Initialisation Expression of a “Let” Binder in ANF

According to *Definition 51*, when we generate code for the initialisation expression of a “let” binder in ANF, we can only find specific shapes of expressions. We discuss two cases: *addition* and *if-then-else*.

Generating Code for Additions

By *Definition 51*, an addition in ANF (which may only appear in the initialisation expression of a “let” binder) can only have the form “ $x_1 + x_2$ ”, i.e. its operands can only be variables. Therefore, we proceed as follows:

1. we make sure that the addition operands x_1 and x_2 and the target variable z are currently stored on a register;
 - to this end, we may need to generate code to spill up to three other variables onto the stack, and assign their registers to z , x_1 , and x_2 ;
2. we generate code for the RISC-V instruction add using the registers of x_1 , x_2 , and z .

Generating Code for If-Then-Else Expressions

By *Definition 51*, an if-then-else expression in ANF (which may only appear in the initialisation expression of a “let” binder) can only have the form “if y then e_t else e_f ” i.e. its condition expression can only be a variable; moreover, both e_t and e_f must be in ANF.

To compile the condition y , we need to make sure that y is stored into a register (and to this end, we may need to generate code to spill some other variable and assign its register to y).

The compilation of e_t and e_f proceeds recursively — but there is a catch:

- the code generated for e_t and e_f may contain code to spill and load variables;
- therefore, the register allocation at the end of e_t may be different from e_f .

This situation is illustrated in *Example 76* below.

Example 76 (Why We Must Synchronise Register Allocation Across “If-Then-Else” Branches)

Consider the following Hygge program:

```

1 let z: bool = true; // Change to 'false' to experiment
2 let x1: int = 1;
3 let x2: int = 2;
4 let x3: int = if z then x1 else 1 + (2 + (3 + 4));
5 assert(x1 = 1);

```

(continues on next page)

(continued from previous page)

```
6 assert(x2 = 2);  
7 assert(if z then x3 = x1 else x3 = 10)
```

Suppose we only have 4 registers available. The following happens:

- variable z is assigned to register $\text{Reg. } r(0)$;
 - variable $x1$ is assigned to register $\text{Reg. } r(1)$;
 - variable $x2$ is assigned to register $\text{Reg. } r(2)$;
 - variable $x3$ is assigned to register $\text{Reg. } r(3)$;
 - when compiling the “if-then-else” expression:
 - the “then” branch does not need to use any additional register;
 - the “else” branch needs 4 registers for the intermediate results of the expression $1 + (2 + (3 + 4))$. Therefore, it will need to spill the values z , $x1$, $x2$, and $x3$ onto the stack;
 - therefore, what is the register allocation when the program reaches line 5? Are the variables on the stack, or on some register? That depends on whether the ‘true’ or ‘false’ branch of the ‘if-then-else’ was taken earlier...
-

The solution to this issue is to add “synchronisation” assembly code to spill/load variables ensuring that, no matter which branch of the “if-then-else” is taken, the known variables have a known register allocation. For example, we can add at the end of the assembly code of the e_t branch some assembly code that spills/loads the known variables until their register allocation exactly matches the allocation at the end of the e_f branch. We reprise this idea when discussing its implementation in *hygdec*, in [Example 78](#) below.

11.6 Implementation: ANF Transformation and Register Allocation in *hygdec*

Tip: The conversion of Hygge programs into ANF (described in this section) is already implemented in the *hygdec Git repository*, in a file called `ANF.fs`. To see what changed since the last module, you can inspect the differences between the tags `union-rec-types` and `anf`. For more details, see also [Implementation: ANF Transformation and Register Allocation in *hygdec*](#).

The following subsections outline the implementation of ANF and register allocation available on the *hygdec Git repository*.

The version of *hygdec* released with this Module adds two option, available in some compiler phases:

- the new option `-a` or `--anf` that can be used to activate and inspect ANF conversion and ANF-based code generation with register allocation; and
- the new option `-r` or `--registers` can be used to limit the number of registers being used for code generation (useful for experiments and debugging).

Therefore, we can now invoke `hyggec` as follows:

- `./hyggec parse -a file.hyg` parses the file, converts it into ANF, and prints the result.
- `./hyggec interpret -a file.hyg` parses the file, converts it into ANF, and interprets the result.
- `./hyggec compile -a -r N file.hyg` parses the file, type-checks it, converts the result into ANF, compiles it using the *ANF-Based Code Generation with Register Allocation* with `N` available registers (default: 18), and displays the resulting RISC-V assembly code. By reducing the value of `N`, it is possible to observe how the generated code has to spill/load variables more often.
- `./hyggec compile -a -r N file.hyg` parses the file, type-checks it, converts the result into ANF, compiles it using the *ANF-Based Code Generation with Register Allocation* with `N` available registers (default: 18), and executes the resulting RISC-V assembly code using *RARS*. By reducing the value of `N`, the generated code will spill/load variables more often.

The new version of the compiler also has two new series of tests:

- `tests/interpreter-anf/` contains a series of tests that are parsed, converted into ANF, and then interpreted (these are the same tests of `tests/interpreter/`);
- `tests/codegen-anf/` contains a series of tests that are parsed, type-checked, converted into ANF, compiled using the ANF-based register allocation, and then executed in *RARS*.

Example 77 (A Hygge Program using (Too) Many Registers (Revised))

Consider again the following Hygge program (from *Example 75*):

```
1 let res: int = 1 + (2 + (3 + 4));
2 assert(res = 10)
```

If we save this file as `hygge-many-regs-simpl.hyg`, we can see its ANF translation by running:

```
./hyggec parse -a hygge-many-regs-simpl.hyg
```

and the output is:

```
Let $anf (1:1-2:16)
├─Ascription: Pretype Id "_"; pos: (1:1-2:16)
├─init: IntVal 1 (1:16-1:16)
└─scope: Let $anf_0 (1:1-2:16)
```

(continues on next page)

(continued from previous page)

```

┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
┆init: IntVal 2 (1:21-1:21)
┆scope: Let $anf_1 (1:1-2:16)
    ┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
    ┆init: IntVal 3 (1:26-1:26)
    ┆scope: Let $anf_2 (1:1-2:16)
        ┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
        ┆init: IntVal 4 (1:30-1:30)
        ┆scope: Let $anf_3 (1:1-2:16)
            ┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
            ┆init: Add (1:26-1:30)
            |     ┆lhs: Var $anf_1 (1:26-1:26)
            |     ┆rhs: Var $anf_2 (1:30-1:30)
            ┆scope: Let $anf_4 (1:1-2:16)
                ┆Ascription: Pretype Id "_"; pos:
↪(1:1-2:16)
                    ┆init: Add (1:21-1:31)
                    |     ┆lhs: Var $anf_0 (1:21-1:21)
                    |     ┆rhs: Var $anf_3 (1:26-1:30)
                    ┆scope: Let $anf_5 (1:1-2:16)
                        ┆Ascription: Pretype Id "_"
↪"; pos: (1:1-2:16)
                            ┆init: Add (1:16-1:32)
                            |     ┆lhs: Var $anf_
↪(1:16-1:16)
                                |     ┆rhs: Var $anf_4_
↪(1:21-1:31)
                                    ┆scope: Let res (1:1-2:16)
                                        ┆Ascription:
↪Pretype Id "_"; pos: (1:1-2:16)
                                            ┆init: Var $anf_5_
↪(1:16-1:32)
                                                ┆scope: Let $anf_
↪6 (1:1-2:16)
                                                    ↪
↪┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
                                                        ┆init:
↪IntVal 10 (2:14-2:15)
                                                            ┆scope:
↪Let $anf_7 (1:1-2:16)
                                                                ↪
↪┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
                                                                    ↪
↪┆init: Eq (2:8-2:15)
                                                                        ↪
↪|     ┆lhs: Var res (2:8-2:10)
                                                                            ↪
↪|     ┆rhs: Var $anf_6 (2:14-2:15)
                                                                                ↪
↪┆scope: Let $anf_8 (1:1-2:16)

```

(continues on next page)

(continued from previous page)

```

↪      ┆Ascription: Pretype Id "_"; pos: (1:1-2:16)
↪      ┆init: Assertion (2:1-2:16)
↪      |      ┆arg: Var $anf_7 (2:8-2:15)
↪      ┆scope: Var $anf_8 (1:1-2:16)

```

We can compile the program above using the new ANF-based register allocation, limited to only 4 registers, by running:

```
./hyggec compile -a -r 4 hygge-many-regs-simpl.hyg
```

and the output is:

```

1  .data:
2
3  .text:
4  mv fp, sp # Initialize frame pointer
5  addi sp, sp, -4 # Extend the stack for variable $anf
6  # Variable $anf allocation: register t0, frame pos. 1
7  li t0, 1
8  addi sp, sp, -4 # Extend the stack for variable $anf_0
9  # Variable $anf_0 allocation: register t1, frame pos. 2
10 li t1, 2
11 addi sp, sp, -4 # Extend the stack for variable $anf_1
12 # Variable $anf_1 allocation: register t2, frame pos. 3
13 li t2, 3
14 addi sp, sp, -4 # Extend the stack for variable $anf_2
15 # Variable $anf_2 allocation: register s1, frame pos. 4
16 li s1, 4
17 sw t0, -4(fp) # Spill variable $anf from register t0 to stack
18 addi sp, sp, -4 # Extend the stack for variable $anf_3
19 # Variable $anf_3 allocation: register t0, frame pos. 5
20 add t0, t2, s1 # $anf_3 <- $anf_1 + $anf_2
21 # Variable $anf_4 allocation: register t2, frame pos. 3
22 add t2, t1, t0 # $anf_4 <- $anf_0 + $anf_3
23 # Variable $anf_5 allocation: register t1, frame pos. 2
24 lw t0, -4(fp) # Load variable $anf onto register t0
25 add t1, t0, t2 # $anf_5 <- $anf + $anf_4
26 # Variable res allocation: register t2, frame pos. 1
27 mv t2, t1 # res <- $anf_5
28 # Variable $anf_6 allocation: register t1, frame pos. 2
29 li t1, 10
30 # Variable $anf_7 allocation: register t0, frame pos. 3
31 beq t2, t1, eq_true
32 li t0, 0 # Comparison result is false
33 j eq_end

```

(continues on next page)

(continued from previous page)

```

34 eq_true:
35     li t0, 1 # Comparison result is true
36 eq_end:
37     # Variable $anf_8 allocation: register t2, frame pos. 1
38     addi t2, t0, -1
39     beqz t2, assert_true # Jump if assertion OK
40     li a7, 93 # RARS syscall: Exit2
41     li a0, 42 # Assertion violation exit code
42     ecall
43 assert_true:
44     lw t0, 0(fp) # Load variable $result onto register t0
45     mv t0, t2 # $result <- $anf_8
46     li a7, 10 # RARS syscall: Exit
47     ecall # Successful exit with code 0

```

By following the comments, we can see when the code generation introduces new variables (by assigning stack space and a register) and spills/loads variables from/to registers. We can observe, in particular, that:

- variable `$anf` is spilled onto the stack on line 17, and later reloaded into a register on line 24; and
- on line 21, variable `anf_4` is assigned register `t2` and frame position 3, that (according to line 12) were previously assigned to `$anf_1` (which “disappears” without being spilled onto the stack — hence `$anf_1` has become unused and has been discarded).

11.6.1 ANF Transformation in `hyggec`

The ANF transformation is implemented in the file `ANF.fs`, and it follows the procedure described in *Transformation of a Hygge Expression into ANF*. The main function is `transform`, which then invokes `toANFDefs` and `toANF`:

```

/// Transform the given AST node into Administrative Normal Form.
let transform (ast: Node<'E, 'T>): Node<'E, 'T> =
    toANF (toANFDefs ast)

```

Here are a few remarks about the rest of the file.

- For efficiency, the function `toANFDefs` builds the list of ANF definitions in reverse order (i.e. with the latest entries at the head of the list); correspondingly, the function `toANF` expects to receive a list of ANF definitions in reverse order;
- The file includes an auxiliary function `substVar` that is very similar to `subst` (in `ASTUtil.fs`) — except that it specifically substitutes the *name* of a variable with another name.
- The unique variable names autogenerated for ANF transformation have a prefix `$anf` (often followed by an integer): this guarantees that the autogenerated names

will not clash with existing variables in the input program, since `Parser.fsy` does not accept programs that use `$` in variable names.

- When generating “let” binders, the function `toANFDefs` inserts the pretype “_”. This does not create issues, because:
 - if the ANF conversion is applied to an `UntypedAST` before calling the interpreter, the pretypes are not resolved and ignored (because the interpreter does not use any type information);
 - if the ANF conversion is applied to a `TypedAST` before code generation, the new pretypes are not resolved and they are ignored (because the code generation uses the type information available in the `AST Nodes`).

11.6.2 ANF-Based Code Generation in `hyggec`

The code generation implemented in the file `ANFRISCVCodegen.fs` follows the procedure described in *ANF-Based Code Generation with Register Allocation*.

Important: The ANF-based code generation in `ANFRISCVCodegen.fs` is only partial, and is intended as a demonstration and a basis for *Project Ideas*. In particular:

- only integer registers are considered, and
- only a few Hygge expressions are implemented (addition, if-then-else, assertions, ...) — just enough to support the examples shown in this Module.

Here is an overview of the main contents of the file.

- The code generation environment has type `ANFCodegenEnv` and it contains the information about known and needed variables, allocated registers, etc.:

```
// Code generation environment.
type internal ANFCodegenEnv = {
  // Target variable for the result of the assembly code execution.
  TargetVar: string
  // Frame position assigned to each known variable.
  Frame: Map<string, int>
  // Size of the stack frame.
  FrameSize: int
  // List of integer variables stored in registers, with newest ones
  // coming
  IntVarsInRegs: List<string * Reg>
  // List of available integer registers.
  FreeRegs: List<Reg>
  // Set of variables that are needed in the surrounding scope, and
  // should
  // never be discarded to reuse their storage.
```

(continues on next page)

(continued from previous page)

```

    NeededVars: Set<string>
}

```

- The main entry point of the ANF-based code generation is the function `codegen`, which in turn invokes `doCodegen`:

```

// Generate RISC-V assembly for the given AST (expected to be in ANF),
↳using
// the given number of registers.
let codegen (node: TypedAST) (registers: uint): RISC.V.Asm =
    // Name of a special variable used to hold the result of the program
    let resultVarName = "$result"
    // Initial codegen environment, with all registers available
    // Note: the definition of 'env' uses list comprehension:
    // https://en.wikibooks.org/wiki/F_Sharp_Programming/Lists#Using_List_
↳Comprehensions
    let env = { TargetVar = resultVarName
                Frame = Map[(resultVarName, 0)]
                FrameSize = 1
                IntVarsInRegs = []
                FreeRegs = [for i in 0u..(registers - 1u) do yield Reg.r(i)]
                NeededVars = Set[resultVarName] }
    let result = doCodegen env node
    // ...

```

Notably, the function `codegen` initialises a code generation environment by targeting a special variable called `$result`, which is initially stored in the first available location on the stack frame, *without* using any register. It also marks the `$result` variable as “needed” (so it can never be discarded during code generation, even if it is unused), and declares all registers as free.

- Unlike the default *Code Generation Strategy* of `hyggec`, the result of `doCodegen` (and other functions) in `ANFRISCVCCodeGen.fs` is not just an `Asm` instance. Instead, `doCodegen` has the following signature:

```

// Code generation function: compile the expression in the given AST node,
// which is expected to be in ANF.
let rec internal doCodegen (env: ANFCodegenEnv)
    (node: TypedAST): ANFCodegenResult = ...

```

Where the type `ANFCodegenResult` is defined as:

```

// Code generation result.
type internal ANFCodegenResult = {
    // Compiled RISC-V assembly code.
    Asm: Asm
    // Updated code generation environment.
    Env: ANFCodegenEnv
}

```

This is because the function `doCodegen` may emit assembly code to spill and load registers — and this changes the register allocation described in the input environment. For this reason, `doCodegen` returns an updated code generation environment, which should be used as input for the subsequent code generation steps.

- The following auxiliary function generates assembly code to spill a variable `varName` onto the stack:

```
// Spill the variable with the given name onto its stack position assigned.
↳in
// 'env'. Return the assembly code that performs the spill and the updated
// codegen environment.
let internal spillVar (env: ANFCodegenEnv)
    (varName: string): ANFCodegenResult = ...
```

The function above is used by the following function, that selects which variable to spill:

```
// Spill the integer variable that has been stored in an integer register.
↳for
// the longest time, saving it in the stack position assigned in 'env'.
↳Choose
// a variable that does not belong to the given 'doNotSpill' list. Return
↳the
// assembly code that performs the spilling, and the updated codegen
// environment.
let internal spillOldestIntVar (env: ANFCodegenEnv)
    (doNotSpill: List<string>): ANFCodegenResult
↳= ...
```

This highlights the strategy used by `hyggec` for selecting which variable to spill when all registers are in use: the compiler picks the variable that has been assigned to a register for the longest time. Other strategies are discussed in the *References and Further Readings*.

- The function `syncANFCodegenEnvs` generates code to synchronise the register allocation across two branches, according to the idea outlined in *Generating Code for If-Then-Else Expressions*.

```
// Generate assembly code that spills/loads variables in 'fromEnv'
↳achieving
// the same configuration of 'toEnv'.
let internal syncANFCodegenEnvs (fromEnv: ANFCodegenEnv)
    (toEnv: ANFCodegenEnv): Asm = ...
```

The function works as follows:

- it takes two code generation environments `fromEnv` and `toEnv` (e.g. one obtained after compiling the “then” branch of an if-then-else, and the other obtained after compiling the “else” branch), and

- it returns the assembly code that spills/loads the variables of `fromEnv` to/from the stack, and (re-)assigns their registers, until they reach the same configuration of `toEnv`.

To see `syncANFCodeGenEnvs` in action, you can try [Example 78](#) below.

Example 78 (Synchronising Register Allocation Across “If-Then-Else” Branches)

Consider again the following Hygge program (from [Example 76](#)):

```

1 let z: bool = true; // Change to 'false' to experiment
2 let x1: int = 1;
3 let x2: int = 2;
4 let x3: int = if z then x1 else 1 + (2 + (3 + 4));
5 assert(x1 = 1);
6 assert(x2 = 2);
7 assert(if z then x3 = x1 else x3 = 10)

```

If we save the example above in a file called `if-then-else-sync-reg-alloc.hyg`, we can see its translation into ANF by running:

```
./hygdec parse -a if-then-else-sync-reg-alloc.hyg
```

and we can observe that the “else” branch has many more ANF variable definitions (hence, it uses more registers) than the “then” branch.

To generate assembly code from the ANF, by limiting register allocation to 4 registers only, we can run:

```
./hygdec compile -a -r 4 if-then-else-sync-reg-alloc.hyg
```

The generated code is quite long, but let’s focus on the code generated for the “then” branch of the “if-then-else” expression:

```

# ...assembly code for the 'false' branch of the 'if'...
j if_end # Jump to skip the 'true' branch of 'if' code
if_true:
  mv s1, t1 # $anf_6 <- x1
  # Branch synchronization code begins here
  sw s1, -16(fp) # Spill variable $anf_6 from register s1 to stack
  sw t1, -8(fp) # Spill variable x1 from register t1 to stack
  sw t2, -12(fp) # Spill variable x2 from register t2 to stack
  sw t0, -4(fp) # Spill variable z from register t0 to stack
  lw t2, -16(fp) # Load variable $anf_6 onto register t2
  # Branch synchronization code ends here
if_end:
  # Rest of the assembly code...

```

We can see that, after the `if_true` label, there is the code for the “then” branch of the “if-then-else”:

- first, an `mv` operation copies variable `x1` into `$anf_6` (which is the ANF variable holding the result of the if-then-else); and then,
 - there is a block of “branch synchronisation code” produced by the function `syncANFCodegenEnvs` above. There we can observe, in particular, that the variable `$anf_6` is first spilled from register `s1`, and then reloaded onto register `t2`, so it gets the same register allocated after the code generation for the “else” branch. This way, no matter which branch was taken, the rest of the assembly code can use register `t2` to get the value of `$anf_6`.
-

11.7 References and Further Readings

ANF was introduced in these seminal papers, with the main purpose of compiling functional languages:

- Amr Sabry and Matthias Felleisen. *Reasoning about programs in continuation-passing style*. In Proceedings of the 1992 ACM conference on LISP and functional programming (LFP '92). <https://doi.org/10.1145/141471.141563>
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. *The essence of compiling with continuations*. In Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93). <https://doi.org/10.1145/155090.155113>

ANF is used, for instance, in the [Racket compiler](#)⁴⁰.

Conceptually, ANF is reminiscent of other intermediate representations used in various compilers — e.g. Static Single Assignment (SSA) form, and Three-Address Code. To know more, see for example:

- Keith D. Cooper and Linda Torczon. *Engineering a Compiler (Third Edition)*. Available on DTU Findit⁴¹.
 - See, in particular, Chapter 4 - Intermediate Representations
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. *Efficiently computing static single assignment form and the control dependence graph*. *ACM Transactions on Programming Languages and Systems*, Oct. 1991. <https://doi.org/10.1145/115372.115320>

The NFA-based register allocation strategy described in this Module is, broadly speaking, a coarse *linear* allocation strategy, which traverses the program once and uses the current set of free variables to find ranges of the program where variables are *live* (i.e. actually used by the program) or become permanently unused (and thus, can be discarded). Register allocation is an active research topic, and there are many different strategies, with trade-offs between complexity, speed of the code generation, and efficiency of the generated code. To know more, here are two surveys:

⁴⁰ https://docs.racket-lang.org/rackpropagator/A-Normal_Form.html

⁴¹ <https://findit.dtu.dk/en/catalog/633248ae8e5bc633f834ced5>

- Fernando Magno Quintão Pereira. *A Survey on Register Allocation*, 2008. <https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/readingMat/RegisterAllocationSurvey.pdf>
- Roberto Castañeda Lozano and Christian Schulte. *Survey on Combinatorial Register Allocation and Instruction Scheduling*. ACM Computing Surveys, May 2020. Available on DTU Findit⁴².

11.8 Project Ideas

For this Module, you should implement *both* the following Project Ideas:

- *Project Idea: Implement ANF Translation for (Some of) Your Hygge Extensions*
- *Project Idea: Implement ANF-Based Code Generation for More Hygge Expressions*

11.8.1 Project Idea: Implement ANF Translation for (Some of) Your Hygge Extensions

The ANF translation in the file `ANF.fs` does not cover the Hygge expressions you should have added in previous Project Ideas. Select at least 2 of these expressions, and extend *Definition 51*, the file `ANF.fs`, and the interpreter test suite accordingly:

- you should explain how you extend *Definition 51* to support the expressions you selected;
- you should add new cases to the functions `substVars` and `toANFDefs` in `ANF.fs`;
- you should add new tests under the directory `tests/interpreter-anf/`. Ideally, you should reuse the same interpreter tests that you have already developed for the expressions you selected, by copying them from `tests/interpreter/`.

11.8.2 Project Idea: Implement ANF-Based Code Generation for More Hygge Expressions

The ANF-based code generation in the file `ANFRISCVCodegen.fs` only supports a few Hygge expressions. For this Project Idea, you should select at least 2 more expressions, and extend `ANFRISCVCodegen.fs` to generate code for them:

- for this Project Idea, you could choose the same expressions you chose in *Project Idea: Implement ANF Translation for (Some of) Your Hygge Extensions* above — but it is not mandatory;
- in `ANFRISCVCodegen.fs`, you should add new cases to the function `doCodegen`;
- you should add more corresponding test cases under the directory `tests/codegen-anf/`.

⁴² <https://findit.dtu.dk/en/catalog/5d24d51dd9001d2d86672c8a>

Hint: If the expression you are adding has some form of branching and conditional jump (e.g. while loops, short-circuit logical operators, ...), then you may need to synchronise the register allocation of the different branches, similarly to *Generating Code for If-Then-Else Expressions*.

Module 12: Optimisation

In this Module we discuss how to optimise the code generated by `hygdec`, with the main goal of reducing the number of assembly instructions needed to execute a program.

After describing the *Overall Objective* of this Module, we will explore three families of optimisations (each one including several variants and specific cases):

- *Partial Evaluation*
- *Copy Propagation and Common Subexpression Elimination (CSE)*
- *Peephole Optimisation*

Then, you will find some *References and Further Readings* and the *Project Ideas* for this Module.

12.1 Overall Objective

There are many kinds of program optimisations, with different objectives. In this Module we will focus on **reducing the number of RISC-V assembly instructions executed by the compiled program** – which usually (but not always!) means that the compiled program runs faster. To measure the reduction, we can inspect the output of `./hygdec rars -v`, as explained in *Example 79* and *Example 80* below.

Example 79 (Instructions Count of Compiled RISC-V Programs)

Try to execute:

```
./hygdec rars -v examples/helloworld.hyg
```

The output will be similar to the following:

```
hygdec: debug: Parsed command line options:
{ File = "examples/helloworld.hyg"
  LogLevel = warning
```

(continues on next page)

(continued from previous page)

```
Verbose = true
ANF = false
Registers = 0u }
hyggec: info: Lexing and parsing succeeded.
hyggec: info: Type checking succeeded.
hyggec: debug: Created temporary directory: /tmp/hyggec-132002481
hyggec: debug: Saved assembly code in: /tmp/hyggec-132002481/code.asm
hyggec: info: Launching RARS: java
Hello, World!
hyggec: info: RARS exited with code: 0 (successful termination)
hyggec: debug: RARS output:

Program terminated by calling exit

22

hyggec: debug: Removing temporary directory: /tmp/hyggec-132002481
```

In this Module we are particularly interested in the number shown after “*Program terminated...*”: that number (in this example, 22) is the number of RISC-V assembly instructions that were executed by RARS. Our goal is to make that number smaller.

Example 80 (Instructions Count of Compiled RISC-V Programs (2))

Consider again the program in [Example 64](#), and compare the outputs of the following commands (which use the *ANF-based code generation*):

1. `./hyggec rars -a -v examples/hygge-many-regs.hyg`
2. `./hyggec rars -a -v -r 3 examples/hygge-many-regs.hyg`

In the first case, the compiler can use the default number of registers (18) and generates code that performs minimal spilling/loading of variables to/from the stack.

In the second case, the compiler is forced to using 3 registers only – therefore, it generates code that frequently spills/loads variables to/from the stack. As a result, the number of RISC-V assembly instructions executed by RARS is much much higher than the first case.

12.2 Partial Evaluation

The idea behind **partial evaluation optimisations** is to perform at compile-time some computations (a.k.a. evaluations) that would be normally performed at run-time. Consider, for example:

```
let x: int = readInt();
let y: int = x + 2 * 3 * 4;
println(y)
```

Notice that the result of $2 * 3 * 4$ will not vary between program executions, so we could compute it before generating the program code. If we do it, the program above would have the same observable behaviour of the following program – which executes less assembly instructions:

```
let x: int = readInt();
let y: int = x + 24;
println(y)
```

Partial evaluation is a rather sophisticated compiler optimisation that, when fully implemented, provides a powerful combination of several well-known optimisation techniques in compiler literature, that we briefly discuss below:

- *Constant Folding*
- *Constant Propagation*
- *Dead Code Elimination*
- *Function Inlining*

12.2.1 Constant Folding

This optimisation technique consists in recognising and computing constant results at compile-time. For example, given the following program:

```
let x: int = 10 * 4;
let y: int = 1 * 2;
println(x + y)
```

We can apply constant folding to obtain the following program, which performs less computations at run-time:

```
let x: int = 40;
let y: int = 2;
println(x + y)
```

12.2.2 Constant Propagation

This optimisation technique replaces variables having a constant value with the value itself. For example, consider again the program we obtained above after constant folding:

```
let x: int = 40;
let y: int = 2;
println(x + y)
```

We can apply constant propagation to obtain the following program:

```
let x: int = 40;
let y: int = 2;
println(40 + 2)
```

...and if we apply constant folding again, the program becomes:

```
let x: int = 40;
let y: int = 2;
println(42)
```

12.2.3 Dead Code Elimination

This optimisation technique removes parts of a program that are “dead” in the sense that they are either:

- unreachable and never executed when a program runs, or
- redundant, because their result is not used by the program.

For example, consider the program we reached above after constant folding and propagation:

```
let x: int = 40;
let y: int = 2;
println(42)
```

The definitions of `x` and `y` are now redundant (because the two variables are never used, and their initialisation is only needed to compute unused values). Therefore, the program above can be optimised as:

```
println(42)
```

As another example, consider:

```
let b: bool = 2 < 3;
if b then println("2 < 3")
  else println("2 >= 3!")
```

If we apply constant folding, the program becomes:


```
let b: bool = true;
if b then println("2 < 3!")
    else println("2 >= 3!")
```

And if we apply constant propagation, we get:

```
let b: bool = true;
if true then println("2 < 3!")
    else println("2 >= 3!")
```

Now, the definition of variable `b` is redundant, and the `else` branch of the “if-then-else” is never executed (i.e. it is unreachable code). Therefore, by applying dead code elimination we can optimise the program as:

```
println("2 < 3!")
```

12.2.4 Function Inlining

This optimisation technique expands the body of a function in the call site, in order to remove the operations needed for the function call itself – i.e. copying the call arguments on registers `a0–a7` or on the heap, jumping to the call address, setting up the return a value, jumping back to the caller.

Consider the following program:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = add(a * 2, b);
println(res)
```

To apply function inlining, we expand the function body in the call site, by substituting the call arguments. In the case of the program above, we could first apply a first optimisation via function

We could apply function inlining in different ways, depending on how we combine it with other techniques. For example, we could first optimise the program above using constant propagation, thus getting:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = add(20 * 2, 2);
println(res)
```

Now, if we apply constant folding, we get:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = add(40, 2);
println(res)
```

We can now apply function inlining, by:

1. taking the body of the function add (i.e., $x + y$);
2. substituting the call arguments, thus getting the expression $40 + 2$; and
3. replacing the call to add with the expression $40 + 2$.

The result is:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = 40 + 2;
println(res)
```

Now, by applying constant folding again, we get:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = 42;
println(res)
```

By applying constant propagation again, we get:

```
fun add(x: int, y: int): int = x + y;

let a: int = 20;
let b: int = 2;
let res: int = 42;
println(42)
```

By applying dead code elimination, we finally get the optimised program:

```
println(42)
```

12.2.5 Implementing Partial Evaluation by Leveraging the `hyggec` Interpreter

Implementing partial evaluation optimisations typically requires a substantial amount of work. Luckily, `hyggec` has an ace up its sleeve: if you observe the optimisation examples in the previous sections, you may notice that they look very similar to the transformations that the program code undergoes while being reduced by the `hyggec` built-in interpreter. Indeed, we can obtain the code optimisations above by suitably *invoking the `hyggec` interpreter after type-checking, and before code generation.*

Reducing Expressions and Their Subexpressions

The idea is that, after type-checking an expression e , we can try to optimise e by reducing it, and (if that is not possible) by reducing the subexpressions of e . More in detail:

1. we try to reduce e into e' , using the function `reduce` in `Interpreter.fs`;
2. we check the result of the attempted reduction:
 - if e reduces into e' , then we take e' and we try to reduce it again, going back to point 1 above;
 - if e cannot reduce, then:
 - if e is a “simple” value (i.e. *not* a lambda term), then we are done;
 - if e is *not* a “simple” value (i.e. it is a stuck expression, or a lambda term), then we take each subexpression of e and we try to rewrite it by reducing it recursively, according to item 1 above.

By iterating this process, we get a new, optimised expression e' that cannot be reduced — and moreover, the subexpressions of e' cannot be reduced, either. We then proceed by generating code for e' .

Avoiding “Excessive” Reductions

When attempting to reduce an expression e into an optimised expression e' (as described *above*), we want to reduce e “as much as possible”, but not “too much” — i.e. we need to be careful about two important aspects.

1. We must **not reduce expressions that perform inputs or outputs**: we want to preserve such expressions and generate code for them, so the compiled program will perform the corresponding inputs and outputs. To ensure this, we can pass to `Interpreter.reduce` a `RuntimeEnv` where the fields `Reader` and `Printer` are `None`. If we do this, an expression like `print("Hello")` will not reduce, and we will be able to generate code for it. This idea is shown in [Example 81](#) below.
2. We must ensure that **the AST of the reduced expression e' does not contain any `Pointer` instance**, because such `Pointer` instances cannot be compiled (they are only used by the interpreter). To ensure this, we must start the reductions of e

using a runtime environment `env` (of type `RuntimeEnv`) where `env.Heap` is empty. Then, we have two options.

- **Simple option:** after each reduction step e of e , we check the updated runtime environment. If its `Heap` is *not* empty, it means that the expression e' after the reduction may contain a `Pointer` instance (e.g. generated by *structure constructors* or *union constructors*) – therefore, we generate code for the expression *before* the last reduction.
- **More sophisticated option:** we reduce e and its subexpressions as much as possible (as described *above*), and *then* we inspect each subexpression of the final expression e' :
 - if e' does *not* contain any `Pointer` subexpression, we can proceed and generate code for e' ;
 - otherwise, if we find some instance of `Pointer` inside e' , we backtrack to the last reductions of e that did *not* contain any `Pointer`. To perform this backtracking, we can keep the “last good reduction” of e that does not have any `Pointer` subexpression, and backtrack to it if necessary.

These options are illustrated in [Example 82](#) below.

Example 81 (Avoiding the Reduction of Input/Output Expressions)

Consider the following Hygge expression:

```
let x: int = 40;
println(x);
println(x + 2)
```

Take a runtime environment `env` (of type `RuntimeEnv` in `Interpreter.fs`) where:

- both `Reader` and `Printer` are `None`, and
- both `Heap` and `PtrInfo` are empty maps.

Using this runtime environment, we can perform one reduction step of the program above by calling `Interpreter.reduce`, which by substitutes the “let” initialisation value (by rule [R-Let-Subst] in [Definition 4](#)) and yields an unchanged runtime environment `env`, and the more optimised expression:

```
println(40);
println(40 + 2)
```

(Observe that the effect of the reduction above yields a combination of constant propagation and dead code elimination optimisations: the constant initialisation value of `x` has been propagated in the program, and the now-redundant `let x...` definition has been discarded.)

Now, we do *not* want to reduce the `println(40)` expression above, because we want to generate an assembly program that actually performs that output. This risk is prevented by the runtime environment `env` above: if we try to use it to reduce `println(40)`,

then `Interpreter.reduce` returns `None` (because the field `env.Printer` is `None`), and this avoids the risk of accidentally reducing print statements.

However, there is opportunity for optimising the expression `println(40 + 2)` by performing constant folding on the subexpression `40 + 2`. To do this, we can leave the top-level `println(40)` expression unchanged, and attempt a reduction step on the subexpression `println(40 + 2)`: in this case, `Interpreter.reduce` yields the expression `println(42)`. After this, the program becomes:

```
println(40);
println(42)
```

There is nothing more we can reduce in this expression using `env`. Therefore, we can proceed and generate code for it.

Example 82 (Avoiding Code Generation of Pointer SubExpressions)

Consider the following Hygge expression:

```
let x: struct {f: int} = struct {f = 40}
println(x.f);
println(x.f + 2)
```

Take a runtime environment `env` (of type `RuntimeEnv` in `Interpreter.fs`) where:

- both `Reader` and `Printer` are `None`, and
- both `Heap` and `PtrInfo` are empty maps.

If we reduce the expression above using `env`, then `Interpreter.reduce` yields a program like the following, where the `struct` construction has returned a pointer `0x0001`, and a runtime environment `env'` where `env'.Heap` contains the new pointer:

```
let x: struct {f: int} = 0x0001
println(x.f);
println(x.f + 2)
```

We cannot compile this program, because there is no code generation for the pointer `0x0001`; to avoid this risk, according to the “**simple option**” described above, it is enough to check the runtime environment `env'` returned by `Interpreter.reduce`: since `env'.Heap` is not empty any more, we reject this reduction and generate code for the initial expression (i.e. the last expression where `env.Heap` was empty).

As an alternative, we could also attempt the “**more sophisticated option**” described above: even if the last reduction produced an expression with a pointer, we keep reducing the expression, updating the runtime environment (according to what is returned by `Interpreter.reduce`), and hoping to reach a more optimised expression that does *not* contain any pointer.

In this example, if we keep reducing the expression above, we get:

```
println(0x0001.f);  
println(0x0001.f + 2)
```

which still contains pointers, so cannot be compiled. If we try another reduction step, we get the following expression: (since `env'.Heap[0x0001]` contains the value 40)

```
println(40);  
println(0x0001.f + 2)
```

Now, if we try to reduce this expression using `env'`, then `Interpreter.reduce` returns `None` (because `env'.Printer` is `None`). However, we can leave the top-level `println(42)` expression unchanged, and attempt a reduction step on the subexpression `println(0x0001.f + 2)`: in this case, `Interpreter.reduce` yields the expression `println(40 + 2)`, hence the program becomes:

```
println(40);  
println(40 + 2)
```

This program does not contain pointers, so it can be compiled. But we can optimise it a bit more: in fact, if we try to further reduce the second subexpression `println(40 + 2)`, we get:

```
println(40);  
println(42)
```

Observe that this expression does not contain any `Pointer` instance; moreover, there is nothing more we can reduce using `env'`. Therefore, we can proceed and generate code for it.

Also observe that, thanks to the reductions above, we have obtained a combination of constant folding, constant propagation, and dead code elimination optimisations.

12.3 Copy Propagation and Common Subexpression Elimination (CSE)

The optimisations we discussed as variants of *Partial Evaluation* have a common limitation: they depend on the availability of constant values and expressions known at compile-time. However, this optimisation approach alone is not always effective, as shown in *Example 83* below.

Example 83 (A Hygge Program that Cannot Be Optimised with Partial Evaluation)

Consider the following Hygge expression:

```
let a: int = readInt();  
let b: int = readInt();
```

(continues on next page)

(continued from previous page)

```

let c: int = a;
let d: int = b;
let res: int = (a + b) * (a + b) * (c + d);
println(res)

```

We cannot apply partial evaluation on this expression, because:

- we cannot reduce the top-level `readInt()` expressions with a runtime environment having `Reader = None` (as required when *Avoiding “Excessive” Reductions*); moreover,
- none of the subexpressions can be reduced, either.

Still, the program in *Example 83* shows some potential for optimisation:

- the addition `a + b` is computed twice, and
- the addition `c + d` has the same value of `a + b`, so that could also be optimised.

To develop these optimisations, it is very convenient to operate on the *ANF transformation of the expression above* – in particular, to operate on the list of ANF definitions provided by the function `toANFDefs(...)` – that, for the program in *Example 83*, is shown in *Example 84* below.

Example 84 (ANF Definitions)

Consider the program in *Example 83*. The result of the function `toANFDefs(...)` applied to that program is:

$$y_5, \left(\begin{array}{l} (a, \text{readInt}()) \\ (b, \text{readInt}()) \\ (c, a) \\ (d, b) \\ (y_1, a + b) \\ (y_2, a + b) \\ (y_3, y_1 * y_2) \\ (y_4, c + d) \\ (res, y_3 * y_4) \\ (y_5, \text{println}(res)) \end{array} \right)$$

By scanning the list of ANF optimisations, we can apply two optimisations:

- *Copy Propagation*
- *Common Subexpression Elimination (CSE)*

12.3.1 Copy Propagation

This optimisation technique simplifies a list of ANF definitions as follows:

1. traverse a list of ANF definitions, starting from the top (i.e. the “oldest” definition);
2. whenever a definition introduces a variable x initialised as a copy of another variable y , then:
 - check if x is the target of an assignment “ $x \leftarrow \dots$ ” in the rest of the ANF definitions:
 - if x is the target of some assignment, leave it unchanged;
 - otherwise, if x is never the target of any assignment:
 - * remove the definition of x from the list, and
 - * replace each occurrence of y with x .

Note: If a variable is defined as immutable (like all variables the list of ANF definitions above), then we know it will never be the target of an assignment, without need to inspect the list of ANF definitions.

Example 85 (Copy Propagation in Action)

Consider the list of ANF definitions in [Example 84](#). We have that:

- c is defined as a copy of a . Therefore, we remove the definition (c, a) and replace c with a in the remaining definitions;
- d is defined as a copy of b . Therefore, we remove the definition (d, b) and replace c with a in the remaining definitions.

After these changes, the resulting list of ANF definitions is:

$$\left[\begin{array}{l} (a, \text{readInt}()) \\ (b, \text{readInt}()) \\ (y_1, a + b) \\ (y_2, a + b) \\ (y_3, y_1 * y_2) \\ (y_4, a + b) \\ (res, y_3 * y_4) \\ (y_5, \text{println}(res)) \end{array} \right]$$

12.3.2 Common Subexpression Elimination (CSE)

This optimisation technique simplifies a list of ANF definitions by avoiding the re-computation of **pure expressions**, i.e. expressions that have no side effects, such as simple additions, multiplications, boolean operations, numerical comparisons.

Therefore, CFE proceeds as follows:

1. traverse a list of ANF definitions, starting from the top (i.e. the “oldest” definition);
2. whenever a definition introduces a variable x initialised with a *pure* expression e , then:
 - check whether the same expression e is used to initialise some other variable y defined after x , but before x is used as target for an assignment “ $x \leftarrow \dots$ ”;
 - replace the initialisation of y with x (instead of e).

Note: If a variable is defined as immutable (like all variables the list of ANF definitions above), then we know it will never be the target of an assignment: therefore, if an immutable x is initialised with expression e , we can simply replace all subsequent occurrences of expression e with x .

Example 86 (Common Subexpression Elimination in Action)

Consider the list of ANF definitions in [Example 85](#). We have that:

- a and b are initialised with expressions that are not pure arithmetic expressions, so leave them as they are;
- y_1 is initialised with a pure arithmetic expression $a + b$, and the same expression is later used to initialise y_2 and y_4 : therefore, we replace the initialisations of y_2 and y_4 with y_1 .

After these changes, the resulting list of ANF definitions is:

$$\left[\begin{array}{l} (a, \text{readInt}()) \\ (b, \text{readInt}()) \\ (y_1, a + b) \\ (y_2, y_1) \\ (y_3, y_1 * y_2) \\ (y_4, y_1) \\ (res, y_3 * y_4) \\ (y_5, \text{println}(res)) \end{array} \right]$$

And if we apply *Copy Propagation* to the list of ANF definitions above, we can replace

the uses of y_2 and y_4 with y_1 , thus getting the following list of ANF definitions:

$$L = \left[\begin{array}{l} (a, \text{readInt}()) \\ (b, \text{readInt}()) \\ (y_1, a + b) \\ (y_3, y_1 * y_1) \\ (res, y_3 * y_1) \\ (y_5, \text{println}(res)) \end{array} \right]$$

And if we apply toANF(y_5, L), we get the optimised Hygge program, which computes the same results of the program in [Example 83](#) without performing duplicated computations:

```
let a: int = readInt();
let b: int = readInt();
let y1: int = a + b;
let y3: int = y1 * y1;
let res: int = y3 * y1;
let y5: unit = println(res);
y5
```

12.4 Peephole Optimisation

This optimisation technique operates close to the target language of the compiler, with the objective of removing redundant or inefficient operations that are often introduced by code generation. In the case of hygge, peephole optimisations could operate either on the list *ANF definitions* of the input program, on the list of RISC-V instructions contained in an Asm data structure (defined in RISC.V.fs). In this section, we focus on the second scenario (but the principles can be easily adapted to the first scenario).

In essence, peephole optimisation works by pattern matching over the list of assembly instructions, by looking at a limited number of instructions each time — with a “peephole” or “window” that moves along the list of instructions. When inefficient patterns are identified, they are replaced with more performant sequences of instructions.

The effectiveness of peephole optimisation depends on the input code that it is given to the code generator, and on the assembly code generation strategy:

- on the one hand, if the code being compiled is already optimised (e.g. after *Partial Evaluation* and/or *Copy Propagation and Common Subexpression Elimination (CSE)*) and the code generation is not too simplistic, peephole optimisation may not have many chances to further optimise the final target code;
- on the other hand, peephole optimisation is a simple technique that may improve the target code in cases that would be more cumbersome to handle by improving earlier phases of the compiler.

We now discuss some examples of peephole optimisations.

12.4.1 Strength Reduction

This optimisation technique replaces some assembly operations with more efficient variants that require less CPU clock cycles. As a consequence, the total number of executed assembly instructions may not change, but the execution speed will improve.

Consider a sequence of assembly instruction like the following (where r_0 , r_1 , ... represent some registers), that multiplies the content of register r_1 by 2:

```
li r0, 2
mul r2, r1, r0
```

The same result could be computed in less clock cycles by the following code, that adds the content of a register to itself:

```
li r0, 2 # Register r0 is not needed by 'add' below, but may be used later
add r2, r1, r1
```

A even more efficient alternative to both examples above is to perform a logical left-shift operation by 1 bit:

```
li r0, 2 # Register r0 is not needed by 'slli' below, but may be used later
slli r2, r1, 1
```

Example 87 (A Hygge Program Optimisable with Strength Reduction)

Consider the following simple Hygge program:

```
42 * 2
```

If we compile it, we get a RISC-V assembly program with the following instructions in the text segment:

```
mv fp, sp # Initialize frame pointer
li t0, 42
li t1, 2
mul t0, t0, t1
# ...more instructions follow...
```

We can see that the 3rd and 4th line show a multiplication pattern which can be improved by strength reduction, producing the optimised assembly code:

```
mv fp, sp # Initialize frame pointer
li t0, 42
li t1, 2
slli t0, t0, 1
# ...more instructions follow...
```

Similar strength reduction optimisations can be achieved for multiplications to other powers of 2: for example, an integer multiplication by 32 can be replaced by a logical

left-shift of 5 bits.

Similarly, an integer division by 8 can be replaced by an arithmetic right-shift of 3 bits (with the RISC-V instruction `srai`, which preserves the sign of the number being shifted).

Other optimisable operations are multiplications and additions involving the constant value 0.

12.4.2 Removal of Redundant Assignments

Some assignment patterns can be simplified. For example, consider:

```
mv r0, r1
mv r1, r0
```

It can be simplified as:

```
mv r0, r1
```

Similarly, an assignment of a value to a register that is immediately overwritten by another assignment can be removed. For example:

```
mv r0, r1
li r0, 42
```

can be optimised by omitting the first `mv` operation. Similarly,

```
mv r0, r1
mv r0, r2
```

can be optimised by omitting the first `mv` operation.

Example 88 (A Hygge Program Optimisable by Removing Redundant Assignments)

Consider the following simple Hygge program:

```
let mutable a: int = 1;
let mutable b: int = 2;
a <- b;
b <- a
```

If we compile it, we get a RISC-V assembly program with the following instructions in the text segment:

```
mv fp, sp # Initialize frame pointer
li t0, 1
li t1, 2
mv t2, t1 # Load variable 'b'
mv t0, t2 # Assignment to variable a
mv t2, t0 # Load variable 'a'
```

(continues on next page)

(continued from previous page)

```

mv t1, t2 # Assignment to variable b
mv t1, t2 # Move 'let' scope result to 'let' target register
mv t0, t1 # Move 'let' scope result to 'let' target register
# ...more instructions follow...

```

We can see that:

- on the 5th and 6th line, there is an assignment of t2 to t0, followed by a redundant assignment of t0 to t2;
- on the 7th and 8th line, there is an assignment of t2 to t1, followed by a redundant assignment of t2 to t1;

If we remove the redundant assignments, we get the optimised RISC-V assembly:

```

mv fp, sp # Initialize frame pointer
li t0, 1
li t1, 2
mv t2, t1 # Load variable 'b'
mv t0, t2 # Assignment to variable a
mv t1, t2 # Assignment to variable b
mv t0, t1 # Move 'let' scope result to 'let' target register
# ...more instructions follow...

```

12.5 References and Further Readings

The literature and research on compiler optimisations is very broad, and the techniques illustrated in this Module are only a representative sample of common techniques that can be implemented in hyggecc with a limited effort.

For instance, many compilers implement loop optimisation techniques. Part of such optimisations can be obtained in hyggecc via *Partial Evaluation* (by reducing the steps of a “while” loop) — but for a broader overview (also including more strength reduction techniques) you can see e.g.:

- João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. *Embedded Computing for High Performance*. Morgan Kaufmann, 2017. [Available on DTU Findit](#)⁴³. See, in particular:
 - Chapter 5 - Source code transformations and optimizations

The general notion of partial evaluation is discussed in this book (while the presentation in this Module is narrowed down to closely fit the hyggecc compiler and interpreter):

- Neil D. Jones, Carsten Gomard, Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. [Available on Peter Sestoft’s website](#)⁴⁴.

⁴³ <https://doi-org.proxy.findit.cvt.dk/10.1016/C2015-0-00283-0>

⁴⁴ <https://www.itu.dk/people/sestoft/pebook/>

The term “peephole optimisation” was introduced in this brief, but very influential article.

- William Marshall McKeeman. *Peephole optimization*. Communications of the ACM, July 1965. <https://doi.org/10.1145%2F364995.365000>

12.6 Project Ideas

For this Module, you should implement two of the following Project Ideas:

- *Implement Optimisations Based on Partial Evaluation*
- *Implement Copy Propagation and/or Common Subexpression Elimination*
- *Implement Some Peephole Optimisations*

Note: You do not need to implement an extensive test suite for these project ideas: it is enough to provide e.g. some examples where the number of executed RISC-V instructions is decreased when running the optimised RISC-V assembly (by inspecting the output of `./hygdec rars -v ...`, as shown in *Example 79*). In the case of *Strength Reduction*, it is enough to show some examples where inefficient assembly instructions are replaced with more efficient ones (even if the instructions count does not decrease).

Tip: To implement these project ideas, you should first pull the latest changes from the upstream *hygdec Git repository*: they add some useful extensions that simplify the work. To see what changed since the last module, you can inspect the differences between the tags `anf` and `optimization`.

12.6.1 Implement Optimisations Based on Partial Evaluation

The goal of this project idea is to implement partial evaluation by suitably invoking the *hygdec* interpreter after type-checking and before generating code, as described in *Implementing Partial Evaluation by Leveraging the hygdec Interpreter*.

Important: **Partial evaluation optimisations should not be enabled by default:** the reason is that, if partial evaluation is always enabled, then the *hygdec* code generation tests might be optimised away into very simple expressions — and as a consequence, the code generation tests would not be covering all aspects of code generation!

To control when optimisations are enabled, the latest version of *hygdec* includes a new option `-O` or `--optimize`, that takes an unsigned integer argument, and can be used as follows:

- `./hygdec compile -O 1 file.hyg`
- `./hygdec rars -O 42 file.hyg`

You can decide how to interpret the argument — as long as `0` (the default) means that partial evaluation is *not* performed. For example, you may decide that `-0 1` enables all optimisations, including partial evaluation. To see how to access the option's integer argument, see `Program.fs` and look for `opt.Optimize` (which is used to enable peephole optimisation in the *Project Idea below*).

Hint:

- You could implement a function that performs partial evaluation as part of an existing `hygdec` file, or in a separate file. If you create a new file, you will need to add it to `hygdec.fsproj`⁴⁵.
 - For better results, you may want to iterate partial evaluation multiple times, until the expression returned by the optimisation is equal to the expression given as input (which means that there is nothing more to optimise). However, if you do this, you may want to be careful if the input program contains infinite loops...
 - For this optimisation to work, it is very important that, when the interpreter reduces an AST node containing an expression e , the reduced AST node maintains the same type of the original AST node. If the type of an AST node changes during reductions, the code generation for the reduced AST node may be incorrect.
-

12.6.2 Implement Copy Propagation and/or Common Subexpression Elimination

The goal of this project idea is to implement *Copy Propagation and Common Subexpression Elimination (CSE)*. The easiest way to obtain this is to edit the file `ANF.fs`, adding the logic that optimises the list of ANF definitions returned by the function `toANFDefs`.

Hint:

- Remember that, for efficiency, the function `toANFDefs` (in the file `ANF.fs`) builds the list of ANF definitions with the most recent at the head of the list.
 - For better results, you may want to iterate this optimisation multiple times, until list of ANF definitions returned by the optimisation is equal to the list given as input (which means that there is nothing more to optimise).
-

⁴⁵ <https://fable.io/docs/your-fable-project/project-file.html>

12.6.3 Implement Some Peephole Optimisations

The goal of this project idea is to implement some cases of *Peephole Optimisation*. The starting point is the new file `Peephole.fs`, which is only used when `hygdec` is invoked with the new option `-0`.

More specifically, `Peephole.fs` contains a function `optimizeText` with a sample case of peephole optimisation: if it sees a sequence of `li` and `add` instructions, performing an addition using a just-loaded immediate value, then it replaces the two instructions with a single `addi`. The side conditions check the registers being used, and whether the immediate value is small enough to fit in 12 bits (as required by `addi`).

```

// Optimize a list of Text segment statements.
let rec internal optimizeText (text: List<TextStmt>): List<TextStmt> =
    match text with
    // If a small-enough constant integer is loaded and then added, perform
    // a direct `addi` operation instead
    | (RV.LI(rd1, value), comment1) ::
      (RV.ADD(rd2, rs1, rs2), comment2) ::
      rest
      when rd1 = rs2 && (isImm12 value) ->
        (RV.ADDI(rd2, rs1, Imm12(value)), comment1 + " " + comment2) ::
        optimizeText rest

    | stmt :: rest ->
      // If we are here, we did not find any pattern to optimize: we skip the
      // first assembly statement and try with the rest
      stmt :: (optimizeText rest)

    | [] -> []

```

You should add new cases to `optimizeText` above, using [F# list pattern matching](#)⁴⁶ (as in the code snippet above) to match sequences of RISC-V instructions that you want to optimise.

To see the effect of the sample peephole optimisation above, you can create a file called e.g. `example.hyg` containing just the expression `1 + 2 + 3`, and then see the difference between the assembly code generated by:

- `./hygdec compile example.hyg` (which does *not* use peephole optimisation)
- `./hygdec compile -0 1 example.hyg` (which *does* use peephole optimisation)

You can also inspect whether there is a difference in the number of executed assembly instructions, by launching:

- `./hygdec rars -v example.hyg` (which does *not* use peephole optimisation)
- `./hygdec rars -v -0 1 example.hyg` (which *does* use peephole optimisation)

⁴⁶ <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching#list-pattern>



ChangeLog

This is the list of changes applied to these lecture notes, with the most recent at the top.

- **01/05/2023**: added more details about the *Oral Group Examination*.
- **28/04/2023**: added examples showing *Strength Reduction* and *Removal of Redundant Assignments*.
- **26/04/2023**: published *Module 12: Optimisation*.
- **22/04/2022**: fixed typo in “let rec...” substitution in *Project Idea: Recursive Functions*, and added explanations.
- **19/04/2023**: published *Module 11: Intermediate Representations and Register Allocation*.
- **14/04/2023**: fixed some typos in *Module 10: Discriminated Unions and Recursive Types*.
- **12/04/2023**: published *Module 10: Discriminated Unions and Recursive Types*.
- **30/03/2023**: added more details to *Project Idea: Recursive Functions*.
- **29/03/2023**: published *Module 9: Closures*; fixed errors in *Definition 2* and *Definition 14*.
- **20/03/2023**: published *Module 8: Lab Day*; fixed typos in *Project Idea: Extend Hygge with Arrays*.
- **14/03/2023**: published *Module 7: Structured Data Types and the Heap*.
- **08/03/2023**: published *Module 6: Functions and the RISC-V Calling Convention*.
- **01/03/2023**: published *Module 5: Mutability and Loops*.
- **21/02/2023**: published *Module 4: Lab Day*.
- **17/02/2023**: added more hints to the *Project Ideas* of *Module 3: Hands-On with hygge*.
- **15/02/2023**: published *Module 3: Hands-On with hygge*.
- **09/02/2023**: published *Module 2: The Hygge0 Language Specification*.

- 01/02/2023: published *Module 0: Overview of the Course and Assessment* and *Module 1: Introduction to RISC-V*.