École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Edition 2026–2027
English version

Foreword

This book gathers lecture notes from a course taught at the École Normale Supérieure since 2008 and at the École Polytechnique since 2016, entitled *Programming Languages and Compilation*.

To program effectively, one must understand the execution model of the programming language(s) being used—that is, how the various constructs are executed, whether through an interpreter or a compiler. It is therefore essential to grasp all the mechanisms involved in this process. Implementing them oneself by writing a compiler is undoubtedly the best way to understand a language. And even if one does not aim to build a compiler, many underlying concepts can be reused in other contexts. Thus, notions such as formal semantics, abstract syntax, and syntactic analysis are used daily in fields as diverse as databases, computer-assisted proof, and the theory of programming languages.

This course does not rely on any specific textbook and is intended to be self-contained. However, some topics are only briefly addressed, and readers wishing to delve deeper are referred to additional resources at the end of each chapter. For example, while this course covers several concepts from language theory (grammars, regular expressions, automata, etc.) as needed for syntactic analysis, it cannot replace a dedicated course on the subject. Readers interested in learning more about formal languages may find Olivier Carton's book [5] particularly valuable.

This course does not assume any specific prior knowledge of programming languages. The OCaml language is occasionally used to illustrate certain concepts, but this remains sporadic. At other times, specific aspects of languages such as Java, Python, C, and C++ are discussed, particularly in Section 7.2, but these sections are designed to be accessible even without prior knowledge of these languages. Chapter 8 focuses on compiling a language like OCaml, but this topic is thoroughly explained beforehand in Chapter 2. Similarly, Chapters 6 and 9 focus on compiling Java and C++, but they begin by explaining the underlying concepts.

This course begins with a chapter on x86-64 assembly. Starting with the target language—the final objective—is a way to dive straight into the heart of compilation. Then, we temporarily set aside assembly to thoroughly explore all the upstream concepts of compilation. It is only in Part III that assembly will reappear.

Acknowledgments. Parts of this course are freely inspired by lecture notes from Xavier Leroy and a compilation course taught at the École Polytechnique by François Pottier between 2005 and 2015, with the kind permission of their authors. I am deeply grateful to them. I also thank Didier Rémy for his excellent LaTeX `exercise` package. I warmly thank all the individuals who contributed to this booklet through their careful proofreading, suggestions for exercises, or by leading the tutorial sessions for these courses at the École Normale Supérieure or the École Polytechnique, namely Léo Andrès, Thibaut Balabonski, Pierre-Gabriel Berlureau, Julien Bertrane, Jérôme Boillot, Noé Briand, Lélio Brun, Basile Clément, Nathanaëlle Courant, Gurvan Debaussart, Quentin Garchery, Léon Gondelman, Jacques-Henri Jourdan, Gaëtan Leurent, Louis Mandel, Valeran Maytié, Simão Melo de Sousa, Robin Morisset, Wendlasida Ouedraogo, Mário Pereira, Samuel Vivien, and Philippe Volte--Vieira. Finally, I extend my sincerest thanks to all the students who attended this course and contributed to its improvement through their many questions.

The author can be contacted by email at the following address:

`Jean-Christophe.Filliatre@cnrs.fr`

Translated from French with the help of Mistral AI.
Version 9, compiled from commit `ff98f35b`.

Contents

Part I

Preliminaries

1 x86-64 Assembly

We have chosen to focus on the x86-64 architecture here, as it is a widely used architecture today. Nevertheless, much of what we will explain remains applicable to other architectures.

The x86-64 architecture is part of a long lineage of architectures designed by Intel since 1974 and then by AMD and Intel since 2000. It belongs to the CISC family, an acronym for *Complex Instruction Set Computing*. This means that its instruction set is extensive, with many instructions allowing direct read or write access to memory, in contrast to the RISC family (*Reduced Instruction Set Computing*), which seeks to limit the instruction set. Examples of RISC architectures include MIPS, RISC-V, and ARM.

1.1 Computer Arithmetic

In computers, numbers are represented in base two. A digit in base two is either 0 or 1 and is called a *bit* (short for *binary digit*). An integer represented by n bits is conventionally written with its least significant digits on the right and its most significant digits on the left, exactly as we do in base 10.

b_{n-1}	b_{n-2}	$\ldots$	b_1	b_0

The bits b_{n-1}, b_{n-2}, etc., are called the *most significant bits*, and the bits b_0, b_1, etc., are called the *least significant bits*. The number of bits, n, is typically 8, 16, 32, or 64. When $n = 8$, it is referred to as a *byte*.

If such an integer is interpreted as *unsigned*, that is, as a natural number, its value is given by

$$\sum_{i=0}^{n-1} b_i 2^i$$

There are a total of 2^n possible values, ranging from 0 ($000\ldots000_2$) to 2^n-1 ($111\ldots111_2$)[1]. With $n = 8$, all integers from 0 to 255 can be represented. Thus, the byte 00101010_2 represents the integer 42, since $42 = 2^5 + 2^3 + 2^1$.

[1]The subscript 2 is used to clearly indicate that the number is in base 2.

To represent a *signed* integer, the *two's complement* method is used, which interprets the most significant bit b_{n-1} differently from the others, as follows:

$$-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

bits	value
100...000	-2^{n-1}
100...001	$-2^{n-1}+1$
⋮	⋮
111...110	-2
111...111	-1
000...000	0
000...001	1
000...010	2
⋮	⋮
011...110	$2^{n-1}-2$
011...111	$2^{n-1}-1$

The bit b_{n-1} is then called the *sign bit*. When $b_{n-1} = 0$, this representation matches that of an unsigned integer on $n-1$ bits, with values ranging from 0 (000...000) to $2^{n-1}-1$ (011...111). If, however, $b_{n-1} = 1$, the values range from -2^{n-1} ($100\ldots000_2$) to -1 ($111\ldots111_2$). Thus, the integer -42 is represented by 11010110 because $-42 = -128+86 = -2^7+2^6+2^4+2^2+2^1$. This asymmetric representation may seem strange at first glance, but it has many advantages. In particular, to increase the size of an integer, it is sufficient to replicate the sign bit in all the new most significant bits; this is known as *sign extension*.

It is important to understand that when n bits are stored in memory, there is no way to determine whether they represent a signed or unsigned integer. The context determines whether the bit b_{n-1} is interpreted as a sign bit. Thus, the same byte 11010110_2 can be interpreted as the signed integer -42 or as the unsigned integer 214 in different contexts.

The computer's memory can be viewed as a large array containing bytes. A specific byte in memory is accessed using its *address*, that is, an integer between 0 and $N-1$ if the memory contains N bytes. For example, if the computer has 1 GB of memory, this means that $N = 10^9$. Bytes stored in memory can represent both data (integers, character strings, addresses, etc.) and instructions. The context determines how the bytes located at a given address are interpreted, and in particular whether several consecutive bytes should be interpreted as a whole. For the machine, there is no difference between two consecutive 16-bit integers in memory and a 32-bit integer.

In the following, we will also use the word *pointer* to refer to an address, even though strictly speaking, a pointer refers rather to a variable or a program expression whose value is an address.

1.2 The x86-64 Architecture

As its name suggests, the x86-64 architecture is a 64-bit architecture, meaning that arithmetic, logical, and memory transfer operations are performed using 64-bit integers. In particular, the size of a pointer is 64 bits. Accessing a computer's memory is an expensive operation. For this reason, a microprocessor contains a small number of *registers* in which a few values can be stored and on which calculations can be performed without accessing memory. The x86-64 architecture provides 16 registers, illustrated in figure 1.1. These are 64-bit registers, whose names begin with `%r`, such as `%rax`. Portions of these registers—32, 16, or 8 bits—can be accessed using different names, as shown in the figure. Thus, the `%esi` register represents the 32 least significant bits of the `%rsi` register, and the `%al` register represents the least significant byte of the `%rax` register.

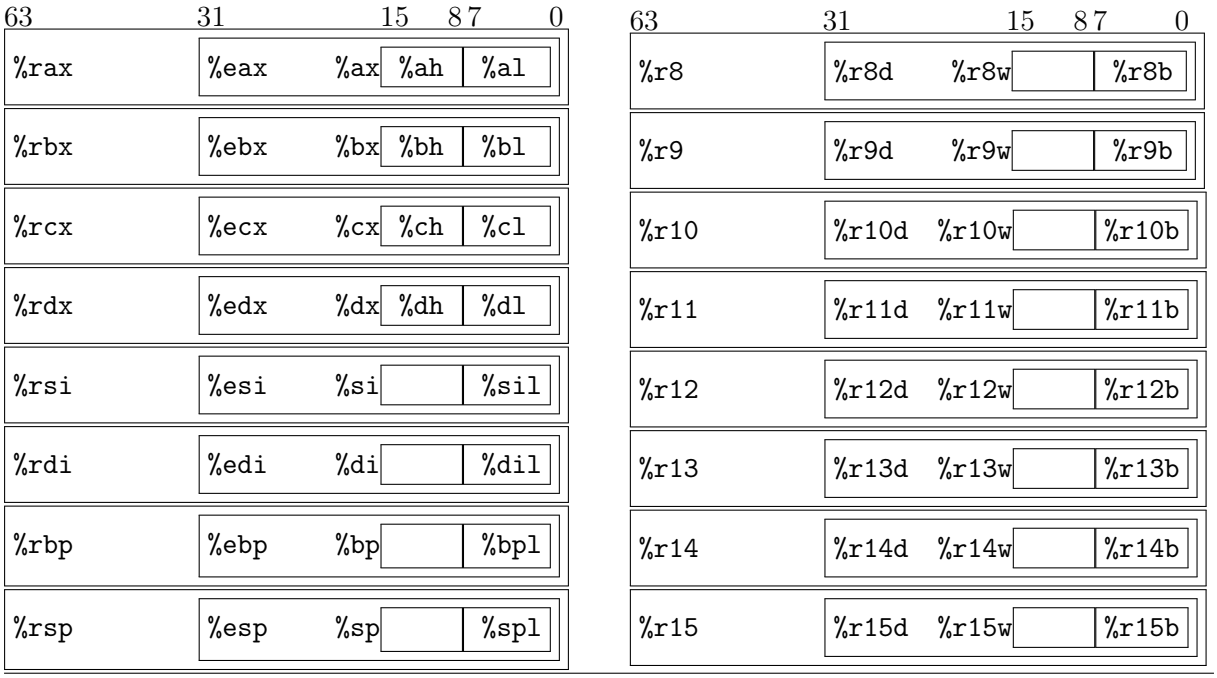


Figure 1.1: The 16 registers of x86-64.

x86-64 Assembly Language. Generally, computers are not programmed directly in machine language, but rather using a language called *assembly language*. Assembly language provides several conveniences, such as the ability to use symbolic labels instead of absolute addresses, or the ability to allocate global data. The assembly language is transformed into machine language by a program also called an *assembler*. This is therefore a compiler.

Here, we use the GNUassembler with the AT&Tsyntax, in a Linuxenvironment. Under other systems, the tools may differ. In particular, the assembler may use a different syntax, known as Intel syntax. Notably, the order of operands is reversed, which can be very confusing.

As with any language, let's start by writing a program that prints the string `"hello, world"` to standard output, followed by a newline. We write our program in a file with the suffix `.s`, for example, `hello.s`. The program text is given in figure 1.2. This program contains instructions of several kinds. First, there are declarations that begin with a dot, such as `.text`, `.globl`, `.data`, and `.string`. There are also labels, followed by a colon, namely `main` and `message`. Finally, there are x86-64 assembly instructions, such as `movq`, `call`, and `ret`.

The `.text` directive indicates that what follows is a program, that is, a sequence of instructions. The `.globl main` directive makes the symbol `main` visible, so that an executable can be built to start at this location. The actual program begins at the label `main`. We will ignore the first two instructions for now; their purpose will be explained in the next section. We want to call the library function `puts`, which prints a string to standard output followed by a newline. To do this, we must pass its argument, the string, in the `%rdi` register. We therefore place the address where the string is stored — here, the address corresponding to the label `message` — into the `%rdi` register using the instruction `movq $message, %rdi`. In the AT&Tassembly language syntax, the destination operand is on the right. The dollar sign before the label `message` indicates that we are interested

```
        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $message, %rdi
        call    puts
        movq    $0, %rax
        popq    %rbp
        ret
        .data
message:
        .string "hello, world"
```

Figure 1.2: Our first assembly program.

in the value (the address) represented by this label, not the value stored at that memory location. The second instruction, `call puts`, calls the `puts` function. After the call, we want to terminate our program. To do this, we place the return value of our program in the `%rax` register using the instruction `movq $0, %rax`. Again, the dollar sign means that we want to place the value 0 into the `%rax` register, not the value stored at address 0. Finally, we execute the `ret` instruction to end our program. The fact that the argument for `puts` must be placed in the `%rdi` register and the return value in the `%rax` register is part of conventions that we will explain later.

The rest of the program allocates the string `"hello, world"`. To do this, we start with the `.data` directive, which indicates that what follows is data, not instructions. Next comes the label `message`, which represents the address where the string is stored. Finally, we declare the string itself using the `.string` directive, which stores the string it receives as an argument in memory, appending a null character terminator (code 0). This null terminator allows `puts` to detect the end of the string, following the C convention. This completes our first assembly language program.

We assemble this program by calling the `as` command, the GNUassembler for Linux, asking it to produce the assembled code in the file `hello.o`:

```
> as hello.s -o hello.o
```

Next, we call the linker `ld` to build an executable from the `hello.o` file. As the name suggests, this is where the link is made between the reference to the `puts` function in our program and its definition in the GNUC library. Calling `ld` directly is quite complex, so we can rely on the C compiler `gcc` to do it for us[2]:

```
> gcc hello.o -o hello
```

Here, the C compiler does nothing more than call the linker, since there are no files to compile on its command line — only an already assembled file. Finally, we can execute our program, with the expected result:

[2]By passing the `-v` option to `gcc`, we can observe how `gcc` calls the linker.

```
./hello
> hello, world
```

Even more simply, we could have directly passed our `hello.s` file to `gcc`, which would have successively called `as` to assemble it and `ld` to link it, yielding the same final result.

If curious, we can examine the result of assembling our program. To do this, we use a *disassembler*, that is, a program that converts machine language back into assembly language. Under Linux, the command `objdump -d` serves this purpose. We can start by inspecting the contents of the `hello.o` file produced by `as`.

```
> objdump -d hello.o
0000000000000000 <main>:
   0:  55                    push   %rbp
   1:  48 89 e5              mov    %rsp,%rbp
   4:  48 c7 c7 00 00 00 00  mov    $0x0,%rdi
   b:  e8 00 00 00 00        call   10 <main+0x10>
  10:  48 c7 c0 00 00 00 00  mov    $0x0,%rax
  17:  5d                    pop    %rbp
  18:  c3                    ret
```

Two things are particularly noteworthy. First, the addresses of the string and the `puts` function are set to zero. Second, the program starts at address 0. The reason is that linking has not yet been performed, and these addresses are not yet known. If, however, we now disassemble the executable, we can observe the difference made by the linking process.

```
> objdump -d hello
0000000000401126 <main>:
  401126:  55                    push   %rbp
  401127:  48 89 e5              mov    %rsp,%rbp
  40112a:  48 c7 c7 30 40 40 00  mov    $0x404030,%rdi
  401131:  e8 fa fe ff ff        call   401030 <puts@plt>
  401136:  48 c7 c0 00 00 00 00  mov    $0x0,%rax
  40113d:  5d                    pop    %rbp
  40113e:  c3                    ret
```

First, the program now starts at address `0x401126`. Second, the string is stored at address `0x404030`, and the `puts` function is at address `0x401030`. We also note that the bytes of the integer `0x404030` are stored in memory in the order `30`, `40`, `40`, `00`, meaning the least significant bytes come first. This is called a *little-endian* machine (en français *petit-boutiste*). Other architectures are instead *big-endian* (en français, *gros-boutiste*), meaning they store bytes in memory starting with the most significant bytes[3]. As a first approximation, the programmer can ignore this detail, as long as integers are manipulated in their entirety using the operations provided by the machine and higher-level languages, including assembly language. However, as soon as one accesses only a portion of an integer in memory, one must be aware of the machine's endianness.

[3]The English terms *little/big-endian* and the French terms *petit/gros-boutiste* refer to Jonathan Swift's *Gulliver's Travels*.

x86-64 Instruction Set. The x86-64 instruction set contains literally thousands of instructions, each with many possible combinations of operands. It is not feasible to describe them all here. Comprehensive documentation of the x86-64 architecture is available online[4]. We will limit our description to the most common instructions, particularly those we will use later.

The operands of an instruction can be of several types. They can be an *immediate operand* of the form `$`n, where the integer n is limited to 32 bits. An operand can also be a register. Finally, an operand can refer to a memory location, either as an immediate address (a 32-bit integer) or as an *indirect address* relative to a register r. In the latter case, the operand is written as `(`r`)` and refers to the memory location pointed to by the address contained in register r. More generally, an indirect operand takes the form n`(`r_1, r_2, m`)`, where n is a constant, r_1 and r_2 are registers, and m is an integer with a value of 1, 2, 4, or 8. Such an operand refers to the memory location at the address $n + r_1 + m \times r_2$.

Data Transfer. We have already encountered the `mov` instruction, used to load a constant into a register. More generally, the instruction `mov` op_1, op_2 copies the operand op_1 to the operand op_2. When the operands do not specify the size of the data to be copied, it can be clarified using `movb` (one byte), `movw` (two bytes, or *word*), `movl` (four bytes, or *long word*), or `movq` (eight bytes, or *quad word*). For example, the instruction `movq $42, (%rdi)` writes the integer 42 as a 64-bit value to the address specified by `%rdi`. The suffix `q` is necessary here to indicate that it is a 64-bit integer. Note that not all combinations of operands are allowed.

It is not possible to perform two memory accesses in a single instruction. Thus, one cannot write `mov (%rax), (%rcx)` and must use two `mov` instructions in sequence, for example, `mov (%rax), %rax` followed by `mov %rax, (%rcx)`.

When copying a value from a smaller operand to a larger operand, one must specify whether to perform sign extension (`movs`) or to pad with zeros in the most significant bits (`movz`). The sizes of both operands must also be specified. For instance, the instruction `movsbq %al, %rdi` copies the 8-bit signed integer contained in `%al` into the 64 bits of the `%rdi` register. Finally, there is a special instruction, `movabsq`, for loading a 64-bit constant into a register.

Arithmetic and Logical Operations. Unsurprisingly, the machine provides arithmetic operations: addition (`add`), subtraction (`sub`), multiplication (`imul`), and division (`idiv`). These operations take only two operands, assigning the result of the calculation to the second operand. For example, `add $2, %rax` performs the operation `%rax` ← `%rax`+2, and `sub %rdi, %rsi` performs the operation `%rsi` ← `%rsi` − `%rdi`. This is why it is referred to as *two-address code*, as opposed to other architectures where arithmetic operations take three operands, referred to as *three-address code*. (See page 20.)

As with `mov`, many combinations of operands are possible, as long as memory is not accessed more than once. Multiplication additionally requires that its destination operand be a register. Division is somewhat special, as it takes a single operand and divides the integer contained in the combined `%rdx` and `%rax` registers by this operand, placing the quotient in `%rax` and the remainder in `%rdx`. For this reason, a special instruction (`cqto`)

[4]For example, at https://developer.amd.com/resources/developer-guides-manuals/.

copies the (signed) value contained in `%rax` into the combined `%rdx`-`%rax` registers, replicating the sign bit of `%rax` throughout `%rdx`. There are also unary arithmetic operations for incrementing (`inc`), decrementing (`dec`), and negating (`neg`).

A special arithmetic operation, `lea`, computes the effective address of an indirect operand $n(r_1, r_2, m)$, that is, the value $n + r_1 + m \times r_2$, without actually accessing memory. This value is stored in the second operand. For example, the instruction `leaq -1(%rdi), %rsi` assigns the value `%rdi` $- 1$ to the `%rsi` register. The `lea` instruction can be advantageously used to perform arithmetic calculations.

Other operations, called *logical operations*, allow bit manipulation without any specific arithmetic interpretation. For example, one can perform a logical AND (`and`), a logical OR (`or`), or an exclusive OR (`xor`) between two operands. Similarly, one can apply a logical NOT (`not`). Bits can also be shifted: to the left (`sal`), introducing 0 bits on the right; to the right (`shr`), introducing 0 bits on the left; or to the right (`sar`), replicating the sign bit. Left and right shift operations can be interpreted as arithmetic operations, respectively multiplication and division by a power of two. For example, `sarq $2, %rdi` can be seen as dividing `%rdi` by four.

Exercise 1. What does the instruction `xorq %rax, %rax` do? Solution □

Exercise 2. What does the instruction `andq $-16, %rsp` do? Solution □

Branch Operations. Arithmetic and logical instructions, except for `lea`, set Boolean *flags* based on the result of their computation.

Among these flags, we notably find ZF (the result is zero), CF (the result caused a carry beyond the most significant bit), SF (the result is negative in signed arithmetic), and OF (the result caused an overflow in signed arithmetic). The subtle difference between CF and OF lies in whether the result is interpreted as a signed or unsigned integer. These flags can then be checked by other instructions to make conditional decisions. The table below presents all the variants of these conditions, along with their meanings and how they are computed based on the flags. (Here, we use the notations from the C language: `&` for AND, `|` for OR, `^` for XOR, and `~` for NOT.)

condition	meaning	flags
`z` `e`	$= 0$	`ZF`
`nz` `ne`	$\neq 0$	`~ZF`
`s`	< 0	`SF`
`ns`	≥ 0	`~SF`
`g`	$>$ signed	`~(SF^OF)&~ZF`
`ge`	$\geq$ signed	`~(SF^OF)`
`l`	$<$ signed	`SF^OF`
`le`	$\leq$ signed	`(SF^OF)|ZF`
`a`	$>$ unsigned	`~CF&~ZF`
`ae`	$\geq$ unsigned	`~CF`
`b`	$<$ unsigned	`CF`
`be`	$\leq$ unsigned	`CF|ZF`

A *conditional branch* instruction is of the form `jcc`, where `cc` is a condition. Thus, the instruction `jz L` will continue program execution at the instruction labeled `L` if the ZF flag indicates a zero result. Otherwise, execution will continue with the next instruction. It is also possible to compute a value of 0 or 1 based on the position of the flags. This value is computed in an 8-bit operand using instructions such as `setz`, `setnz`, `sets`, etc. One can also perform a `mov` operation conditionally based on the flags, using instructions like `cmovz`, `cmovnz`, `cmovs`, etc. If the condition is not met, the `mov` instruction is ignored.

Two instructions allow setting the flags without modifying any registers. These are the `cmp` instruction, which sets the flags as the `sub` instruction would, and the `test` instruction, which sets the flags as the `and` instruction would. For example, to obtain

the result of the test `%rdi` ≤ 100 in `%al`, one can use `cmpq $100, %rdi` followed by `setle %al`. Care must be taken here with the direction of the subtraction, which can be confusing.

Unconditional Branch. Finally, note that an unconditional jump can be performed using the `jmp` instruction. The operand can be a label, as with a conditional jump, but it can also be a computed address, using the syntax `*`*op*. For example, `jmp *%rax` jumps to the address contained in the `%rax` register. This ability to transfer control to a computed address is crucial for compiling functional or object-oriented languages (see chapters 8 and 6).

Symbolic and Numeric Labels. In assembly language, symbolic labels can be used to denote addresses where instructions or data are located, such as the labels `main` and `message` in figure 1.2. However, one can also use *numeric labels*, specifically the integers `0` to `9`. Unlike symbolic labels, a numeric label can be used multiple times in the same program because it is used *locally*. Thus, if a label `1` is placed in the code, `1b` can be used to refer to the first occurrence of the label `1` higher up in the code (`b` for *backward*), and `1f` can be used to refer to the first occurrence of the label `1` further down in the code (`f` for *forward*). Here is an example of their use:

```
1:  incq  %rdi
    cmpq  %rdi, $1000
    jne   1b
```

This (admittedly not very useful) loop increments the `%rdi` register as long as its value is not equal to 1000.

Exercise 3. Assuming an input x given in `%rdi` and an output y expected in `%rax`, propose an assembly language code for the following calculations:

1. $y = 13x + 1$
2. $y = (x + 1)^2$
3. $y = x \oplus (x - 1)$ (exclusive OR)

(for 64-bit signed integers). Bonus: Try to use only two instructions each time.

Solution □

Exercise 4. What is the effect of the following code snippets?

1.
```
cmpq   $42, %rdi
setg   %al
movzbl %al, %eax
```

2.
```
movq %rdi, %rax
salq $4, %rax
leaq -42(%rdi,%rax), %rax
```

3.
```
movq   %rsi, %rax
leaq   -3(%rsi), %rdx
testq  %rdi, %rdi
cmovne %rdx, %rax
```

4.
```
movq  %rdi, %rax
notq  %rax
shrq  $63, %rax
```

Solution □

Exercise 5. Assuming that the register `%rdi` contains a value X and the register `%rsi` contains a value $N \geq 0$, explain what the following loop computes in the register `%rax`.

```
   movq  $1, %rax
1: movq  %rax, %rcx
   imul  %rdi, %rcx
   imul  %rdi, %rdi
   shr   $1, %rsi
   cmovb %rcx, %rax
   jnz   1b
```

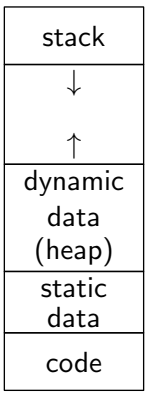
Hint: The `shr` instruction sets the `ZF` flag based on the value obtained for `%rsi` after the shift, and the `CF` (carry) flag with the least significant bit of `%rsi` before the shift.

Solution □

1.3 The Challenge of Compilation

The challenge of compilation is to translate a program from a high-level language into the instruction set we have just presented. In particular, we must translate a large number of concepts that are not directly available in assembly language, such as control structures (conditionals, loops, exceptions, etc.), function calls, complex data structures (arrays, records, objects, closures, etc.), and dynamic memory allocation.

Let's start with function calls. A first observation is that function calls can be arbitrarily nested, and the sixteen registers may not be sufficient for all the variables manipulated by these functions. Therefore, we will need to allocate memory for this purpose. Fortunately, in most programming languages, functions follow a well-nested logic: if a function f calls a function g, then the call to g will complete before the call to f does. This property means that we can use a *stack* to organize the information related to function calls.



Every running program has such a stack, called the *call stack*. This stack is located at the top of memory and grows toward decreasing addresses. At any time, the `%rsp` register (*stack pointer*) points to the top of the stack. The rest of the memory available to the program is called the *heap*. Data that must outlive function calls is allocated there. In the Linux system, the program code is located at the very bottom of memory, statically allocated data (the data segment) is just above it, followed by the heap. This arrangement ensures that memory areas do not overlap. This organization is possible because each program operates under the illusion of having the entire memory to itself. The operating system creates this illusion using a hardware mechanism called the MMU.

To manipulate the stack, we have the instructions `push` and `pop`, which allow us to push and pop values, respectively. For example, the instruction `pushq $42` pushes a 64-bit representation of the integer 42 onto the stack, and the instruction `popq %rdi` pops 64 bits from the stack and writes them into the `%rdi` register. Both of these instructions update the value of the stack pointer `%rsp`.

Of course, the stack can also be manipulated explicitly. For instance, `addq $48, %rsp` removes 48 bytes from the stack in one go.

Let's now explain how a function call is compiled. When a function `f`, called the *caller*, wants to call a function `g`, called the *callee*, we cannot simply execute the instruction `jmp g`, because we need to return to the code of `f` once `g` has finished. The solution is to use the stack. Two instructions are provided for this purpose.

First, the instruction

```
call    g
```

pushes the address of the instruction immediately following the `call` onto the stack and transfers control to the address `g`.

Second, the instruction

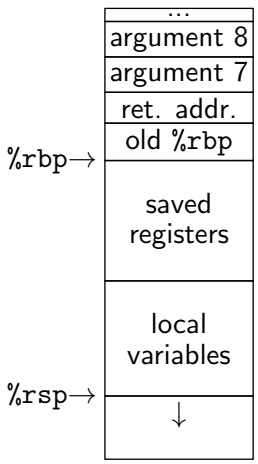
```
ret
```

pops an address from the stack and transfers control to that address. In our first program written earlier, we used the `ret` instruction to terminate our program. In reality, the program was not completely finished. We had simply returned to a piece of code installed by the system, which had called our program. The return address was at the top of the stack, and our program had not modified the state of the stack — otherwise, we would have encountered an unpleasant surprise.

We now have a mechanism to perform a function call and return to the call site once it is completed. However, we still face a significant issue: any register modified by the callee may potentially be lost for the caller. There are multiple ways to address this, but generally, we rely on *calling conventions*.

These conventions depend on the architecture and sometimes even the operating system. On the x86-64 architecture, the conventions are as follows. The first six arguments are passed in the registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. Any additional arguments are passed on the stack, if necessary. The return value is passed in `%rax`. The registers `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, and `%r15` are called *callee-saved*, meaning the callee is responsible for saving them. These registers are typically used for long-lived data that must survive across function calls. The remaining registers are *caller-saved*, meaning the caller is responsible for saving them. These registers are typically used for data that does not need to survive across function calls. Of course, if the callee does not modify a callee-saved register, it does not need to save it. Similarly, if the caller does not modify a caller-saved register, it does not need to save it either. Finally, the registers `%rsp` and `%rbp` are reserved for stack management.

Calling a function `f` therefore involves constructing a more or less large set of data at the top of the stack. At a minimum, this includes the return address. However, it may also include additional arguments (if there are more than six), saved callee-saved registers that the function `f` wishes to use, local variables that cannot be allocated in registers, etc. All of these data elements together form what is called the *stack frame* of this call. Since the size of this record can vary during the execution of the call, it is customary to use the `%rbp` register to point to the beginning of the stack frame[5]. Thus, the two registers `%rbp` and `%rsp` frame the stack frame located at the top of the stack. Since the `%rbp` register is one of the callee-saved registers, it must be saved, which means its



[5]The name `%rbp` stands for *base pointer*.

previous value itself becomes part of the stack frame. This results in the situation illustrated alongside.

The construction and destruction of the stack frame during a function call are divided into four steps, as follows:

1. Just before the call, the caller passes the first six arguments in `%rdi`, ..., `%r9`, and any additional arguments on the stack if there are more than six; saves the caller-saved registers it intends to use after the call (in its own stack frame); and executes the `call` instruction.

2. At the beginning of the call, the callee saves `%rbp` and then sets it, for example, by executing the two instructions `pushq %rbp` and `movq %rsp, %rbp`; allocates its stack frame, for example, by decreasing the value of `%rsp`; and saves the callee-saved registers it will need.

3. At the end of the call, the callee places the result in `%rax`; restores the saved registers; deallocates its stack frame and restores `%rbp`, for example, by executing the two instructions[6] `movq %rbp, %rsp` and `popq %rbp`; and finally executes `ret`.

4. Just after the call, the caller removes any arguments it had placed on the stack; and restores the caller-saved registers it had saved.

Finally, the calling conventions stipulate that the stack must be *aligned* at the time of a call, meaning that `%rsp` must be a multiple of 16 when executing a `call` instruction (and will thus be 8 modulo 16 immediately after the call, since the return address will have been pushed onto the stack by `call`). This stack alignment is particularly required by certain functions that save vector registers on the stack. Notably, library functions such as `scanf` or `printf` may crash if the alignment is not respected. Stack alignment can be done explicitly:

```
f:   subq $8, %rsp  # align the stack
     ...
     ... # because we make calls to external functions
     ...
     addq $8, %rsp
     ret
```

or achieved for free by saving a register on the stack:

```
f:   pushq %rbp  # save %rbp
     ...
     ...
     popq  %rbp  # and restore it
     ret
```

It is important to understand that calling conventions are just conventions. In particular, you are free not to follow them as long as you stay within the scope of your own code. However, if you interface with external code, you must respect the calling conventions.

Conventions Are Just Conventions. If we observe the assembly code produced by the OCaml compiler for the following function,

```
let f x y z = x + y + 8*z
```

we immediately notice that the OCaml compiler does not use the usual x86-64 conventions:

```
f: addq  %rbx, %rax
   leaq -9(%rax,%rdi,8), %rax
   ret
```

Indeed, we observe that the three parameters `x`, `y`, and `z` are passed in the registers `%rax`, `%rbx`, and `%rdi`, respectively. Passing the first parameter in the same register as the one used for the result is an excellent idea when composing functions. Thus, when compiling an expression like `f (g x)`, the result of `g` is directly in the correct register as the argument for `f`. And composing functions is natural in a functional programming language.

We also note an arithmetic "oddity": the function appears to incorrectly compute `x + y + 8z - 9` instead of `x + y + 8 * z`. This is because the OCaml compiler represents the integer value n of type `int` as the machine integer $2n + 1$, so that the garbage collector (GC) can distinguish pointers, which are necessarily even for alignment reasons, from integers, which are necessarily odd in this representation. Perhaps surprisingly, OCaml integers are therefore 63-bit signed integers.

1.4 A Compilation Example

To truly understand the challenges of compilation, there is no better exercise than compiling a program by hand. Consider the C program shown in figure 1.3. This is an extremely clever program that calculates the number of solutions to the n-queens problem, *i.e.*, the number of different ways to place n queens on an $n \times n$ chessboard so that none threaten each other. This program is particularly interesting from a compilation perspective, as it includes a conditional test, a loop, a recursive function, and various calculations, all while remaining reasonably concise. Incidentally, it is also a program of interest in its own right, as it provides an excellent solution to the n-queens problem.

Exercise 6. Explain how this program works. Hints: The program attempts to fill the chessboard row by row; the integers `a`, `b`, `c`, `d`, and `e` should be viewed as arrays of Booleans. Solution □

Let's start with the compilation of the recursive function `t`. We need to allocate registers for this function. The three arguments `a`, `b`, and `c` are passed in the registers `%rdi`, `%rsi`, and `%rdx`. Since the result will be returned in `%rax`, we choose to allocate the local variable `f` in this register to simplify the compilation of the `return f` instruction. For the other two local variables, `d` and `e`, we choose to allocate them in the registers `%r8` and `%rcx`, respectively. In the case of a recursive call, the values of the six variables `a`, `b`, `c`, `d`, `e`, and `f` will need to be saved, as they are all used after the call. Since these

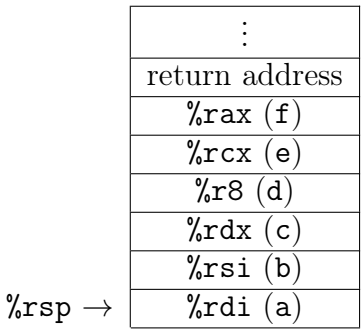
[6] The `leave` instruction is equivalent to these two instructions.

```
int t(int a, int b, int c) {
  int f = 1;
  if (a) {
    int d, e = a & ~b & ~c;
    f = 0;
    while (d = e & -e) {
      f += t(a-d, (b+d)*2, (c+d)/2);
      e -= d;
    }
  }
  return f;
}
int main() {
  int n;
  scanf("%d", &n);
  printf("q(%d) = %d\n", n, t(~(~0<<n), 0, 0));
}
```

Figure 1.3: A C program to compile.

are six caller-saved registers, it is the caller's responsibility to save them. In the case of a recursive call, as here, the caller and the callee are the same function, so it makes no difference whether we choose caller-saved registers saved by the caller or callee-saved registers saved by the callee.

If we forgo the use of `%rbp`, which is unnecessary here, the stack frame of the function `t` takes the following form:



The assembly code for the function `t` is given in figure 1.4, in the left column. We start by assigning the value 1 to `f` (line 4), then we test the value of `a` (lines 5–6). If `a` is zero, we jump to the end of the function (label `t_return`, line 43). Otherwise, we allocate 48 bytes for the stack frame (line 7). Then, we set `f` to 0 (line 8) and assign the value of the expression `a & ~b & ~c` to the variable `e` (lines 9–15)[7]. Next comes the `while` loop. To perform only one branch per loop iteration, we can place the loop body first (lines 17–35) followed by the loop test (lines 36–42). Consequently, we start with an unconditional jump to the test (line 16). The loop body begins by saving the six variables in the stack frame (lines 18–23). Then, we prepare the three arguments for the recursive call (lines 24–28)

[7]The x86-64 instruction set provides an `andn` operation that combines an `and` and a `not` and would save a few instructions here.

before making the call (line 29). Once the call is completed, we restore the variables while updating the variables `e` and `f` (lines 30–35). Finally, the loop test consists of assigning to `d` the value of the expression `e & -e` (lines 37–40)[8] and then testing if the result is non-zero (line 41). Here, we use the fact that the last arithmetic operation performed (`andq` on line 40) set the flags with the value of `d`. When we exit the loop, line 42 deallocates the stack frame.

Let's now examine the assembly code for the `main` function, given in figure 1.4 in the right column. We start by saving `%rbp` and setting it, which also aligns the stack. Next, we prepare the call to `scanf`. The two arguments are placed in `%rdi` and `%rsi`, respectively. Note that the second argument is the *address* of the variable `n`, not its value, which is why we must write `$n` rather than `n` (line 49). Since `scanf` is a variadic function, we must indicate the number of floating-point arguments in `%rax`. As there are none here, we set `%rax` to zero (line 50). Then, we prepare the arguments for the call to the function `t`. In particular, we compute the value of `~(~0 << n)` in `%rdi` (lines 53–57). When the shift value is not constant, the `salq` instruction requires the shift amount to be in the `%cl` register, which is why we had to copy the value of `n` into `%rcx`. After the call to `t` (line 60), all that remains is to make the call to `printf` (lines 62–66). As with `scanf`, we must set `%rax` to zero (line 65). We conclude our program by setting the return value (lines 67–69). Beyond the code, we find the statically allocated data, namely the variable `n` and the two format strings passed to `scanf` and `printf`, respectively. The directive `.quad 0` reserves eight bytes initialized with the value 0. The directive `.string` allocates a string terminated by a null character (ASCII code 0), following the convention of the C language and, in particular, the functions `scanf` and `printf` used here.

Exercise 7. Write in assembly language the following function that computes the integer part of the square root of n:

```
int isqrt(int n) {
  int c = 0, s = 1;
  while (s <= n) {
    c++;
    s += 2*c + 1;
  }
  return c;
}
```

Display the value of `isqrt`(17). Solution □

Exercise 8. Write the factorial function in assembly language, first using a loop, then using a recursive function. Solution □

[8] There is an instruction `blsi` that corresponds exactly to this calculation, but we have chosen to stick with elementary operations here.

```
1          .text
2          .globl main
3  t:
4          movq    $1, %rax
5          testq   %rdi, %rdi
6          jz      t_return
7          subq    $48, %rsp
8          xorq    %rax, %rax
9          movq    %rdi, %rcx
10         movq    %rsi, %r9
11         notq    %r9
12         andq    %r9, %rcx
13         movq    %rdx, %r9
14         notq    %r9
15         andq    %r9, %rcx
16         jmp     loop_test
17 loop_body:
18         movq    %rdi,  0(%rsp)
19         movq    %rsi,  8(%rsp)
20         movq    %rdx, 16(%rsp)
21         movq    %r8,  24(%rsp)
22         movq    %rcx, 32(%rsp)
23         movq    %rax, 40(%rsp)
24         subq    %r8, %rdi
25         addq    %r8, %rsi
26         salq    $1, %rsi
27         addq    %r8, %rdx
28         shrq    $1, %rdx
29         call    t
30         addq    40(%rsp), %rax
31         movq    32(%rsp), %rcx
32         subq    24(%rsp), %rcx
33         movq    16(%rsp), %rdx
34         movq     8(%rsp), %rsi
35         movq     0(%rsp), %rdi
36 loop_test:
37         movq    %rcx, %r8
38         movq    %rcx, %r9
39         negq    %r9
40         andq    %r9, %r8
41         jnz     loop_body
42         addq    $48, %rsp
43 t_return:
44         ret
```

```
45 main:
46         pushq   %rbp
47         movq    %rsp, %rbp
48         movq    $input, %rdi
49         movq    $n, %rsi
50         xorq    %rax, %rax
51         call    scanf
52
53         xorq    %rdi, %rdi
54         notq    %rdi
55         movq    (n), %rcx
56         salq    %cl, %rdi
57         notq    %rdi
58         xorq    %rsi, %rsi
59         xorq    %rdx, %rdx
60         call    t
61
62         movq    $msg, %rdi
63         movq    (n), %rsi
64         movq    %rax, %rdx
65         xorq    %rax, %rax
66         call    printf
67         xorq    %rax, %rax
68         popq    %rbp
69         ret
70
71         .data
72 n:
73         .quad   0
74 input:
75         .string "%d"
76 msg:
77         .string "q(%d) = %d\n"
```

Figure 1.4: The result of compiling the program from figure 1.3.

Exercise 9. Explain what the following function `f` computes:

```
f:      movq    %rsi, %rax
1:      subq    %rax, %rdi
        jg      1b
        jz      2f
        addq    %rdi, %rax
        negq    %rdi
        jmp     1b
2:      ret
```

Solution □

Exercise 10. Using *Compiler Explorer* (`godbolt.org`), observe how `gcc` compiles a *loop* (for example, calculating a sum or a factorial). In particular, vary

- the type of loop: `while`, `do`, or `for`;
- the optimization level: `-Og`, `-O1`, `-O2`.

Solution □

Bibliographical Notes. The book *Computer Systems* by Bryant and O'Hallaron [4] is an excellent source of information about computer architecture, particularly x86-64, from the perspective of both general programmers and those specifically interested in writing a compiler. The website *Compiler Explorer* (`godbolt.org`) is an unparalleled tool for observing and understanding the code produced by compilers. To learn more about how a linker works, we highly recommend the book *Linkers and Loaders* by John Levine [20], as well as Chapter 7 of *Computer Systems* [4].

SIMD Registers. The x86-64 architecture also includes vector registers, known as *SIMD* for *Single Instruction Multiple Data*. There are 16 SSE vector registers, named `%xmm0`,...,`%xmm15`. A vector register is 128 bits wide and can hold, depending on the context, one or more integers or floating-point numbers: two 64-bit integers, four 32-bit integers, two double-precision floating-point numbers, etc. The machine provides instructions that perform the same operation in parallel on the different elements of the vector registers, such as simultaneously adding four pairs of 32-bit integers contained in two vector registers (using the `paddd` instruction). One can easily imagine the usefulness of such instructions for numerical computations, but a good compiler will also use them to optimize code in other situations.

These registers are also used for floating-point calculations, even when the computation is not vectorized (*i.e.*, applies only to single operands). For example, the following assembly program

```
square:
  mulsd %xmm0, %xmm0
  ret
```

is a function that takes a double-precision floating-point number in the `%xmm0` register and returns its square in the same register. (This is the calling convention for floating-point numbers.) Only the lower part of the `%xmm0` register is used here.

Today, processors with 256-bit vector registers (type AVX2, named `%ymm0`...) or even 512-bit (type AVX-512, named `%zmm0`...) are available. The latter architecture even offers 32 such registers.

Example of a RISC Architecture: RISC-V. This box provides a brief introduction to the RISC-V architecture, aiming to illustrate both the differences and similarities with the x86-64 architecture we have just discussed. Here, we focus on a 32-bit architecture.

The main characteristics of RISC-V are: few instructions, particularly very few instructions for accessing memory; highly regular instructions (each instruction is encoded in 32 bits), with limited operands for each instruction. This is motivated by efficiency considerations: lower cost, reduced heat dissipation, smaller footprint, etc. Additionally, RISC-V features a large number of registers, specifically 32 registers of 32 bits each (named `x0`, `x1`, ..., `x31`, but also accessible under other names that reveal their traditional usage).

A notable difference from x86-64 is that most instructions have three operands: two source operands and one destination operand. For example, the addition instruction `add x1, x2, x3` assigns the sum of registers `x2` and `x3` to register `x1`. (Note that the order is different from the AT&T syntax!) This is known as *three-address code*. However, there is nothing preventing the use of multiple identical operands, allowing for two-address code in specific cases. Furthermore, the `zero` register (`x0`) always contains the value 0. Here is a function that computes the n-th Fibonacci number, with the argument and result in `a0`:

```
fib:add  a2, zero, a0  # a2 = loop iterations
    addi a0, zero, 0   # a0 = F(i)
    addi a1, zero, 1   # a1 = F(i+1)
    jal  zero, 1f      # jump to 1
  0:add  a1, a0, a1
    sub  a0, a1, a0
  1:addi a2, a2, -1
    bge  a2, zero, 0b  # loop while a2 >= 0
    jalr zero, ra, 0   # return = jump to ra
```

Here, we used RISC-V instructions directly, but the assembler also provides many *pseudo-instructions* that are expanded into one or two elementary instructions. For instance, one could simply write `li a1, 1` to load the constant 1 into register `a1`, and the assembler would transform it into an addition with the `zero` register.

Beyond these differences, the same core principles as in x86-64 are found:

- a stack at the top of memory, with a `sp` register as the stack pointer;
- calling conventions with registers for arguments (`a0`–`a7`) and results (`a0`–`a1`), caller-saved registers (`ra`, `a0`–`a7`, `t0`–`t6`), and callee-saved registers (`sp`, `s0`–`s11`).

Note that there is no `call` instruction analogous to x86-64 that would place the return address on the stack. Instead, there is a simpler instruction, `jal`, which places the return address in a register (typically `ra`). It is the user's responsibility to save it on the stack at the function entry if needed. (There is a `call` instruction in RISC-V, but it is merely a pseudo-instruction built on top of `jal` for long jumps.)

2 What Is a Programming Language?

How can we define the meaning of programs written in a programming language? Most of the time, we rely on an informal description in natural language: ISO standards, reference books, online documentation, etc. Such a definition often turns out to be incomplete, imprecise, or even ambiguous.

The Javalanguage, for example, is defined in the book *The Java Language Specification* [10]. Although very precise, this book still contains some ambiguities. For instance, it includes the following specification regarding the order of evaluation of operands:

> The Java programming language guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

However, this is immediately followed by a comment that can only leave the reader perplexed:

> It is recommended that code not rely crucially on this specification.

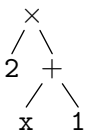
Nevertheless, we are far from blaming the authors of this Java definition. Many languages do not offer such a detailed definition and often provide their compiler as the sole reference. Programmers must then engage in experimental discovery of the semantics or a careful reading of the compiler's code. When they lack the courage to do so, they rely on others to do this work. Websites like `stackoverflow.com` contain countless questions about the semantics of programming languages. The archiving of all these answers becomes a *de facto* reference for many programmers.

However, there are situations where an informal definition is not sufficient. This is particularly the case when we want to *reason* about programs, for example, to prove the correctness of a type system or a compiler. *Formal semantics* is a mathematical tool that allows us to give a precise meaning to a program by describing the computations it performs.

Before attempting to define a formal semantics, it is necessary to agree on what a program is. As a syntactic object, that is, the sequence of characters that make up the source code of the program, it is too difficult to manipulate. In particular, the presence of whitespace, comments, or redundant parentheses would complicate the mathematical discourse. Instead of this notion of concrete syntax, we prefer the notion of *abstract syntax*.

2.1 Abstract Syntax

Abstract syntax is a representation of a program in the form of a tree-like data type. Thus, a program expression that represents the multiplication of the constant 2 by another expression that represents the addition of the variable `x` and the constant 1 is materialized by the following *abstract syntax tree*:



In the concrete syntax of a particular programming language, such an expression could be written as:

```
2*(x+1)
```

But it could also be written with more parentheses, which are unnecessary here:

```
(2 * ((x) + 1))
```

Or even with a comment, which is also unnecessary here:

```
2 * (* I multiply by two *) ( x + 1 )
```

These three program expressions, and many others, all represent the same abstract syntax tree given above.

Let us formally define what an abstract syntax tree is for such arithmetic expressions. By limiting ourselves to the four constructions used above, an expression e is either a constant, a variable, the addition of two expressions, or the multiplication of two expressions. We formally express this using a *grammar* as follows:

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e + e & \textit{addition} \\ & | & e \times e & \textit{multiplication} \end{array}$$

This can be seen as the definition of a type of trees whose leaves are constants or variables and whose internal nodes are additions or multiplications. Here, c represents any constant (e.g., `1`, `42`, etc.) and x represents any variable (e.g., `a`, `foo`, etc.). In particular, we must not confuse the meta-variable x, which represents any variable, with the variable named `x`.

The attentive reader has already noticed that the abstract syntax does not explicitly represent parentheses among its constructions. Parentheses are useful in concrete syntax, for example, to distinguish between the two expressions `2*(x+1)` and `2*x+1` if we assume multiplication has priority over addition, as is customary. In abstract syntax, these two expressions are represented by two different trees, namely:

There is no need to retain the parentheses of the first expression in the abstract syntax tree. The shape of these two trees is sufficient in itself. We will see in chapter 4 how parsing translates concrete syntax into abstract syntax and, in particular, how the presence of parentheses influences the construction of abstract syntax trees.

In the grammar above, we reused infix notations for addition and multiplication that echo those of the concrete syntax. This is simply for convenience. We could just as well have chosen different symbols, such as $\oplus$ or *Add*. For convenience, we will not systematically draw abstract syntax trees but will use a linear notation where parentheses are used only to show the structure. Thus, we will allow ourselves to write $2 \times (x + 1)$, while being aware that this is abstract syntax and not concrete syntax, and that the parentheses are only there to show that the root of the tree is the multiplication.

It is not only parentheses that constitute a difference between concrete syntax and abstract syntax. Some constructions in concrete syntax can be expressed directly using other constructions in abstract syntax. This is called *syntactic sugar*. Let us add, for example, to our arithmetic expressions a new syntactic construction `succ` to represent the $+1$ operation, allowing us to write `2 * succ x` instead of `2 * (x + 1)`. Such an operation could be added to the abstract syntax as a fifth construction. However, it can also be expressed using operations that already exist, namely addition and the constant `1`. The advantage of the second option is to limit the number of constructions in the abstract syntax. For example, in the language C, the construction `a[i]` is syntactic sugar for `*(a+i)`, and in the language OCaml, the construction `[a; b; c]` is syntactic sugar for `a :: b :: c :: []`.

Machine Representation. Abstract syntax is not just a tool for the mathematician seeking to define a formal semantics. It is also the representation of programs in a machine within an interpreter or a compiler. In a language like OCaml, an algebraic type is an immediate transcription of the abstract syntax:

```
type expression =
  | Cte of int
  | Var of string
  | Add of expression * expression
  | Mul of expression * expression
```

The constructors `Cte` and `Var` contain the value of the constant and the name of the variable, respectively.

Exercise 11. Write a function `succ: expression -> expression` corresponding to the $+1$ operation. Solution □

Exercise 12. Write a function `eval: (string -> int) -> expression -> int` that computes the value of an expression. The first argument is a function that gives the value of each variable. Solution □

2.2 Operational Semantics

Operational semantics describes the sequence of elementary computations that lead from a program expression to its value. It operates directly on the abstract syntax. There are

two forms of operational semantics: *natural semantics*, also called big-step semantics, and *reduction semantics*, also called small-step semantics.

To illustrate these two notions, we choose the Mini-ML language. This is a fragment of ML reduced to its essentials. It includes first-class functions, pairs, and local variables. The abstract syntax of Mini-ML is as follows:

e	::=	x	identifier
	|	c	constant ($1, 2, \ldots, \mathit{true}, \ldots$)
	|	op	primitive ($+, \times, \mathit{fst}, \ldots$)
	|	$\texttt{fun}\ x \to e$	anonymous function
	|	$e\ e$	application
	|	(e, e)	pair
	|	$\texttt{let}\ x = e\ \texttt{in}\ e$	local binding

Even though it is called "mini", this language still contains all the complexity of a real programming language. In fact, the mere presence of variables, abstraction (*i.e.*, `fun`), and application is sufficient to capture any model of computation[1]. Mini-ML adds constants, primitive operations, pairs, and local variables to bring it closer to a typical programming language.

We intentionally remain vague about the set of constants and primitive operations. To be able to write interesting examples, we can assume that this includes at least integers and usual arithmetic operations. Thus, we can write the program:

$$\begin{array}{l}\texttt{let}\ \mathit{compose} = \texttt{fun}\ f \to \texttt{fun}\ g \to \texttt{fun}\ x \to f\ (g\ x)\ \texttt{in} \\ \texttt{let}\ \mathit{plus} = \texttt{fun}\ x \to \texttt{fun}\ y \to +\ (x, y)\ \texttt{in} \\ \mathit{compose}\ (\mathit{plus}\ 2)\ (\mathit{plus}\ 4)\ 36\end{array} \tag{2.1}$$

or the program:

$$\begin{array}{l}\texttt{let}\ \mathit{distr\_pair} = \texttt{fun}\ f \to \texttt{fun}\ p \to (f\ (\mathit{fst}\ p),\ f\ (\mathit{snd}\ p))\ \texttt{in} \\ \texttt{let}\ p = \mathit{distr\_pair}\ (\texttt{fun}\ x \to x)\ (40, 2)\ \texttt{in} \\ +\ (\mathit{fst}\ p,\ \mathit{snd}\ p)\end{array} \tag{2.2}$$

Exercise 13. What is the difference between $+$ and $\texttt{fun}\ x \to \texttt{fun}\ y \to +\ (x, y)$?

Solution □

The reader may be surprised not to find in this language a conditional construct of the type `if-then-else` or a recursive function. If we want such constructs in our programs, we can add them as special primitives. Thus, the conditional can be materialized by a primitive *opif* and associated with the following syntactic sugar:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \quad \stackrel{\text{def}}{=} \quad \mathit{opif}\ (e_1, ((\texttt{fun}\ \_ \to e_2), (\texttt{fun}\ \_ \to e_3)))$$

We note that the two branches e_2 and e_3 are represented by abstractions, in order to freeze the corresponding computation, in a sense that will be precisely defined by the semantics we are about to define.

[1] This is called the λ-calculus. The abstraction $\texttt{fun}\ x \to e$ is denoted there as $\lambda x.e$ and is called a λ-abstraction for this reason.

Similarly, recursion can be introduced by another primitive, *opfix*, also associated with syntactic sugar:

$$\texttt{rec}\ f\ x = e \quad \stackrel{\text{def}}{=} \quad \mathit{opfix}\ (\texttt{fun}\ f \to \texttt{fun}\ x \to e)$$

It may seem that introducing such essential constructions as the conditional or recursion as primitives is a clumsy way to limit the number of constructions in Mini-ML. But as we said earlier, abstraction and application are the truly fundamental constructions, containing the entire essence of computation. We will even show later that it is possible to *define* the primitives *opif* and *opfix*.

Free and Bound Variables. In the expression $\texttt{fun}\ x \to e$, the variable x is said to be *bound* in the expression e, and we say that the `fun` construct is a *binder*. The name of the variable matters less than the binding itself. Thus, one can write either $\texttt{fun}\ x \to +(x, 1)$ or $\texttt{fun}\ y \to +(y, 1)$, and these two expressions denote exactly the same abstraction. They are said to be α-equivalent.

Similarly, the `let` construct is a binder. In the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, the variable x is bound in the expression e_2. However, it is not bound in the expression e_1. A variable that is not bound is said to be *free*. The set of free variables of an expression can be defined recursively.

Definition 1 (free variables)**.** The set of *free variables* of an expression e, denoted by $\mathit{fv}(e)$, is defined recursively on e as follows:

$$\begin{array}{rcl}
\mathit{fv}(x) & = & \{x\} \\
\mathit{fv}(c) & = & \emptyset \\
\mathit{fv}(op) & = & \emptyset \\
\mathit{fv}(\texttt{fun}\ x \to e) & = & \mathit{fv}(e) \setminus \{x\} \\
\mathit{fv}(e_1\ e_2) & = & \mathit{fv}(e_1) \cup \mathit{fv}(e_2) \\
\mathit{fv}((e_1, e_2)) & = & \mathit{fv}(e_1) \cup \mathit{fv}(e_2) \\
\mathit{fv}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \mathit{fv}(e_1) \cup (\mathit{fv}(e_2) \setminus \{x\})
\end{array}$$

An expression without free variables is said to be *closed*. □

The removal of the variable x from the set calculated for the `fun` and `let` constructs precisely reflects the bound nature of this variable. In particular, we have correctly parenthesized the last case to express a binding in e_2 but not in e_1. Thus, we have:

$$\begin{array}{rcl}
\mathit{fv}\,(\texttt{let}\ x = +(20, 1)\ \texttt{in}\ (\texttt{fun}\ y \to +(y, y))\ x) & = & \emptyset \\
\mathit{fv}\,(\texttt{let}\ x = z\ \texttt{in}\ (\texttt{fun}\ y \to (x\ y)\ t)) & = & \{z, t\}
\end{array}$$

While we have the freedom to rename bound variables at will, we must be careful about potential conflicts with free variables. For example, in the expression:

$$\texttt{fun}\ x \to \texttt{let}\ y = +(x,\ x)\ \texttt{in}\ \times (x,\ y)$$

we cannot rename x to y in the first binder, nor y to x in the second, without changing the nature of the expression. This conflict of renaming between a free variable and a bound variable of the same name is known as *variable capture*.

2.2.1 Big-Step Semantics

Big-step semantics seeks to define a relation between an expression e and a *value* v, denoted as $e \twoheadrightarrow v$, whose meaning is "the evaluation of the expression e terminates with the value v". It is necessary to define what we mean here by *value*. It refers to the subset of expressions defined as follows:

$$\begin{array}{rcll} v & ::= & c & \text{constant} \\ & | & op & \text{unapplied primitive} \\ & | & \texttt{fun}\ x \to e & \text{function} \\ & | & (v, v) & \text{pair} \end{array}$$

This definition is inductive, as a pair is a value only if it is a pair of values. Note that an abstraction is always a value. Its body remains an expression, which reflects our intention not to evaluate under abstractions. Thus, the expression $\texttt{fun}\ x \to +(1,\ 2)$ is a value, even though the expression $+(1,\ 2)$ could be evaluated.

Substitution. To define big-step semantics, we need a *substitution* operation to replace, during an application, the formal parameter with the actual argument in the body of an abstraction.

Definition 2 (substitution)**.** If e is an expression, x a variable, and v a value, we denote $e[x \leftarrow v]$ as the *substitution* of x by v in e, defined recursively on e as follows:

$$\begin{array}{rcl} x[x \leftarrow v] & = & v \\ y[x \leftarrow v] & = & y \quad \text{if } y \neq x \\ c[x \leftarrow v] & = & c \\ op[x \leftarrow v] & = & op \\ (\texttt{fun}\ x \to e)[x \leftarrow v] & = & \texttt{fun}\ x \to e \\ (\texttt{fun}\ y \to e)[x \leftarrow v] & = & \texttt{fun}\ y \to e[x \leftarrow v] \quad \text{if } y \neq x \\ (e_1\ e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v]\ e_2[x \leftarrow v]) \\ (e_1, e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v], e_2[x \leftarrow v]) \\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)[x \leftarrow v] & = & \texttt{let}\ x = e_1[x \leftarrow v]\ \texttt{in}\ e_2 \\ (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[x \leftarrow v] & = & \texttt{let}\ y = e_1[x \leftarrow v]\ \texttt{in}\ e_2[x \leftarrow v] \\ & & \text{if } y \neq x \end{array}$$

In the case of the `fun` and `let` constructs, we can assume that $y \notin fv(v)$, otherwise the substitution would create a variable capture. If this is not the case, it is sufficient to rename the bound variable y by `fun` or `let`. □

Here are some examples of substitutions. Care should be taken to properly parenthesize expressions and binders.

$$\begin{array}{rcl} ((\texttt{fun}\ x \to +(x, x))\ x)\,[x \leftarrow 21] & = & (\texttt{fun}\ x \to +(x, x))\ 21 \\ (+(x, \texttt{let}\ x = 17\ \texttt{in}\ x))\,[x \leftarrow 3] & = & +(3, \texttt{let}\ x = 17\ \texttt{in}\ x) \\ (\texttt{fun}\ y \to y\ y)\,[y \leftarrow 17] & = & \texttt{fun}\ y \to y\ y \end{array}$$

Inference Rules. To define the relation $e \twoheadrightarrow v$ that interests us, we will use the concept of *inference rules* from logic. This involves defining a relation as the *smallest relation* satisfying a set of axioms of the form:

$$\frac{}{P}$$

and a set of implications of the form:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

For instance, we can define the relation $\mathsf{Even}(n)$, which characterizes even natural numbers, by the two rules:

$$\frac{}{\mathsf{Even}(0)} \qquad \text{and} \qquad \frac{\mathsf{Even}(n)}{\mathsf{Even}(n+2)}$$

These two rules should be read as:

$$\begin{array}{rl} \text{on the one hand} & \mathsf{Even}(0), \\ \text{and on the other hand} & \forall n.\ \mathsf{Even}(n) \Rightarrow \mathsf{Even}(n+2). \end{array}$$

We can easily convince ourselves that the smallest relation satisfying these two properties coincides with the property "n is an even natural number." On the one hand, even natural numbers are clearly included, by induction. On the other hand, if there were at least one odd integer, we could remove the smallest one, which would contradict the minimality of the relation.

Given a relation defined by inference rules, a *derivation* is a tree whose nodes correspond to the rules and the leaves to the axioms. With the previous example, we have, among others, the tree:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Even}(0)}}{\mathsf{Even}(2)}}{\mathsf{Even}(4)}$$

We can see this tree as a proof that $\mathsf{Even}(4)$ is true. More generally, the set of all possible derivations exactly characterizes the smallest relation satisfying the inference rules.

To establish a property of a relation defined by a set of inference rules, we can reason by *structural induction* on the derivation. This means that we can apply the induction hypothesis to any sub-derivation. Equivalently, we can say that we reason by induction on the height of the derivation. In practice, we reason by induction on the derivation and by case on the last rule used. Thus, assuming that n is a relative integer in the definition of $\mathsf{Even}(n)$ above, we can show $\forall n,\ \mathsf{Even}(n) \Rightarrow n \geq 0$ by induction on the derivation of $\mathsf{Even}(n)$.

Definition of Big-Step Semantics. We are now able to define the big-step semantics of Mini-ML. It is given in the form of six inference rules, one for each language construct other than a variable. The first three rules express that a constant, a primitive, and an abstraction are already values.

$$\frac{}{c \twoheadrightarrow c} \qquad \frac{}{op \twoheadrightarrow op} \qquad \frac{}{(\texttt{fun}\ x \to e) \twoheadrightarrow (\texttt{fun}\ x \to e)}$$

The following rule defines the evaluation of a pair. Very naturally, it expresses that each of the two components must be evaluated to a value and that the final value is the pair of these two values.

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)}$$

The last two rules are the most complex. They involve binders and the substitution operation. For the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, the semantics express that e_1 must evaluate to a value v_1, which is then substituted for x in e_2 before continuing the evaluation.

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \twoheadrightarrow v}$$

Similarly, the evaluation of an application expresses the fact that the argument must first be evaluated, then its value must be substituted for the formal parameter in the body of the abstraction before evaluating it.

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

These last two rules thus express the choice of a *call-by-value* strategy, where the argument is fully evaluated before the call. We could have made other choices, such as call by name or call by need[2].

To these rules must be added specific rules for each of the primitives. Thus, to ensure that the primitive $+$ effectively represents the addition of two integers, we add the following rule:

$$\frac{e_1 \twoheadrightarrow + \quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}$$

Here, n_1 and n_2 represent integer values, *i.e.*, constants that happen to be integers (as opposed to other constants of other types, such as *true* or *false*). The premise $n = n_1 + n_2$ means that n denotes the integer constant that is the sum of n_1 and n_2.

Figure 2.1 groups together all the rules defining the big-step semantics of Mini-ML, with the primitives *opif* and *opfix* introduced earlier and the primitives *fst* and *snd* for the two projections associated with pairs. The two rules (IF-TRUE) and (IF-FALSE) require, in particular, that the two branches evaluate to functions, whose bodies e_3 and e_4 are not evaluated. Thus, only the expression e_3 (resp. e_4) is evaluated when the test is true (resp. false). The last rule, called (FIX), expresses the fact that the primitive *opfix* represents a *fixed-point operator* satisfying the identity $opfix\ f = f\ (opfix\ f)$ for any function f.

Example. By applying the previous rules, we can show that the expression $\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \to +(y,y))\ x$ evaluates to the value 42. The derivation tree has the following form:

$$\dfrac{\dfrac{\dfrac{}{+ \twoheadrightarrow +} \quad \dfrac{\overline{20 \twoheadrightarrow 20} \quad \overline{1 \twoheadrightarrow 1}}{(20,1) \twoheadrightarrow (20,1)}}{+(20,1) \twoheadrightarrow 21} \quad \dfrac{\dfrac{}{\texttt{fun}\ldots \twoheadrightarrow \texttt{fun}\ldots} \quad \overline{21 \twoheadrightarrow 21} \quad \dfrac{\vdots}{+(21,21) \twoheadrightarrow 42}}{(\texttt{fun}\ y \to +(y,y))\ 21 \twoheadrightarrow 42}}{\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \to +(y,y))\ x \twoheadrightarrow 42}$$

(Some parts of the derivation have been omitted.)

Exercise 14. Give the derivation of the expression

$$(\texttt{rec}\ \mathit{fact}\ n = \texttt{if}\ =(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1)))\ 2$$

Recall that the constructs `rec` and `if-then-else` were introduced earlier as syntactic sugar for the primitives *opfix* and *opif*. Solution □

[2]These concepts will be introduced in chapter 7.

$$\frac{}{c \twoheadrightarrow c}(\text{CST}) \qquad \frac{}{op \twoheadrightarrow op}(\text{OP}) \qquad \frac{}{(\texttt{fun}\ x \to e) \twoheadrightarrow (\texttt{fun}\ x \to e)}(\text{ABS})$$

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1,\, e_2) \twoheadrightarrow (v_1,\, v_2)}(\text{PAIR}) \qquad \frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \twoheadrightarrow v}(\text{LET})$$

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{APP})$$

$$\frac{e_1 \twoheadrightarrow + \quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}(\text{ADD})$$

$$\frac{e_1 \twoheadrightarrow \mathit{fst} \quad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_1}(\text{FST}) \qquad \frac{e_1 \twoheadrightarrow \mathit{snd} \quad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_2}(\text{SND})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opif} \quad e_2 \twoheadrightarrow (\mathit{true}, ((\texttt{fun}\ \_ \to e_3), (\texttt{fun}\ \_ \to e_4))) \quad e_3 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{IF-TRUE})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opif} \quad e_2 \twoheadrightarrow (\mathit{false}, ((\texttt{fun}\ \_ \to e_3), (\texttt{fun}\ \_ \to e_4))) \quad e_4 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{IF-FALSE})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opfix} \quad e_2 \twoheadrightarrow (\texttt{fun}\ f \to e) \quad e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \to e)] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{FIX})$$

Figure 2.1: Big-step semantics of Mini-ML.

There are expressions e for which there is no value v such that $e \twoheadrightarrow v$. The simplest example is probably the expression 1 2, *i.e.*, the application of the constant 1 to the constant 2. Clearly, the rule (APP) does not apply because 1 does not evaluate to an abstraction. And among the rules specific to primitives, none deals with the case of an application where the function evaluates to an integer constant. We are not at all shocked by the fact that the expression 1 2 has no value; on the contrary, it is rather reassuring.

However, there are other less problematic expressions that have no value in big-step semantics. Consider the expression:

$$(\texttt{fun}\ x \to x\ x)\ (\texttt{fun}\ x \to x\ x) \tag{2.3}$$

i.e., the application of the expression $\texttt{fun}\ x \to x\ x$, which we will call Δ from now on, to itself. This is the application of an abstraction to a value, and the first step of computation is therefore the substitution of x by the argument Δ in the body of the function, *i.e.*, $x\ x$. We thus fall back exactly on the same expression. We understand that we have just found an expression whose evaluation does not terminate.

Exercise 15. Show that there is no value for the expression (2.3) in big-step semantics.
Solution □

Here is a first very simple result concerning our semantics.

Proposition 1 (evaluation produces closed values)**.** If $e \twoheadrightarrow v$ and if, moreover, e is closed, then v is also closed.

Proof. We proceed by induction on the derivation $e \twoheadrightarrow v$. Consider the case of an application, *i.e.*, a derivation whose conclusion is of the form:

$$\frac{\begin{array}{ccc}(D_1) & (D_2) & (D_3)\\ \vdots & \vdots & \vdots\\ e_1 \twoheadrightarrow (\texttt{fun}\ x \to e_3) & e_2 \twoheadrightarrow v_2 & e_3[x \leftarrow v_2] \twoheadrightarrow v\end{array}}{e_1\ e_2 \twoheadrightarrow v}$$

If e is closed, then the expressions e_1 and e_2 are also closed. By induction hypotheses applied to the derivations (D_1) and (D_2), the values $\texttt{fun}\ x \to e_3$ and v_2 are also closed. We deduce that the expression $e_3[x \leftarrow v_2]$ is closed, because x is the only possible free variable of e_3, and therefore that v is closed by the induction hypothesis applied to (D_3).

The other cases are left to the reader. □

Another less obvious property of our semantics is that it is deterministic, meaning that an expression has only one possible value.

Proposition 2 (determinism of evaluation)**.** If $e \twoheadrightarrow v$ and $e \twoheadrightarrow v'$ then $v = v'$.

Proof. We proceed by induction on the derivations $e \twoheadrightarrow v$ and $e \twoheadrightarrow v'$. Let's examine the case of a pair $e = (e_1, e_2)$. We therefore have two derivations of the following form:

$$\frac{\begin{array}{cc}(D_1) & (D_2)\\ \vdots & \vdots\\ e_1 \twoheadrightarrow v_1 & e_2 \twoheadrightarrow v_2\end{array}}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad \frac{\begin{array}{cc}(D'_1) & (D'_2)\\ \vdots & \vdots\\ e_1 \twoheadrightarrow v'_1 & e_2 \twoheadrightarrow v'_2\end{array}}{(e_1, e_2) \twoheadrightarrow (v'_1, v'_2)}$$

By induction hypotheses, we have $v_1 = v_1'$ and $v_2 = v_2'$. We deduce that

$$v = (v_1, v_2) = (v_1', v_2') = v'.$$

The other cases are left to the reader. □

It is important to note that the determinism of evaluation is in no way a necessity. We could, for example, imagine adding a primitive *random* and the rule:

$$\frac{e_1 \twoheadrightarrow \mathit{random} \quad e_2 \twoheadrightarrow n_1 \quad 0 \leq n < n_1}{e_1\ e_2 \twoheadrightarrow n}$$

We would then have $\mathit{random}\ 2 \twoheadrightarrow 0$ as well as $\mathit{random}\ 2 \twoheadrightarrow 1$.

Revisiting the Primitives *opif* and *opfix*. Now that we know the semantics of Mini-ML, we can show how to define the primitives *opif* and *opfix*. Recall that we defined the conditional as:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \stackrel{\text{def}}{=} \mathit{opif}\ (e_1, ((\texttt{fun}\ \_ \to e_2), (\texttt{fun}\ \_ \to e_3)))$$

From this, we can define the boolean constants and the primitive *opif* as follows:

$$\begin{aligned} \mathit{true} &\stackrel{\text{def}}{=} \texttt{fun}\ x \to \texttt{fun}\ y \to x\ () \\ \mathit{false} &\stackrel{\text{def}}{=} \texttt{fun}\ x \to \texttt{fun}\ y \to y\ () \\ \mathit{opif} &\stackrel{\text{def}}{=} \texttt{fun}\ p \to \mathit{fst}\ p\ (\mathit{fst}\ (\mathit{snd}\ p))\ (\mathit{snd}\ (\mathit{snd}\ p)) \end{aligned}$$

Here, () denotes a constant whose value is not significant; we could have used an integer. The idea is simple: when e_1 evaluates to *true* (resp. *false*), we apply the function whose body is e_2 (resp. e_3). We could even simplify further, to avoid pairs, by defining $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ from the start as $e_1\ (\texttt{fun}\ \_ \to e_2)\ (\texttt{fun}\ \_ \to e_3)$.

Similarly, it is possible to give a definition to the *opfix* operator, which we had introduced to define recursive functions, as follows:

$$\mathit{opfix} \stackrel{\text{def}}{=} \texttt{fun}\ f \to (\texttt{fun}\ x \to f\ (\texttt{fun}\ y \to x\ x\ y))\ (\texttt{fun}\ x \to f\ (\texttt{fun}\ y \to x\ x\ y))$$

The body of this function strangely resembles the term $\Delta\ \Delta$, of which we have shown that the evaluation does not terminate.

2.2.2 Small-Step Semantics

Natural semantics does not allow distinguishing expressions whose computation "crashes," such as 1 2, from expressions whose evaluation does not terminate, such as $\Delta\ \Delta$. This is quite problematic because when we focus on static typing in chapter 5, we will seek to show that well-typed programs evaluate without crashing. And obviously, we will not want to reject programs at typing because they are likely not to terminate.

Operational *small-step* semantics addresses this problem by introducing a notion of elementary computation step, denoted as $e_1 \to e_2$, which we will iterate. It then becomes possible to distinguish three situations: either the iteration results in a value:

$$e \to e_1 \to e_2 \to \cdots \to v$$

or the iteration gets stuck on an irreducible e_n that is not a value:

$$e \to e_1 \to e_2 \to \cdots \to e_n$$

or finally, the iteration does not terminate:

$$e \to e_1 \to e_2 \to \cdots$$

To define the relation $\to$, we start by defining a simpler relation $\xrightarrow{\epsilon}$, corresponding to a "head" reduction, *i.e.*, involving the outermost construction of the expression. There are, in particular, two head reduction rules corresponding to the binding of a value to a variable:

$$\begin{array}{rcl} (\texttt{fun}\ x \to e)\ v & \xrightarrow{\epsilon} & e[x \leftarrow v] \\ \texttt{let}\ x = v\ \texttt{in}\ e & \xrightarrow{\epsilon} & e[x \leftarrow v] \end{array}$$

These two rules reflect the choice of a *call-by-value* strategy, as in big-step semantics. We also give head reduction rules for the primitives:

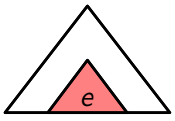
$$\begin{array}{rcll} +\ (n_1,\, n_2) & \xrightarrow{\epsilon} & n & \text{with } n = n_1 + n_2 \\ \\ \mathit{fst}\ (v_1,\, v_2) & \xrightarrow{\epsilon} & v_1 & \\ \mathit{snd}\ (v_1,\, v_2) & \xrightarrow{\epsilon} & v_2 & \\ \\ \mathit{opfix}\ (\texttt{fun}\ f \to e) & \xrightarrow{\epsilon} & e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \to e)] & \\ \\ \mathit{opif}\ (\mathit{true},\, ((\texttt{fun}\ \_ \to e_1),\, (\texttt{fun}\ \_ \to e_2))) & \xrightarrow{\epsilon} & e_1 & \\ \mathit{opif}\ (\mathit{false},\, ((\texttt{fun}\ \_ \to e_1),\, (\texttt{fun}\ \_ \to e_2))) & \xrightarrow{\epsilon} & e_2 & \end{array}$$

We then need to explain how the computation can be performed "in depth." If we consider, for example, the expression $\texttt{let}\ x = +(1,\, 2)\ \texttt{in}\ +(x,\, x)$, none of the above rules apply because it is not directly the addition of two integers. We must first reduce the sub-expression $+(1,\, 2)$ before we can obtain the head reduction of $\texttt{let}\ x = 3\ \texttt{in}\ +(x,\, x)$.

To designate a sub-expression such as $+(1,\, 2)$ in the example above, we introduce the notion of *context*. A context E is defined by the following grammar:

$$\begin{array}{rcl} E & ::= & \square \\ & \mid & E\ e \\ & \mid & v\ E \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ e \\ & \mid & (E,\, e) \\ & \mid & (v,\, E) \end{array}$$

A context is a "term with a hole," the latter being represented by the symbol $\square$. As can be seen, a context E contains exactly one hole. We then define $E(e)$ as the context E in which the hole has been replaced by the expression e, the result being an expression. This can be illustrated as follows:

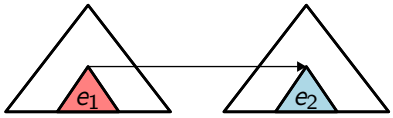


The white part represents the context E. In the example given above, the context of the sub-expression $+(1, 2)$ is $\texttt{let}\ x = \square\ \texttt{in}\ +(x, x)$.

We can now define small-step semantics, *i.e.*, the relation $\rightarrow$, using the notion of context to perform head reductions in any context. A single rule suffices for this:

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

It expresses that a head reduction $e_1 \xrightarrow{\epsilon} e_2$ can be performed at the location specified by the context E. This can be illustrated as follows:



The white part still represents the context E. If we take the example of the expression $\texttt{let}\ x = +(1, 2)\ \texttt{in}\ +(x, x)$, we can show that it evaluates to the value 6 in three reduction steps as follows:

$$\begin{array}{lll} & \texttt{let}\ x = +(1, 2)\ \texttt{in}\ +(x, x) & \\ \rightarrow & \texttt{let}\ x = 3\ \texttt{in}\ +(x, x) & \text{with } E = \texttt{let}\ x = \square\ \texttt{in}\ +(x, x) \\ \rightarrow & +(3, 3) & \text{with } E = \square \\ \rightarrow & 6 & \text{with } E = \square \end{array}$$

At each step, a head reduction is used (respectively, addition, the `let`, and again addition) associated with a context indicated to the right of the reduction. The set of rules for small-step semantics is given in figure 2.2.

Exercise 16. Give the reduction steps of the expression

$$(\texttt{rec}\ \mathit{fact}\ n = \texttt{if}\ =(n, 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n, \mathit{fact}\ (+(n, -1)))) \ 2$$

Recall that the constructs `rec` and `if-then-else` were introduced in the previous section as syntactic sugar for the primitives *opfix* and *opif*. Solution □

We observe that contexts have a very specific form, which reflects a left-to-right evaluation order. Indeed, we have the context $E\ e$ that allows reducing to the left of an application, but we only have the context $v\ E$ to reduce to the right of an application. Thus, we are forced to reduce the left member of an application first. Similarly, the contexts for evaluating a pair, respectively (E, e) and (v, E), force us to evaluate the left component first. Thus, $(+(1, 2), \square)$ is not a valid evaluation context.

We could very well have made another choice, such as evaluating from right to left, or even not fixing an evaluation order by proposing both the context $E\ e$ and the context $e\ E$. In the case of Mini-ML, this would not make a difference in the value of an expression, but in a more complex language with side effects or exceptional behaviors, evaluating in one order rather than another can make a difference. We will return to this in section 2.4.

Definition 3 (evaluation and normal form)**.** We denote $\xrightarrow{\star}$ the reflexive and transitive closure of the relation $\rightarrow$, *i.e.*, $e_1 \xrightarrow{\star} e_2$ if and only if e_1 reduces to e_2 in zero, one, or several steps. We call a *normal form* any expression e such that there is no expression e' such that $e \rightarrow e'$. □

In an obvious way, values are normal forms. The normal forms that are not values are erroneous expressions, such as 1 2.

head reductions

$$(\texttt{fun}\ x \to e)\ v \xrightarrow{\epsilon} e[x \leftarrow v]$$
$$\texttt{let}\ x = v\ \texttt{in}\ e \xrightarrow{\epsilon} e[x \leftarrow v]$$

$$+\ (n_1, n_2) \xrightarrow{\epsilon} n \qquad \text{with } n = n_1 + n_2$$

$$\mathit{fst}\ (v_1, v_2) \xrightarrow{\epsilon} v_1$$
$$\mathit{snd}\ (v_1, v_2) \xrightarrow{\epsilon} v_2$$

$$\mathit{opfix}\ (\texttt{fun}\ f \to e) \xrightarrow{\epsilon} e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \to e)]$$

$$\mathit{opif}\ (\mathit{true}, ((\texttt{fun}\ \_ \to e_1), (\texttt{fun}\ \_ \to e_2))) \xrightarrow{\epsilon} e_1$$
$$\mathit{opif}\ (\mathit{false}, ((\texttt{fun}\ \_ \to e_1), (\texttt{fun}\ \_ \to e_2))) \xrightarrow{\epsilon} e_2$$

reduction contexts

$$\begin{array}{rcl} E & ::= & \square \\ & | & E\ e \\ & | & v\ E \\ & | & \texttt{let}\ x = E\ \texttt{in}\ e \\ & | & (E, e) \\ & | & (v, E) \end{array}$$

reduction in a context

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \to E(e_2)}$$

Figure 2.2: Small-step semantics of Mini-ML.

2.2.3 Equivalence of the Two Semantics

We will show that the two operational semantics defined above, big-step and small-step, are equivalent for expressions whose evaluation terminates with a value, *i.e.*,

$$e \twoheadrightarrow v \quad \text{if and only if} \quad e \xrightarrow{\star} v$$

Let's start with the direction "big-step implies small-step." We need the following lemma, which states that small-step reductions can be performed in depth.

Lemma 1 (context passage of reductions)**.** Suppose $e \to e'$. Then for any expression e_2 and any value v, we have

1. $e\ e_2 \to e'\ e_2$
2. $v\ e \to v\ e'$
3. $\texttt{let}\ x = e\ \texttt{in}\ e_2 \to \texttt{let}\ x = e'\ \texttt{in}\ e_2$
4. $(e, e_2) \to (e', e_2)$
5. $(v, e) \to (v, e')$

Proof. From $e \to e'$, we know there exists a context E such that

$$e = E(r) \qquad e' = E(r') \qquad r \xrightarrow{\epsilon} r'$$

Consider the context $E_1 \stackrel{\text{def}}{=} E\ e_2$; then

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \to E_1(r')} \qquad \text{i.e.} \qquad \frac{r \xrightarrow{\epsilon} r'}{e\ e_2 \to e'\ e_2}$$

We proceed similarly for cases 2 to 5. □

We can now show the first implication.

Proposition 3 (big-step implies small-step)**.** If $e \twoheadrightarrow v$, then $e \xrightarrow{\star} v$.

Proof. We proceed by induction on the derivation of $e \twoheadrightarrow v$. Suppose that the last rule is that of a function application:

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

By induction hypotheses for each of the three derivations in the premises, and by denoting v_1 as the value $\texttt{fun}\ x \to e_3$, we have the three small-step evaluations:

$$\begin{aligned} &e_1 \to \cdots \to v_1 \\ &e_2 \to \cdots \to v_2 \\ &e_3[x \leftarrow v_2] \to \cdots \to v \end{aligned}$$

By context passage (previous lemma), we also have the three evaluations:

$$\begin{aligned} &e_1\ e_2 \to \cdots \to v_1\ e_2 \\ &v_1\ e_2 \to \cdots \to v_1\ v_2 \\ &e_3[x \leftarrow v_2] \to \cdots \to v \end{aligned}$$

By inserting the reduction:

$$(\texttt{fun}\ x \to e_3)\ v_2 \xrightarrow{\epsilon} e_3[x \leftarrow v_2]$$

between the second and third lines, we obtain the complete reduction:

$$e_1\ e_2 \to \cdots \to v$$

The other cases are left to the reader. □

To show the other implication, *i.e.*, the direction "small-step implies big-step," we start by establishing two lemmas. The first one is obvious.

Lemma 2 (values are already evaluated)**.** $v \twoheadrightarrow v$ for any value v.

The second one indicates that a small step followed by a big step is a big step.

Lemma 3 (reduction and evaluation)**.** If $e \to e'$ and $e' \twoheadrightarrow v$, then $e \twoheadrightarrow v$.

Proof. We start with head reductions, *i.e.*, $e \xrightarrow{\epsilon} e'$. Suppose, for example, that $e = (\texttt{fun}\ x \to e_1)\ v_2$ and $e' = e_1[x \leftarrow v_2]$. We can construct the derivation:

$$\frac{(\texttt{fun}\ x \to e_1) \twoheadrightarrow (\texttt{fun}\ x \to e_1) \quad v_2 \twoheadrightarrow v_2 \quad e_1[x \leftarrow v_2] \twoheadrightarrow v}{(\texttt{fun}\ x \to e_1)\ v_2 \twoheadrightarrow v}$$

using the previous lemma ($v_2 \twoheadrightarrow v_2$) and the hypothesis $e' \twoheadrightarrow v$. We similarly handle the other cases of a head reduction.

Now, let's show that if $e \xrightarrow{\epsilon} e'$ and $E(e') \twoheadrightarrow v$, then $E(e) \twoheadrightarrow v$ for any context E, by structural induction on E (or, if preferred, by induction on the height of the context). We have just done the base case $E = \Box$.

Consider, for example, the case $E = E'\ e_2$. We have $E(e') \twoheadrightarrow v$, *i.e.*, $E'(e')\ e_2 \twoheadrightarrow v$. In the case of an abstraction, this reduction has the form:

$$\frac{E'(e') \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e')\ e_2 \twoheadrightarrow v}$$

By the induction hypothesis, we have $E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3)$ and therefore:

$$\frac{E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e)\ e_2 \twoheadrightarrow v}$$

i.e., $E(e) \twoheadrightarrow v$. The other cases are left to the reader. □

We can then deduce the second direction of the equivalence.

Proposition 4 (small-step implies big-step)**.** If $e \xrightarrow{\star} v$, then $e \twoheadrightarrow v$.

Proof. Write the small-step reduction in the form $e \to e_1 \to \cdots \to e_n \to v$. We have $v \twoheadrightarrow v$ and therefore, by the previous lemma, $e_n \twoheadrightarrow v$. Similarly, $e_{n-1} \twoheadrightarrow v$ and so on until $e \twoheadrightarrow v$. □

2.3 Interpreter

An *interpreter* is a program that executes another program on given inputs. It differs from a compiler in that the work is redone each time, for every new program or new inputs. The compiler, on the other hand, builds a program once and for all, which is then executed by the machine on any input.

We can program an interpreter by following the rules of the semantics[3]. We define a type for the abstract syntax of the language and write a function that, given an expression in the language, computes its value. Essentially, this is what exercise 12 page 23 proposed to do on a minimal language of arithmetic expressions. Let's write here an interpreter for Mini-ML, in the OCaml language, which follows the natural semantics of figure 2.1 page 29.

We start by defining an OCaml type for the abstract syntax of Mini-ML.

```
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

Constants are here limited to integers and primitives are represented by character strings. We then implement the substitution operation $e[x \leftarrow v]$ for a closed value v. Since there is no possible variable capture, the definition remains simple[4].

```
let rec subst e x v = match e with
  | Var y ->
      if y = x then v else e
  | Const _ | Op _ ->
      e
  | Fun (y, e1) ->
      if y = x then e else Fun (y, subst e1 x v)
  | App (e1, e2) ->
      App (subst e1 x v, subst e2 x v)
  | Pair (e1, e2) ->
      Pair (subst e1 x v, subst e2 x v)
  | Let (y, e1, e2) ->
      Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
```

Finally, we write the interpreter in the form of a function `eval` that takes as an argument a closed expression and returns its value. For a constant, a primitive, or an abstraction, the expression is already a value.

[3] We can also define the semantics of a language by providing an interpreter, which is then called a *reference interpreter*.

[4] Without the hypothesis that v is closed, we may need to rename the binders under which the value v is substituted.

```
let rec eval = function
  | Const _ | Op _ | Fun _  as v ->
      v
```

For a pair, it is sufficient to evaluate each of the components.

```
  | Pair (e1, e2) ->
      Pair (eval e1, eval e2)
```

For an expression of the form `let x = e1 in e2`, we start by evaluating `e1`, then substitute its value in `e2` and continue with the evaluation of this new expression.

```
  | Let(x, e1, e2) ->
      eval (subst e2 x (eval e1))
```

Since we evaluate a closed expression, the value of `e1` is a closed expression (by property 1 page 30) and we can therefore substitute it with `subst`. Finally, it remains to evaluate the application. Here, we limit ourselves, apart from the case of applying an abstraction, to the primitives + and *fst* and `snd`.

```
  | App (e1, e2) ->
      begin match eval e1 with
      | Fun (x, e) ->
          eval (subst e x (eval e2))
      | Op "+" ->
          let (Pair (Const n1, Const n2)) = eval e2 in
          Const (n1 + n2)
      | Op "fst" ->
          let (Pair(v1, v2)) = eval e2 in v1
      | Op "snd" ->
          let (Pair(v1, v2)) = eval e2 in v2
      end
```

The pattern matching here is intentionally non-exhaustive. In particular, our interpreter fails if the argument of + does not evaluate to a pair of integers, if the argument of *fst* or *snd* is not a pair, or if the expression `e1` evaluates to something other than an abstraction or one of these three primitives[5].

```
# eval (App (Const 1, Const 2));;
Exception: Match_failure ("", 87, 6).
```

In this sense, our interpreter performs *dynamic typing*, as opposed to the static typing we will see in chapter 5. Our interpreter can also fail to terminate, for example on (`fun` $x \to x\ x$) (`fun` $x \to x\ x$).

```
# let b = Fun ("x", App (Var "x", Var "x")) in
  eval (App (b, b));;
Interrupted.
```

Exercise 17. Add the primitives *opif* and *opfix* to this interpreter. Solution □

[5] Of course, we could program this failure more cleanly, with an `option` type or a more explicit exception.

A More Efficient Interpreter. Our Mini-ML interpreter is not very efficient because it spends its time performing substitutions and thus reconstructing expressions. A simple and natural idea to avoid these substitutions is to maintain the values of known variables in a dictionary. Such a dictionary is called an *environment*. Our `eval` function will therefore take the following type:

```
val eval: environment -> expression -> value
```

The `environment` type can be easily implemented in a purely applicative manner using the dictionaries from the `Map` module of OCaml's standard library:

```
module Smap = Map.Make(String)
type environment = value Smap.t
```

However, there is a difficulty with this approach. If we evaluate the expression:

$$\texttt{let } x = 1 \texttt{ in fun } y \to +(x, y)$$

The result is a function that must "remember" that $x = 1$. In our first interpreter, this was ensured by substituting x with 1 in $\texttt{fun } y \to +(x, y)$. But since we are precisely trying to avoid substitution, we need to find another way to remember the value of x in $\texttt{fun } y \to +(x, y)$.

The solution is to attach an environment to any value of the form $\texttt{fun } y \to e$. This is called a *closure*[6]. We therefore define a new `value` type for values:

```
type value =
  | Vconst of int
  | Vop    of string
  | Vpair  of value * value
  | Vfun   of string * environment * expression
and environment = value Smap.t
```

We can now write the `eval` function that computes the value of an expression in a given environment. We start with the simple cases of an expression that is already a value:

```
let rec eval env = function
  | Const n ->
      Vconst n
  | Op op ->
      Vop op
  | Fun (x, e) ->
      Vfun (x, env, e)
```

In the latter case, we save the environment `env` in the value of type `Vfun`. For this reason, it is important to have chosen a purely applicative structure here to represent environments. For a pair, it is sufficient to evaluate its two components:

```
  | Pair (e1, e2) ->
      Vpair (eval env e1, eval env e2)
```

The main change compared to the previous interpreter lies in the construction and use of the environment. We add the value of a variable when we encounter a `let` construct:

[6] We will discuss this again in Chapter 8 when we *compile* languages like Mini-ML.

```
  | Let (x, e1, e2) ->
      eval (Smap.add x (eval env e1) env) e2
```

And we consult it when the expression is a variable:

```
  | Var x ->
      Smap.find x env
```

Finally, there is the case of an application $e_1\ e_2$. As in the previous interpreter, we start by evaluating e_1 and examining its value. The interesting case is that of applying a function `fun` $x \to e$. The environment contained in the closure `Vfun` is retrieved, extended with the value of x, *i.e.*, the value of e_2, and then the body e of the function is evaluated in this environment:

```
  | App (e1, e2) ->
      begin match eval env e1 with
      | Vfun (x, clos, e) ->
          eval (Smap.add x (eval env e2) clos) e
```

We use the fact that a variable y appearing in e is either the variable x or a variable whose value is necessarily given by the environment `clos`, since we are evaluating a closed expression. The other cases of application correspond to primitives and do not differ from what we had written previously:

```
      | Vop "+" ->
          let Vpair (Vconst n1, Vconst n2) = eval env e2 in
          Vconst (n1 + n2)
      | Vop "fst" ->
          let Vpair (v1, _) = eval env e2 in v1
      | Vop "snd" ->
          let Vpair (_, v2) = eval env e2 in v2
      end
```

Exercise 18. Explain why it would be difficult to implement the environment with an imperative structure, for example, `type environment = value Hashtbl.t`.

Solution □

2.4 Imperative Languages

We can define an operational semantics, either big-step or small-step, for a language with imperative features such as mutable variables, arrays, exceptions, etc. Let's take the example of a small imperative language with mutable variables, which we will assume are only global and fixed in advance for simplicity. The abstract syntax of expressions is:

$$\begin{array}{rcll} e & ::= & c & \text{constant } (\texttt{true}, 42, \ldots) \\ & | & x & \text{global variable} \\ & | & e\ op\ e & \text{binary operator } (+, <, \ldots) \end{array}$$

To define the semantics of an expression, we need to provide a *state* in which to evaluate this expression. A state here is a function E that associates each variable with the value

Undefined Behavior. As we said earlier, there are expressions in our language whose semantics is not defined, such as 1 2 or, more subtly, any expression whose evaluation would lead to trying to apply a value that is not a function. This is referred to as *undefined behavior* (abbreviated as UB). In our interpreter, we chose to fail with an exception (in this case, `Match_failure`) when encountering an expression whose semantics is not defined. This is quite natural, but we could just as well have continued the computation arbitrarily, leaving the programmer responsible for not using undefined behaviors.

Similarly, a compiler is free to act as it sees fit in the case of undefined behavior. It can produce code that will lead to a failure, but just as well code that will give an arbitrary result. For example, an overflow in signed arithmetic is undefined behavior in the C language, and a C compiler can therefore optimize the test `x+1<10` by compiling it as the test `x<9`. If the value of `x` happens to be the largest integer, then the test `x<9` is negative where the computation of `x+1` would have given a negative value and a test `x+1<10` would be positive. The behavior "changes," but since it is not defined, the compiler is not at fault.

It is important to understand that undefined behavior is different from *unspecified behavior*, such as the order of evaluation of parameters in C or OCaml (see section 7.1).

Similarly, undefined behavior is different from a *programming error*, such as modifying a mutable data structure during its traversal or making unsynchronized concurrent accesses to it. In these cases, the code is correctly compiled, with a well-defined semantics, even if the observed result is probably not at all what is expected.

it contains. The reduction relation, whether big-step or small-step, then no longer relates only expressions but state/expression pairs. Thus, the big-step semantics will take the form:

$$E,\, e \twoheadrightarrow v$$

where v denotes the value of the expression e in the state E. Such a relation is easily defined with rules such as:

$$\frac{}{E, c \twoheadrightarrow c} \qquad \frac{}{E, x \twoheadrightarrow E(x)} \qquad \frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2}{E, e_1 + e_2 \twoheadrightarrow n} \qquad \text{etc.}$$

We could also choose a small-step semantics, but it is unnecessary because the evaluation of an expression always terminates.

We now provide an abstract syntax for statements[7]:

s	::=	$x \leftarrow e$	assignment
	|	`if` e `then` s `else` s	conditional
	|	`while` e `do` s	loop
	|	s; s	sequence
	|	`skip`	do nothing

Since the evaluation of a statement may not terminate, we will provide a small-step semantics for statements, in the form of a relation:

$$E,\, s \to E'$$

[7] Such a language is known as a *while language*.

where E and E' represent the state at the beginning and at the end of the evaluation of the statement s. The rules defining this relation are as follows:

$$\frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \to E\{x \mapsto v\}, \texttt{skip}}$$

$$\frac{}{E, \texttt{skip}; s \to E, s} \qquad \frac{E, s_1 \to E_1, s_1'}{E, s_1; s_2 \to E_1, s_1'; s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to E, s_1} \qquad \frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to E, s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{while } e \texttt{ do } s \to E, s; \texttt{while } e \texttt{ do } s}$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while } e \texttt{ do } s \to E, \texttt{skip}}$$

Note in particular how the `while` loop is "unfolded" when its test evaluates to `true`.

Finally, we can say that the evaluation of a statement s terminates in a state E if we have:

$$E, s \to^\star E', \texttt{skip}$$

for some state E'.

Exercise 19. Provide a big-step semantics for the evaluation of statements. Solution □

2.5 Proof of Compiler Correctness

Formal semantics is a powerful tool that can be used, among other things, to demonstrate the correctness of a compiler. By this, we mean that if the source language is equipped with a semantics $\to_s$ and the machine language with a semantics $\to_m$, and if the expression e is compiled into $C(e)$, then we must have a "commutative diagram" of the form:

$$\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$$

where $v \approx v'$ expresses that the values v and v' coincide.

Let's illustrate this with the minimalist example of an arithmetic expression language containing only integer constants and additions:

$$e ::= n \mid e + e$$

Our compiler here produces x86-64 assembly code intended to eventually place the value

of the expression in the `%rdi` register, using the stack to store intermediate results:

$$\begin{aligned} code(n) &= \texttt{movq}\ \$n, \texttt{\%rdi} \\ code(e_1 + e_2) &= code(e_1) \\ &\quad \texttt{addq}\ \ \$-8, \texttt{\%rsp} \\ &\quad \texttt{movq}\ \ \texttt{\%rdi}, (\texttt{\%rsp}) \\ &\quad code(e_2) \\ &\quad \texttt{movq}\ \ (\texttt{\%rsp}), \texttt{\%rsi} \\ &\quad \texttt{addq}\ \ \$8, \texttt{\%rsp} \\ &\quad \texttt{addq}\ \ \texttt{\%rsi}, \texttt{\%rdi} \end{aligned}$$

To formally show the correctness of this micro-compiler, we start by defining a semantics for each of the two languages. Here, we opt for small-step operational semantics. For the source language, it is simply defined with the following notions of value and context:

$$\begin{aligned} v &::= n \\ E &::= \square \mid E + e \mid v + E \end{aligned}$$

and the unique head reduction rule:

$$n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{with } n = n_1 + n_2.$$

For the target language, here x86-64 assembly, we start by introducing the notions of instruction m and register r, limiting ourselves here to the instructions and registers used by the compiler:

$$\begin{aligned} m &::= \texttt{movq}\ \$n, r \\ &\quad \mid \texttt{addq}\ \$n, r \mid \texttt{addq}\ r, r \\ &\quad \mid \texttt{movq}\ (r), r \mid \texttt{movq}\ r, (r) \mid \\ r &::= \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp} \end{aligned}$$

Since this is an imperative language, we define a state as the values of the three registers on the one hand, denoted as R, and a memory state on the other hand, denoted as M:

$$\begin{aligned} R &::= \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\ M &::= \mathbb{N} \to \mathbb{Z} \end{aligned}$$

The operational semantics of an instruction m can then be defined by a reduction of the form:

$$R, M, m \xrightarrow{m} R', M'$$

where R, M denotes the state before the execution of the instruction and R', M' the state after execution. This reduction is defined without difficulty, with one rule for each instruction:

$$\begin{aligned} R, M, \texttt{movq}\ \$n, r &\xrightarrow{m} R\{r \mapsto n\}, M \\ R, M, \texttt{addq}\ \$n, r &\xrightarrow{m} R\{r \mapsto R(r) + n\}, M \\ R, M, \texttt{addq}\ r_1, r_2 &\xrightarrow{m} R\{r_2 \mapsto R(r_1) + R(r_2)\}, M \\ R, M, \texttt{movq}\ (r_1), r_2 &\xrightarrow{m} R\{r_2 \mapsto M(R(r_1))\}, M \\ R, M, \texttt{movq}\ r_1, (r_2) &\xrightarrow{m} R, M\{R(r_2) \mapsto R(r_1)\} \end{aligned}$$

We can now show the correctness of the compilation. Specifically, we want to show that:

$$\text{if } e \xrightarrow{\star} n \text{ and } R, M, \mathit{code}(e) \xrightarrow{m}{}^{\star} R', M' \text{ then } R'(\texttt{\%rdi}) = n.$$

We naturally proceed by structural induction on e. Unfortunately, the proof does not go through because, when compiling $e_1 + e_2$, the induction hypothesis on e_2 does not allow us to ensure that the value of e_1 stored on the stack has not been corrupted by the compilation of e_2. We therefore need to establish a stronger result, namely:

$$\text{if } e \xrightarrow{\star} n \text{ and } R, M, \mathit{code}(e) \xrightarrow{m}{}^{\star} R', M' \text{ then } \begin{cases} R'(\texttt{\%rdi}) = n \\ R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M'(a) = M(a) \end{cases}$$

The case $e = n$ is trivially established, since we have $e \xrightarrow{\star} n$ and $\mathit{code}(e) = \texttt{movq}\ \$n, \texttt{\%rdi}$. In the case where $e = e_1 + e_2$, we have $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2$ with $e_1 \xrightarrow{\star} n_1$ and $e_2 \xrightarrow{\star} n_2$, which allows us to invoke the induction hypothesis on e_1 and e_2. The proof is then conducted with the following steps:

code	state	justification
$\mathit{code}(e_1)$	R_1, M_1	by induction hypothesis $R_1(\texttt{\%rdi}) = n_1$ and $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}),\ M_1(a) = M(a)$
`addq` $\$-8, \texttt{\%rsp}$ `movq` $\texttt{\%rdi}, (\texttt{\%rsp})$	R'_1, M'_1	$R'_1 = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M'_1 = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$\mathit{code}(e_2)$	R_2, M_2	by induction hypothesis $R_2(\texttt{\%rdi}) = n_2$ and $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8,\ M_2(a) = M'_1(a)$
`movq` $(\texttt{\%rsp}), \texttt{\%rsi}$ `addq` $\$8, \texttt{\%rsp}$ `addq` $\texttt{\%rsi}, \texttt{\%rdi}$	R', M_2	$R'(\texttt{\%rdi}) = n_1 + n_2$ $R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8 + 8 = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp})$, $M_2(a) = M'_1(a) = M_1(a) = M(a)$

This concludes the proof of our micro-compiler.

Bibliographical Notes. Operational semantics was introduced by Gordon Plotkin in 1981 [24]. An excellent introductory book on operational semantics is *Types and Programming Languages* by Benjamin Pierce [23]. It also covers interpreters and typing—we will discuss typing later, in chapter 5. Mini-ML was introduced by Gilles Kahn and his colleagues in a 1985 paper [12]. The CompCert compiler is a C compiler whose correctness has been verified with the Rocq proof assistant. CompCert was designed and developed by Xavier Leroy [18].

There are many other ways to define the semantics of a programming language. For example, one can choose an *axiomatic semantics* that characterizes the constructions of

the language by the logical properties they allow to establish. The most well-known axiomatic semantics is *Hoare logic*, introduced in the now famous article *An axiomatic basis for computer programming* [11]. This article introduces the notion of a triple $\{P\}\ i\ \{Q\}$ meaning "if the formula P is true before the execution of the instruction i, then the formula Q will be true after," as well as a set of deduction rules to establish such triples. The rule for assignment, for example, is stated as follows:

$$\{P[x \leftarrow E]\}\ x := E\ \{P(x)\}$$

With this rule, one can establish the validity of the triple $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$, assuming that the addition does not cause an arithmetic overflow.

Denotational semantics associates with each program expression e its denotation $[\![e]\!]$, which is a mathematical object representing the computation designated by e. If we take the example of very simple arithmetic expressions with *only one variable* `x`, *i.e.*,

$$e ::= \texttt{x} \mid n \mid e \texttt{+} e \mid e \texttt{*} e \mid \ \ldots$$

the denotation can be a function that associates the value of the variable `x` with the value of the expression, *i.e.*,

$$\begin{array}{rcl} [\![\texttt{x}]\!] & = & x \mapsto x \\ [\![n]\!] & = & x \mapsto n \\ [\![e_1 \texttt{+} e_2]\!] & = & x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\ [\![e_1 \texttt{*} e_2]\!] & = & x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x) \end{array}$$

Finally, let's also mention *translational semantics*, also called Strachey-style denotational semantics [26]. It consists of defining the semantics of a language by translating it into a language whose semantics is already known.

Part II

Front End of the Compiler

3 Lexical Analysis

Lexical analysis involves breaking down the source text into "words." Just as in natural languages, this tokenization facilitates the work of the next phase, syntactic analysis, which will be explained in the following chapter. In the context of lexical analysis, the words are called *tokens* (or *lexèmes* in French).

For example, if we are performing lexical analysis on a language like OCaml, and the source text is of the form:

```
fun x -> (* my function *)
    x + 1
```

Then the list of tokens constructed by the lexical analyzer will be of the form:

`fun`	`ident(x)`	`->`	`ident(x)`	`+`	`const(1)`

That is, a sequence of six tokens, representing successively the keyword `fun`, the identifier `x`, the symbol `->`, etc. In particular, the spaces, line breaks, and comments present in the initial source text, which are called *whitespace*, do not appear in the returned token sequence. However, they do play a role in lexical analysis. They are used, among other things, to separate two tokens. Thus, `funx`, written without a space between `fun` and `x`, would be understood as a single token, namely the identifier `funx`. Other whitespace is unnecessary, such as the spaces in `x + 1`, and is simply ignored.

Lexical conventions differ between languages, and some of the whitespace characters can sometimes be significant. One example is the input language of the `make` tool, where leading tabs are significant. Languages like Python and Haskell are examples where line breaks and leading spaces are significant, as they are used to define structure.

As we have just mentioned, comments act as whitespace. For example, if we write,

```
fun(* and hop *)x -> x + (* I add one *) 1
```

then the comment `(* and hop *)` acts as whitespace, useful for separating two tokens, and the comment `(* I add one *)` acts as unnecessary whitespace (in addition to being an unnecessary comment). Comments are sometimes used by certain tools (`ocamldoc`, `javadoc`, etc.), which then treat them differently in their own lexical analysis.

To perform lexical analysis, we will use *regular expressions* on the one hand to describe tokens, and *finite automata* on the other hand to recognize them. We particularly exploit

the ability to mechanically construct a finite automaton that recognizes the language described by a regular expression.

In all that follows, we are given an alphabet A, which represents the set of characters in the source texts to be analyzed. In practice, this will be 7-bit ASCII characters, UTF-8 characters, etc. A *word* over the alphabet A is a finite, possibly empty, sequence of characters. If a word is formed by the characters $a_1, a_2, \ldots, a_n$ in this order, we naturally denote it as $a_1 a_2 \cdots a_n$ and call n its length. The empty word, of length zero, is denoted as ϵ. A *language* is a set of words, not necessarily finite. The set of all possible words is one language among others, denoted as $A^\star$.

3.1 Regular Expression

Regular expressions provide a way to define languages. We begin by introducing their syntax.

Definition 4 (regular expression)**.** The abstract syntax of regular expressions is as follows:

$$\begin{array}{rcll} r & ::= & \emptyset & \text{empty language} \\ & | & \epsilon & \text{empty word} \\ & | & a & \text{character } a \in A \\ & | & r\,r & \text{concatenation} \\ & | & r \mid r & \text{alternative} \\ & | & r\star & \text{star} \end{array}$$

□

To write regular expressions in the following, we adopt the convention that the star has the highest precedence, followed by concatenation, and finally the alternative. If r is a regular expression and n is a natural number, we define the regular expression r^n recursively on n by setting $r^0 = \epsilon$ and $r^{n+1} = r\,r^n$.

Definition 5 (language of a regular expression)**.** The *language* defined by the regular expression r is the set of words $L(r)$ defined by:

$$\begin{aligned} L(\emptyset) &= \emptyset \\ L(\epsilon) &= \{\epsilon\} \\ L(a) &= \{a\} \\ L(r_1\,r_2) &= \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\} \\ L(r_1 \mid r_2) &= L(r_1) \cup L(r_2) \\ L(r\star) &= \textstyle\bigcup_{n \geq 0} L(r^n) \end{aligned}$$

□

We note in particular that $L(r_1\,(r_2\,r_3)) = L((r_1\,r_2)\,r_3)$, *i.e.*, that concatenation is associative. For this reason, we will avoid unnecessary parentheses by directly writing $r_1\,r_2\,r_3$, as there is no ambiguity. Similarly, we will write $r_1 \mid r_2 \mid r_3$ because $L(r_1 \mid (r_2 \mid r_3)) = L((r_1 \mid r_2) \mid r_3)$.

Examples. On the alphabet $A = \{a, b\}$, the regular expression $r = (a|b)(a|b)(a|b)$ defines the language of three-character words, *i.e.*, $L(r) = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$. The regular expression $(a|b) \star a$ defines the (infinite) language of words ending with a. Finally, the regular expression $(b|\epsilon)(ab) \star (a|\epsilon)$ defines the language of words alternating a and b.

Exercise 20. Still on the alphabet $A = \{a, b\}$, give a regular expression defining the language of words containing at most two characters b. Solution □

3.2 Finite Automaton

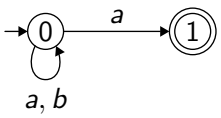
Finite automata provide another way to define languages.

Definition 6 (finite automaton)**.** A finite automaton over an alphabet A is a quadruple (Q, T, I, F) where:

- Q is a finite set of states;
- $T \subseteq Q \times A \times Q$ is a set of transitions;
- $I \subseteq Q$ is a set of initial states;
- $F \subseteq Q$ is a set of final states.

If, in addition, T is a function of its first two arguments, *i.e.*, for $q \in Q$ and $a \in A$ there is at most one $q' \in Q$ such that $(q, a, q') \in T$, the automaton is said to be *deterministic*. □

Example. On the alphabet $A = \{a, b\}$, we can consider the automaton defined by $Q = \{0, 1\}$, $T = \{(0, a, 0), (0, b, 0), (0, a, 1)\}$, $I = \{0\}$, and $F = \{1\}$. Such an automaton is traditionally represented as follows,



that is, with an incoming arrow on the initial states and a double circle around the final states. A transition $(q, c, q') \in Q$ is represented by an arc connecting the states q and q', labeled with the character c. This automaton is not deterministic, as there are two transitions labeled with a leaving state 0.

Definition 7 (language of a finite automaton)**.** A word $a_1 a_2 \dots a_n \in A^\star$ is *recognized* by an automaton (Q, T, I, F) if and only if there exist transitions:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

with $s_0 \in I$, $s_n \in F$, and $(s_{i-1}, a_i, s_i) \in T$ for all i such that $1 \leq i \leq n$.

The language defined by an automaton is the set of words recognized by this automaton. □

Thus, the finite automaton given in the example above defines the language of words ending with the character a.

Regular expressions and finite automata are linked by a fundamental result in language theory, which states that they define the same languages. This means that a language defined by a regular expression is also defined by (at least) one finite automaton, and that the language defined by a finite automaton is also defined by (at least) one regular expression. Thus, the finite automaton above defines the same language as the regular expression $(a|b) \star a$.

Construction of the Finite Automaton. There are multiple ways to construct the finite automaton corresponding to a given regular expression. Here, we illustrate a method that is effectively used in the construction of lexical analyzers. The idea is to match the characters of a recognized word with those appearing in the regular expression. For example, the word *aabaab* belongs to the language of the regular expression $(a|b) \star a(a|b)$, and we can make the following correspondence:

$$\begin{array}{ll} a & (\mathbf{a}|b) \star a(a|b) \\ a & (\mathbf{a}|b) \star a(a|b) \\ b & (a|\mathbf{b}) \star a(a|b) \\ a & (\mathbf{a}|b) \star a(a|b) \\ a & (a|b) \star \mathbf{a}(a|b) \\ b & (a|b) \star a(a|\mathbf{b}) \end{array}$$

This correspondence is not necessarily unique, although in this case it is.

We will construct a finite automaton whose states are sets of characters from the regular expression. To do this, we will distinguish the different characters of the regular expression, for example by associating distinct indices with them:

$$(a_1|b_1) \star a_2(a_3|b_2)$$

This distinction remains conceptual: a_1, a_2, and a_3 still represent the character a, and b_1 and b_2 still represent the character b. From a state s of our automaton, we will recognize words that are suffixes of words recognized by the regular expression and whose first letter belongs to s. In the example above, from the state $\{a_1, a_2, b_1\}$, we will recognize words whose first character can be matched with a_1, a_2, or b_1.

To construct the transitions between a state s_1 and a state s_2, we need to determine the characters that can appear after another in a recognized word. Let $\mathit{follow}(c, r)$ denote the characters that can appear after the character c in a word recognized by the regular expression r. For example, with $r = (a_1|b_1) \star a_2(a_3|b_2)$, we have:

$$\mathit{follow}(a_1, r) = \{a_1, a_2, b_1\}$$

To compute *follow*, we need to compute the possible first (resp. last) letters of a recognized word. Let $\mathit{first}(r)$ (resp. $\mathit{last}(r)$) denote this set for a given regular expression r. With the same example, we have:

$$\begin{array}{rcl} \mathit{first}(r) & = & \{a_1, a_2, b_1\} \\ \mathit{last}(r) & = & \{a_3, b_2\} \end{array}$$

And to compute *first* and *last*, we need one last notion: does the empty word belong to the language $L(r)$? Let $null(r)$ denote this property. It turns out that it is very easy to determine $null(r)$ by structural induction on the regular expression r, as follows:

$$\begin{aligned} null(\emptyset) &= \texttt{false} \\ null(\epsilon) &= \texttt{true} \\ null(a) &= \texttt{false} \\ null(r_1\, r_2) &= null(r_1) \wedge null(r_2) \\ null(r_1 \mid r_2) &= null(r_1) \vee null(r_2) \\ null(r\star) &= \texttt{true} \end{aligned}$$

We can then deduce the definition of *first* and *last*, again by structural induction on the regular expression r. For *first*, we proceed as follows:

$$\begin{aligned} first(\emptyset) &= \emptyset \\ first(\epsilon) &= \emptyset \\ first(a) &= \{a\} \\ first(r_1\, r_2) &= first(r_1) \cup first(r_2) \quad \text{if } null(r_1) \\ &= first(r_1) \quad \text{otherwise} \\ first(r_1 \mid r_2) &= first(r_1) \cup first(r_2) \\ first(r\star) &= first(r) \end{aligned}$$

The only subtlety lies in the concatenation $r_1 r_2$. Indeed, when $null(r_1)$ is true, we must also include $first(r_2)$ in the result. The definition of *last* is similar and left as an exercise.

Exercise 21. Give the definition of *last*. Solution □

This finally leads to the definition of *follow*, again by structural induction:

$$\begin{aligned} follow(c, \emptyset) &= \emptyset \\ follow(c, \epsilon) &= \emptyset \\ follow(c, a) &= \emptyset \\ follow(c, r_1\, r_2) &= follow(c, r_1) \cup follow(c, r_2) \cup first(r_2) \quad \text{if } c \in last(r_1) \\ &= follow(c, r_1) \cup follow(c, r_2) \quad \text{otherwise} \\ follow(c, r_1 \mid r_2) &= follow(c, r_1) \cup follow(c, r_2) \\ follow(c, r\star) &= follow(c, r) \cup first(r) \quad \text{if } c \in last(r) \\ &= follow(c, r) \quad \text{otherwise} \end{aligned}$$

Again, the only difficulty lies in the concatenation $r_1\, r_2$. Indeed, the character c and the one following it can be straddling r_1 and r_2, when $c \in last(r_1)$, in which case $first(r_2)$ must be included in the result. The same situation can occur with the concatenation of two occurrences of r in the case of $r\star$.

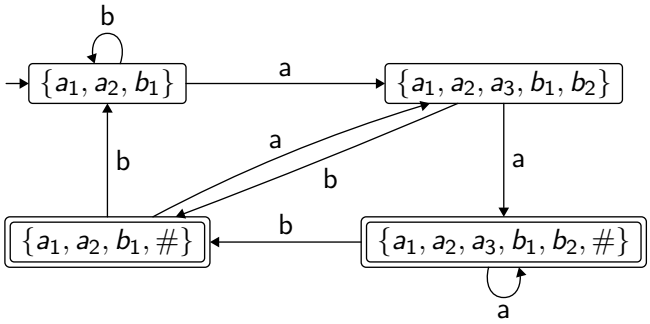
We now have everything we need to construct the finite automaton recognizing the language of r. We start by adding a character not belonging to the alphabet A at the

end of r; let's denote it as #. The initial state of our automaton is the set $\mathit{first}(r\#)$. We then construct the other states and transitions as needed, using the following algorithm:

while there exists a state s whose transitions need to be calculated
  for each character c in the alphabet
    let s' be the state $\bigcup_{c_i \in s} \mathit{follow}(c_i, r\#)$
    add the transition $s \xrightarrow{c} s'$

It is clear that this construction terminates, as the possible states are finite in number (they are subsets of the set of characters in the regular expression). The final states are the states containing the character #.

For the example $(a|b) \star a(a|b)$, we obtain the following automaton:



We can easily verify that it indeed recognizes the language of words containing a character a in the penultimate position. It happens that this automaton is minimal in terms of the number of states, but this is not necessarily always the case.

Exercise 22. Give the result of this construction for the regular expression $(a|b) \star a$. Compare with the automaton given on page 51. Solution □

3.3 Lexical Analyzer

A *lexical analyzer* is a finite automaton for the "union" of all regular expressions defining the tokens. The operation of a lexical analyzer, however, is different from the simple recognition of a word by an automaton, because:

- it must decompose a word (the source text) into a *sequence* of recognized words;
- there can be *ambiguities*;
- it must construct the tokens, instead of simply recognizing words, *i.e.*, the final states contain *actions* to be performed.

We have given examples of ambiguity above. For instance, if we have written `funx` in a language where `fun` is a keyword, then we could recognize `funx` as an identifier but also as the keyword `fun` followed by the identifier `x`. To resolve this ambiguity, we generally choose to recognize the longest possible token.

Another potential ambiguity arises from the fact that the word `fun` is recognized both by the regular expression of the keyword `fun` and by that of identifiers. Rather than trying to define a regular expression for identifiers that excludes all keywords, which would be

extremely tedious, we instead choose to rank the regular expressions by order of priority. At equal length, it is the highest-priority regular expression that defines the action to be taken.

Finally, a notable feature of a lexical analyzer is that we do not seek to backtrack in case of failure. Thus, if a, ab, and bc are the three possible tokens in our language, our lexical analyzer will *fail* on the input:

$$abc$$

Even though this text can be decomposed into two tokens, namely a and bc. The reason for this failure is that after recognizing the longest possible token, namely ab, we no longer seek to backtrack on this decision. We therefore fail on the input c, which is not in the language of tokens.

It is only a pragmatic consideration that leads to this decision not to backtrack, for better performance.

3.4 The ocamllex Tool

In practice, we have tools that construct a lexical analyzer from its description using regular expressions. This is the large family of the `lex` tool, which is available for a majority of languages: `flex` for C, `jflex` for Java, `ocamllex` for OCaml, etc. We present the `ocamllex` tool here, but the operation of other tools in the `lex` family is very similar.

An example of an `ocamllex` file is given in figure 3.1 for the lexical analysis of a small language of arithmetic expressions with literal constants, addition, multiplication, and parentheses. Such a file begins with an arbitrary OCaml code prelude between braces. Here, we define a `token` type for the tokens. Next comes the declaration of a lexical analysis function `token`, in a syntax specific to the `ocamllex` tool, of the form:

```
rule token = parse
| regexp1 { action1 }
| regexp2 { action2 }
| ...
```

Each `regexp`i is a regular expression. The keyword `parse` means that it is the longest possible prefix of the input recognized by one of the regular expressions that determines the selected line[1]. At equal length, it is the regular expression that comes first that is selected.

Each regular expression `regexp`i is associated with an action `action`i, which is an arbitrary piece of OCaml code. In our example, this code constructs a value of type `token`. For example, the two lines:

```
  | ['0'-'9']+ as s
      { CONST (int_of_string s) }
```

recognize literal constants with the regular expression `['0'-'9']+`, *i.e.*, a repetition of at least one character among `'0'`, ..., `'9'`. The associated action returns a `CONST` token with the value of the constant. This value is obtained by retrieving the string `s` recognized by the regular expression with `as s`. The other tokens are handled similarly. The `EOF` token represents the end of input, and for this, we use the special regular expression `eof`.

[1] Another keyword, `shortest`, allows selecting the shortest prefix instead.

```
{
  type token =
    | CONST of int
    | PLUS
    | TIMES
    | LEFTPAR
    | RIGHTPAR
    | EOF
}

rule token = parse
  | [' ' '\t' '\n']+
      { token lexbuf }
  | ['0'-'9']+ as s
      { CONST (int_of_string s) }
  | '+'
      { PLUS }
  | '*'
      { TIMES }
  | '('
      { LEFTPAR }
  | ')'
      { RIGHTPAR }
  | eof
      { EOF }
  | _
      { failwith "lexical error" }
```

Figure 3.1: Lexical analysis of arithmetic expressions with `ocamllex`.

In addition to constructing tokens, we must also handle two other cases. On the one hand, we need to ignore whitespace. For this, we use the regular expression `[' ' '\t' '\n']+` which recognizes a sequence of space, tab, and newline characters. To ignore these characters, we recursively call the lexical analyzer `token`, passing it an argument `lexbuf` that is implicit in the `ocamllex` code. It represents the data structure on which we are performing the lexical analysis (a string, a file, etc.). On the other hand, we must consider illegal characters. We recognize any character with the regular expression `_` and then signal the lexical error by raising an exception. In practice, we should make an additional effort to signal the nature and position of this lexical error.

Such an `ocamllex` source is written in a file with the suffix `.mll`, for example `lexer.mll`. It is compiled with the `ocamllex` tool, like this:

```
> ocamllex lexer.mll
```

This produces an OCaml file `lexer.ml` that contains the definition of the `token` type written in the prelude and a `token` function:

```
type token = ...
val token : Lexing.lexbuf -> token
```

The data structure on which the lexical analysis is performed has the type `Lexing.lexbuf`, from the `Lexing` module of OCaml's standard library. This module provides several ways to construct a value of this type, such as a `from_channel` function when you want to analyze the contents of a file.

Shortcuts. It is possible to define shortcuts for regular expressions using the `let` construct of `ocamllex`. Thus, we can advantageously write a lexical analyzer containing identifiers, integer, and floating-point literal constants as follows:

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let decimals = '.' digit*
let exponent = ['e' 'E'] ['+' '-']?  digit+

rule token = parse
  | letter (letter | digit | '_')*          { ... }
  | digit+                                  { ... }
  | digit+ (decimals | decimals? exponent)  { ... }
  | ...
```

Recognition of Keywords. When the analyzed language contains keywords, we take care to indicate them *before* the identifiers:

```
rule token = parse
| "fun"        { FUN     }
| letter+ as s { IDENT s }
```

Indeed, at equal length, the first rule that applies will be used, as explained above. For the word `fun`, we must construct the `FUN` token and not an identifier. If the language contains a very large number of keywords, we can group them in a table and then write something like:

```
rule token = parse
| letter+ as s { try Hashtbl.find table s with Not_found -> IDENT s }
```

In particular, the automaton constructed by `ocamllex` will be much smaller.

Multiple Lexical Analyzers. The `ocamllex` tool allows defining several lexical analyzers simultaneously, in a mutually recursive manner. Thus, we can define, for example, a `token` analyzer serving as the main entry point and another `string` analyzer to recognize character strings.

```
rule token = parse
| ...
| '"'    { string lexbuf }
and string = parse
| '"'    { ... }
| "\\n"  { ...; string lexbuf }
| _ as c { ...; string lexbuf }
| eof    { raise UnterminatedString }
```

This allows, in particular, handling escape sequences (such as `\n`) in a specific way or failing with a specific error on an unterminated string. If we had chosen instead to recognize a string with a single regular expression, it would have been much more difficult to achieve.

A secondary lexical analyzer also allows recognizing nested comments. For comments of the form `(*...*)`, as in OCaml, we can thus give the following rules:

```
rule token = parse
  | "(*"  { comment lexbuf; token lexbuf }
  | ...

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { raise UnterminatedComment }
```

The principle here is that the `comment` analyzer recognizes a comment until its end, without returning anything. Thus, it can call itself recursively to recognize a nested comment. There is no need for a counter here; the call stack counts for us.

Bibliographical Notes. Regular languages are the subject of the first chapter of Olivier Carton's book [5]. The construction of the finite automaton from the regular expression presented in this chapter is due to Berry and Sethi [3].

4 Parsing

The goal of syntactic analysis, or *parsing*, is to recognize sentences belonging to the syntax of the language. Its input is the stream of tokens constructed by lexical analysis (previous chapter), and its output is an abstract syntax tree (section 2.1). In particular, syntactic analysis must detect syntax errors and report them precisely.

To perform syntactic analysis, we will use tools analogous to the regular expressions and finite automata used for lexical analysis, namely *context-free grammars* to describe the syntax (section 4.2), *pushdown automata* to recognize them (sections 4.3 and 4.4), and tools to automate all of this (section 4.5). However, we will begin this chapter with something more elementary.

4.1 Elementary Syntactic Analysis

Before describing the techniques and tools of syntactic analysis, we will attempt to create a syntactic analyzer using elementary means. There is indeed much to learn from such an attempt. To keep it simple, but not too simple, let's limit ourselves to analyzing arithmetic expressions written using literal constants, additions, multiplications, and parentheses, with the usual convention that multiplication has priority over addition. In case of success, we want to construct the corresponding abstract syntax tree, whose type is as follows:

```
type expr =
| Const of int
| Add   of expr * expr
| Mul   of expr * expr
```

Thus, the analysis of a string such as `2 + 5 * (4 + 4)` should succeed and give the abstract syntax tree `Add (Const 2, Mul (Const 5, Add (Const 4, Const 4)))`. On the other hand, the analysis of a string like `1+*2` or `(3+4` should signal a syntax error.

The great researcher Gilles Kahn once told me that to write a syntactic analyzer, one should start by writing a printing function. Let's follow his advice. We will use the `Format` module from OCaml's standard library and write a general printing function of type:

```
val print: Format.formatter -> expr -> unit
```

The first argument contains all the machinery necessary for printing, such as the file in which we are writing. The `Format` module provides, among other things, a `fprintf` function to write using such a `formatter`.

Of course, we can write a very crude printing function that parenthesizes every subexpression to avoid any ambiguity, as follows[1]:

```
let rec print fmt = function
  | Const n      -> fprintf fmt "%d" n
  | Add (e1, e2) -> fprintf fmt "(%a + %a)" print e1 print e2
  | Mul (e1, e2) -> fprintf fmt "(%a * %a)" print e1 print e2
```

This is a correct solution, but rather disappointing. Apart from the fact that we use parentheses unnecessarily, we make no effort to insert line breaks or to show the structure. Let's therefore try to write an *elegant* printing function, known as a *pretty-printer*.

As for inserting line breaks and showing the structure, the `Format` library offers everything we need. It indeed defines a notion of boxes, in the typographical sense, and various strategies to combine them, as well as a notion of breakable space. We use them as follows:

```
let rec print fmt = function
  | Const n      -> fprintf fmt "%d" n
  | Add (e1, e2) -> fprintf fmt "(@[%a +@ %a@])" print e1 print e2
  | Mul (e1, e2) -> fprintf fmt "(@[%a *@ %a@])" print e1 print e2
```

The directives `@[` and `@]` open and close a box respectively, and the directive `@` indicates a breakable space. Here, we have chosen to associate a box with each parenthesized expression. If a parenthesized expression does not fit on the line, and a line break is inserted, then the printing will continue on the column following the opening parenthesis, as follows:

```
(2 + (3 * ((5 + 6) *
           (7 * 8))))
```

Of course, this is a personal choice of box usage. Other uses are possible, with different results.

The problem of unnecessary parentheses remains. If we take the example of the expression `2 + 5 * (4 + 4)`, only one pair of parentheses is necessary for its printing. It is made necessary by the fact that an argument of the multiplication is an addition, namely `4+4`, *i.e.*, an operation of lower priority. We might therefore be tempted to write a printing function taking an additional argument representing the priority of the operation being printed. This is a possible solution. However, there is another, equivalent but more elegant, solution: writing several printing functions, one for each priority level. Thus, we can write a function `print_expr` to print expressions of the lowest priority, *i.e.*, sums, then a function `print_term` to print expressions of the next priority, *i.e.*, products, and finally a third function `print_factor` to print expressions of the highest priority, *i.e.*, parenthesized expressions.

The first of these functions prints sums, *i.e.*, expressions of the form `Add`. It does so without using parentheses.

[1] The `Format` library's `%a` conversion allows passing a printing function and an argument for that function. Here, it is our own `print` function that we pass recursively.

```
type expr =
  | Const of int
  | Add   of expr * expr
  | Mul   of expr * expr

open Format

let rec print_expr fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
  | e            -> print_term fmt e

and print_term fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
  | e            -> print_factor fmt e

and print_factor fmt = function
  | Const n -> fprintf fmt "%d" n
  | e       -> fprintf fmt "(@[%a@])" print_expr e
```

Figure 4.1: Pretty-printer for arithmetic expressions.

```
  let rec print_expr fmt = function
    | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
```

If, on the other hand, the expression is not of the form `Add`, it passes the task to the second function, `print_term`:

```
    | e            -> print_term fmt e
```

We proceed similarly in `print_term`, displaying expressions of the form `Mul`, if applicable, and otherwise passing the task to the third function.

```
  and print_term fmt = function
    | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
    | e            -> print_factor fmt e
```

The third function handles the remaining case, namely literal constants.

```
  and print_factor fmt = function
    | Const n -> fprintf fmt "%d" n
```

If, on the other hand, the expression is not a constant, it prints it in parentheses, with `print_expr`.

```
    | e       -> fprintf fmt "(@[%a@])" print_expr e
```

The complete code is given in figure 4.1. The effect obtained is correct, in particular because an argument of `Mul` of type `Add` will not be handled by `print_term` but passed to `print_factor`, which will print it in parentheses. On the example given above, we obtain `2 + 3 * (5 + 6) * 7 * 8`, which is the expected result.

Let's now address the actual problem of syntactic analysis. We assume we are given a lexical analyzer such as the one presented at the end of the previous chapter (figure 3.1 page 56), *i.e.*, a `token` type for tokens and a function:

```
val token: Lexing.lexbuf -> token
```

which we call each time we want to obtain the next token. We set up some machinery to read the tokens, with a reference `tok` containing the next token to be examined and a function `next` to read the next token.

```
let tok = ref EOF
let next () = tok := token lb
```

We assume that `lb` here represents the structure of type `Lexing.lexbuf` on which the lexical analysis is performed (typically a string or a file). We start by initializing `tok` with the very first token.

```
let () = next ()
```

Finally, we set up a function to signal any syntax error.

```
let error () = failwith "syntax error"
```

The printing function we have just written has put us on the right track by distinguishing sums, products, and factors. Let's therefore write three syntactic analysis functions `parse_expr`, `parse_term`, and `parse_factor` based on the same idea. These three functions have the type `unit -> expr`. Our invariant is that each of these three functions consumes all the tokens that make up the recognized expression. The function `parse_expr` must recognize a sum. It starts by recognizing a first term by calling the function `parse_term`.

```
let rec parse_expr () =
  let e = parse_term () in
```

Then it examines the next token. If it is `PLUS`, *i.e.*, the symbol `+`, then it consumes it with `next` and continues reading a sum with another call to `parse_expr`. Otherwise, the reading of the sum is finished and it returns `e`.

```
    if !tok = PLUS then begin next (); Add (e, parse_expr ()) end else e
```

The function `parse_term` proceeds similarly, recognizing a product of factors separated by the token `TIMES`.

```
and parse_term () =
  let e = parse_factor () in
  if !tok = TIMES then begin next (); Mul (e, parse_term ()) end else e
```

Finally, the function `parse_factor` must recognize literal constants and parenthesized expressions. There is no difficulty with the former. We just need to remember to consume the token.

```
and parse_factor () = match !tok with
  | CONST n -> next (); Const n
```

For parenthesized expressions, it is more subtle. We start by consuming the token corresponding to the opening parenthesis, then we recognize an expression with `parse_expr`.

```
  | LEFTPAR ->
      next ();
      let e = parse_expr () in
```

Then, we need to check that we indeed find a closing parenthesis right after the expression `e`. If not, we signal a syntax error. Otherwise, we consume the closing parenthesis and return `e`.

```
        if !tok <> RIGHTPAR then error ();
        next (); e
```

If the first token is neither a constant nor an opening parenthesis, then we signal a syntax error.

```
    | _ ->
        error ()
```

To recognize the entire expression, it is sufficient to call the function `parse_expr`. Indeed, if the expression is not a sum, it will reduce to a single term recognized by `parse_term`. If similarly this term is not a product, it will reduce to a single factor recognized by `parse_factor`. However, we must remember to check that the entire input has been consumed. We do this as follows:

```
let e = parse_expr ()
let () = if !tok <> EOF then error ()
```

Without this last check, an input like `1+2 3` would be accepted. The complete code is given in figure 4.2.

4.2 Grammar

We have already informally used the notion of grammar in section 2.1 to define abstract syntax. We now give a formal definition of what a grammar is.

Definition 8 (grammar)**.** A context-free grammar is a quadruple (N, T, S, R) where:

- N is a finite set of *non-terminal symbols*;
- T is a finite set of *terminal symbols*;
- $S \in N$ is the start symbol (called *axiom*);
- $R \subseteq N \times (N \cup T)^\star$ is a finite set of *production rules*. □

Example. Here is an example of a very simple grammar for arithmetic expressions consisting of integer constants, additions, multiplications, and parentheses.

$$
\begin{aligned}
&N = \{E\} \\
&T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{int}\} \\
&S = E \\
&R = \{\, (E, E\texttt{+}E),\ (E, E\texttt{*}E),\ (E, \texttt{(}E\texttt{)}),\ (E, \texttt{int}) \,\}
\end{aligned}
$$

In practice, we write the rules in a more readable form, like this:

$$
\begin{array}{rcl}
E & \to & E \texttt{ + } E \\
& \mid & E \texttt{ * } E \\
& \mid & \texttt{(} E \texttt{)} \\
& \mid & \texttt{int}
\end{array}
\tag{4.1}
$$

```
let lb = Lexing.from_channel stdin
let tok = ref EOF               (* the next token to be examined *)
let next () = tok := token lb
let () = next ()

let error () = failwith "syntax error"

let rec parse_expr () =
  let e = parse_term () in
  if !tok = PLUS then begin next (); Add (e, parse_expr ()) end else e
and parse_term () =
  let e = parse_factor () in
  if !tok = TIMES then begin next (); Mul (e, parse_term ()) end else e
and parse_factor () = match !tok with
  | CONST n ->
      next (); Const n
  | LEFTPAR ->
      next (); let e = parse_expr () in
      if !tok <> RIGHTPAR then error (); next (); e
  | _ ->
      error ()

let e = parse_expr ()
let () = if !tok <> EOF then error ()
```

Figure 4.2: Elementary parsing of arithmetic expressions.

In our context, the terminals of the grammar are the tokens produced by lexical analysis. Here, `int` denotes the token corresponding to an integer constant. □

Our goal is now to define the language of words accepted by this grammar. To do this, we start by introducing the notion of derivation.

Definition 9 (derivation)**.** A word $u \in (N \cup T)^\star$ *derives* into a word $v \in (N \cup T)^\star$, and we denote this as $u \to v$, if there exists a decomposition:

$$u = u_1 X u_2$$

with $X \in N$, $X \to \beta \in R$, and:

$$v = u_1 \beta u_2.$$

A sequence $w_1 \to w_2 \to \cdots \to w_n$ is called a derivation. We speak of a *leftmost derivation* (resp. *rightmost derivation*) if the non-terminal reduced is systematically the leftmost, *i.e.*, $u_1 \in T^\star$ (resp. the rightmost, *i.e.*, $u_2 \in T^\star$). We denote $\to^\star$ as the reflexive transitive closure of $\to$. □

With the grammar above, we have, for example, the following leftmost derivation:

$$\begin{aligned} E &\to E \texttt{ * } E \\ &\to \texttt{int * } E \\ &\to \texttt{int * (} E \texttt{)} \\ &\to \texttt{int * (} E \texttt{ + } E \texttt{)} \\ &\to \texttt{int * (int + } E \texttt{)} \\ &\to \texttt{int * (int + int)} \end{aligned}$$

The language defined by a grammar then comes as no surprise:

Definition 10. The *language* defined by a context-free grammar $G = (N, T, S, R)$ is the set of words in $T^\star$ derived from the axiom, *i.e.*,

$$L(G) = \{\, w \in T^\star \mid S \to^\star w \,\}.$$

□

With the same grammar as above, we have therefore established above that:

$$\texttt{int * (int + int)} \in L(G).$$

Definition 11 (derivation tree)**.** For a given grammar, a *derivation tree* is a tree whose nodes are labeled by symbols of the grammar, as follows:

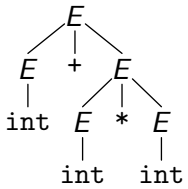
- the root is the axiom S;
- every internal node X is a non-terminal whose children are labeled by $\beta \in (N \cup T)^\star$ with $X \to \beta$ a rule of the grammar. □

For a derivation tree whose leaves form the word w in infix order, it is clear that $S \to^\star w$. Conversely, for any derivation $S \to^\star w$, we can associate a derivation tree whose leaves form the word w in infix order (proof by induction on the length of the derivation).

Example. The leftmost derivation:

$$E \rightarrow E \texttt{ + } E \rightarrow \texttt{int + } E \rightarrow \texttt{int + } E \texttt{ * } E \rightarrow \texttt{int + int * } E \rightarrow \texttt{int + int * int}$$

corresponds to the following derivation tree:



But this derivation tree also corresponds to the rightmost derivation:

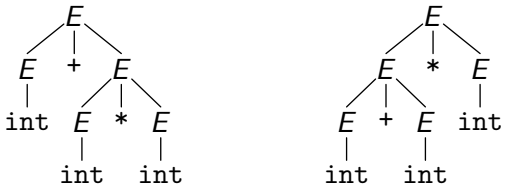
$$E \rightarrow E \texttt{ + } E \rightarrow E \texttt{ + } E \texttt{ * } E \rightarrow E \texttt{ + } E \texttt{ * int} \rightarrow E \texttt{ + int * int} \rightarrow \texttt{int + int * int}$$

as well as to other derivations that are neither leftmost nor rightmost.

The notion of derivation tree allows us to identify a whole set of derivations that are "basically the same." If, on the other hand, a single word admits several derivation trees, this is a symptom of ambiguity in the grammar.

Definition 12 (ambiguity)**.** A grammar is said to be *ambiguous* if at least one word admits several derivation trees.

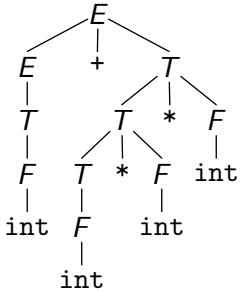
Example. The grammar used as an example so far is ambiguous. Indeed, the word `int + int * int` admits the following two derivation trees:



However, it is possible to propose another, unambiguous grammar that defines the same language. Here is one solution (but it is not the only one):

$$\begin{array}{rcl} E & \rightarrow & E \texttt{ + } T \\ & | & T \\ T & \rightarrow & T \texttt{ * } F \\ & | & F \\ F & \rightarrow & \texttt{(} E \texttt{)} \\ & | & \texttt{int} \end{array} \tag{4.2}$$

This new grammar reflects the priority of multiplication over addition and the choice of left associativity for these two operations. Thus, the word `int + int * int * int` now has only one derivation tree, namely:



corresponding to the leftmost derivation:

$$\begin{aligned} E &\to E + T \to T + T \to F + T \to \texttt{int} + T \to \texttt{int} + T * F \\ &\to \texttt{int} + T * F * F \to \texttt{int} + F * F * F \to \texttt{int} + \texttt{int} * F * F \\ &\to \texttt{int} + \texttt{int} * \texttt{int} * F \to \texttt{int} + \texttt{int} * \texttt{int} * \texttt{int} \end{aligned}$$

Exercise 23. Show that this new grammar recognizes the same language as the previous one. Solution □

Exercise 24. Here is a possible grammar for the terms of the λ-calculus (assuming that we use what are called *de Bruijn indices*):

$$\begin{aligned} T &\to \texttt{nat} \\ &\mid T\ T \\ &\mid \texttt{lam}\ T \end{aligned}$$

Show that this grammar is ambiguous. Propose another grammar that recognizes the same language and is not ambiguous. Solution □

Unfortunately, determining whether a grammar is ambiguous or not is *not decidable*[2]. We will therefore use *decidable sufficient criteria* to guarantee that a grammar is unambiguous, for which we also know how to decide membership in the language efficiently (with a deterministic pushdown automaton). The classes of grammars defined by these criteria have strange names like LL(1), LR(0), SLR(1), LALR(1), or LR(1), which we will introduce in the following sections. But for this, we need some additional notions about grammars.

The first of these notions is the property for a non-terminal X to derive the empty word, *i.e.*, $X \to^\star \epsilon$. We denote this property as $\text{NULL}(X)$. We define it more generally for any word.

Definition 13 (NULL)**.** Let $\alpha \in (T \cup N)^\star$. The property $\text{NULL}(\alpha)$ is true if and only if we can derive ϵ from α, *i.e.*, $\alpha \to^\star \epsilon$.

The second notion characterizes the terminal symbols that can appear in the first position of a word derived by the grammar.

Definition 14 (FIRST)**.** Let $\alpha \in (T \cup N)^\star$. We define $\text{FIRST}(\alpha)$ as the set of all first terminals of the words derived from α, *i.e.*, $\text{FIRST}(\alpha) = \{a \in T \mid \exists w.\, \alpha \to^\star aw\}$.

Finally, the third notion characterizes the terminal symbols that can appear after a non-terminal symbol in a derivation.

Definition 15 (FOLLOW)**.** Let $X \in N$. We define $\text{FOLLOW}(X)$ as the set of all terminals that can appear after X in a derivation of a recognized word, *i.e.*, $\text{FOLLOW}(X) = \{a \in T \mid \exists u, w.\, S \to^\star uXaw\}$.

These notions are analogous to those we introduced for regular expressions on page 52. It remains to show how to compute them.

[2]We recall that decidable means that we can write a program that, for any input, terminates and answers yes or no.

Computing NULL, FIRST, and FOLLOW. To compute $\text{NULL}(\alpha)$, it is sufficient to determine $\text{NULL}(X)$ for each $X \in N$. Indeed, $\text{NULL}(X)$ is true if and only if there exists a production $X \to \epsilon$ in the grammar or there exists a production $X \to Y_1 \dots Y_m$ with $\text{NULL}(Y_i)$ for all i. This is therefore a set of mutually recursive equations that define the values of $\text{NULL}(X)$ in terms of themselves. In other words, if $N = \{X_1, \dots, X_n\}$ and if we set $\vec{V} = (\text{NULL}(X_1), \dots, \text{NULL}(X_n))$, we are looking for the *smallest solution* to an equation of the form:

$$\vec{V} = F(\vec{V})$$

Fortunately, we are in a situation where there is a result that will allow us to solve such an equation easily, namely a simple form of the Knaster–Tarski theorem.

Theorem 1 (existence of a smallest fixed point)**.** Let A be a finite set equipped with an order relation $\leq$ and a smallest element ε. Every increasing function $f : A \to A$, *i.e.*, such that $\forall x, y.\, x \leq y \Rightarrow f(x) \leq f(y)$, admits a *smallest fixed point*, *i.e.*, there exists $a_0 \in A$ such that:

$$f(a_0) = a_0$$

and such that for any other fixed point $b = f(b)$, we have $a_0 \leq b$.

Proof. Since ε is the smallest element, we have $\varepsilon \leq f(\varepsilon)$. Since the function f is increasing, we therefore have $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ for all k. Since the set A is finite, there therefore exists a smallest k_0 such that $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$. We have just found a fixed point $a_0 = f^{k_0}(\varepsilon)$ of f.

Let b be another fixed point of f. We have $\varepsilon \leq b$ and therefore $f^k(\varepsilon) \leq f^k(b)$ for all k. In particular, $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$. The fixed point a_0 is therefore the smallest fixed point of f. □

In the case of computing NULL, we have $A = \text{BOOL} \times \cdots \times \text{BOOL}$ with $\text{BOOL} = \{\texttt{false}, \texttt{true}\}$. We can equip BOOL with the order $\texttt{false} \leq \texttt{true}$ and A with the pointwise order, *i.e.*,

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \quad \text{if and only if} \quad \forall i.\, x_i \leq y_i.$$

The theorem then applies by taking:

$$\varepsilon = (\texttt{false}, \dots, \texttt{false})$$

because the function computing $\text{NULL}(X)$ from the $\text{NULL}(X_i)$ is increasing. In other words, it is sufficient to initially consider that all $\text{NULL}(X)$ are false, then apply the NULL computation function until the value of $\text{NULL}(X)$ no longer changes.

Let's take the example of this grammar:

$$\begin{array}{rcl} E & \to & T\, E' \\ E' & \to & \texttt{+}\, T\, E' \\ & | & \epsilon \\ T & \to & F\, T' \\ T' & \to & \texttt{*}\, F\, T' \\ & | & \epsilon \\ F & \to & \texttt{(}\, E\, \texttt{)} \\ & | & \texttt{int} \end{array} \tag{4.3}$$

This is a third variant of the grammar of arithmetic expressions. The computation will converge in two steps, as follows:

E	E'	T	T'	F
`false`	`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`false`

We deduce that the empty word can be derived from the non-terminal symbols E' and T', but not from the symbols E, T, and F. This was not entirely obvious *a priori*, because the rule $E \to T\,E'$ could have led to $\textsc{null}(E)$ if we had also had $\textsc{null}(T)$.

Exercise 25. Justify that we are looking for a *smallest* fixed point to compute the values of NULL. Solution □

Let's now move on to the computation of FIRST. To compute $\textsc{first}(\alpha)$ for any word, it is sufficient to know how to compute $\textsc{first}(X)$ for each non-terminal X. Indeed, we have:

$$\begin{aligned}
\textsc{first}(\epsilon) &= \emptyset \\
\textsc{first}(a\beta) &= \{a\}, \quad \text{if } a \in T \\
\textsc{first}(X\beta) &= \textsc{first}(X), \quad \text{if } \neg\textsc{null}(X) \\
\textsc{first}(X\beta) &= \textsc{first}(X) \cup \textsc{first}(\beta), \quad \text{if } \textsc{null}(X)
\end{aligned}$$

And to compute $\textsc{first}(X)$, it is sufficient to consider all the productions for X in the grammar:

$$\textsc{first}(X) = \bigcup_{X \to \beta} \textsc{first}(\beta)$$

We thus find ourselves again with recursive equations defining the sets $\textsc{first}(X)$. We can again use the Knaster–Tarski theorem, this time on the Cartesian product $A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$ equipped, pointwise, with the inclusion as the order relation. The smallest element is $\varepsilon = (\emptyset, \ldots, \emptyset)$.

If we take the example of grammar (4.3), we converge in four steps, as follows:

E	E'	T	T'	F
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	{`+`}	$\emptyset$	{`*`}	{`(`, `int`}
$\emptyset$	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}
{`(`, `int`}	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}
{`(`, `int`}	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}

We now need to compute FOLLOW. Obviously, the set $\textsc{first}(\beta)$ is part of $\textsc{follow}(X)$ if we have a production of the form $Y \to \alpha X \beta$. More subtly, if $\textsc{null}(\beta)$, we must also add to $\textsc{follow}(X)$ all the elements of $\textsc{follow}(Y)$. We therefore have:

$$\textsc{follow}(X) = \bigcup_{Y \to \alpha X \beta} \textsc{first}(\beta) \;\cup \bigcup_{Y \to \alpha X \beta, \textsc{null}(\beta)} \textsc{follow}(Y)$$

Again, these are mutually recursive equations for which we will proceed with a fixed point computation, on the same domain as for FIRST. In practice, it will be useful to add a

special terminal symbol, denoted as #, to represent the end of input. We can add it directly to FOLLOW(S).

If we take the example of grammar (4.3), we converge in four steps, as follows:

E	E'	T	T'	F
$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\#,)\}$	$\{\#\}$	$\{+, \#\}$	$\emptyset$	$\{*\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#\}$	$\{*, +, \#\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$

Exercise 26. Here is a possible grammar for the terms of the λ-calculus:

$$\begin{array}{rcl} S & \to & T\ \# \\ T & \to & \texttt{nat} \\ & | & \texttt{(}\ T\ T\ \texttt{)} \\ & | & \texttt{lam}\ T \end{array} \tag{4.4}$$

Compute NULL, FIRST, and FOLLOW for this grammar. Solution □

Exercise 27. Here is the grammar of the LISP language:

$$\begin{array}{rcl} S & \to & E\ \# \\ E & \to & \texttt{sym} \\ & | & \texttt{(}\ L\ \texttt{)} \\ L & \to & \epsilon \\ & | & E\ L \end{array} \tag{4.5}$$

Compute NULL, FIRST, and FOLLOW for this grammar. Solution □

4.3 Top-Down Parsing

The idea of *top-down parsing* is to proceed by successive expansions of the leftmost non-terminal. We thus construct a leftmost derivation. We start from the start symbol S and use a *table* indicating, for a non-terminal X to be expanded and the first k characters of the input, the expansion $X \to \beta$ to be performed. Let's assume $k = 1$ in the following and denote this table as $T(X, c)$ for a given non-terminal X and terminal c. We also assume that a terminal symbol # denotes the end of the input and that the table T thus also indicates the expansion to be performed for this character.

Top-down parsing uses a stack, which is a word in $(N \cup T)^\star$. Initially, the stack is reduced to the start symbol S. At each moment, we examine the top of the stack and the first character c of the input, performing the following actions:

- if the stack is empty, we stop; there is success if and only if c is #.
- if the top of the stack is a terminal a, then a must be equal to c, we pop a and consume c; otherwise, we fail.
- if the top of the stack is a non-terminal X, then we replace X with the word $\beta = T(X, c)$ at the top of the stack, if applicable, by pushing the characters of β starting from the last; otherwise, we fail.

stack	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

Figure 4.3: Top-down parsing of the word `int+int*int`.

Let's illustrate this with an example. We revisit the example of grammar (4.3) for arithmetic expressions and provide an expansion table (we will explain later how to construct it).

$$
\begin{array}{rcl}
E & \to & T\,E' \\
E' & \to & \texttt{+}\,T\,E' \\
 & | & \epsilon \\
T & \to & F\,T' \\
T' & \to & \texttt{*}\,F\,T' \\
 & | & \epsilon \\
F & \to & \texttt{(}\,E\,\texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

	`+`	`*`	`(`	`)`	`int`	`#`
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		`int`	

Let's analyze the word `int+int*int` using this table. The different steps are illustrated in figure 4.3. Initially, the stack contains the start symbol E and the input contains the word to be analyzed followed by the terminal #. Our first decision involves the symbol E at the top of the stack and the symbol `int` at the beginning of the input. Since E is a non-terminal, we consult the table, which indicates to perform the expansion $E \to TE'$. We thus pop the symbol E and push TE', starting from the end. It is therefore T that ends up at the top of the stack. We consult the table again, this time with T and `int`. It indicates to proceed with the expansion $T \to FT'$. We thus pop T to push T' then F. A third table consultation indicates the expansion $F \to$ `int`. We thus pop F to push `int`. This time, we find ourselves with a terminal symbol, `int`, at the top of the stack. We verify that it matches the beginning of the input and, since it does, both are removed. The top of the stack becomes T' and the beginning of the input becomes `+`. The table indicates to perform the expansion $T' \to \epsilon$, which has the effect of popping T'. And so

on. We finally end up with an empty stack and input reduced to #, which completes our analysis successfully.

A top-down parser is very easy to program by introducing a function for each non-terminal of the grammar. Each function examines the input and, depending on the case, consumes it or recursively calls functions corresponding to other non-terminals, according to the expansion table. In our example, we would have five mutually recursive functions E, E', T, T', and F. Unknowingly, we had performed a top-down analysis for this grammar in section 4.1, with only three functions `parse_expr`, `parse_term`, and `parse_factor` corresponding to E, T, and F. The functions corresponding to E' and T' were directly expanded in the functions `parse_expr` and `parse_term`. The expansion table was not explicit: it was in the code of each function. The stack was also not explicit: it was implemented by the call stack. Of course, we could have made them explicit.

Constructing the Expansion Table. It remains to explain how to construct the expansion table from the grammar. The idea is simple: to decide whether to perform the expansion $X \to \beta$ when the first character of the input is c, we determine if c is part of the *first* characters of the words recognized by β, *i.e.*, if $c \in \text{FIRST}(\beta)$. A difficulty arises for a production such as $X \to \epsilon$. We must then also consider the set of characters that can *follow* X, *i.e.*, $\text{FOLLOW}(X)$. We thus define the table T as follows: for each production $X \to \beta$,

- we set $T(X, a) = \beta$ for all $a \in \text{FIRST}(\beta)$;
- if $\text{NULL}(\beta)$, we also set $T(X, a) = \beta$ for all $a \in \text{FOLLOW}(X)$.

Exercise 28. Calculate the expansion table of grammar (4.3) and verify that we indeed find the table given on page 71. □

Of course, nothing prevents us from ending up with an expansion table containing several different expansions in the same cell. This is not necessarily a symptom of an ambiguous grammar, but only of a grammar that does not lend itself to such top-down analysis. This is why we introduce the following definition.

Definition 16 (LL(1) grammar)**.** A grammar is said to be LL(1) if, in the previous table, there is at most one production in each cell. □

In this acronym, LL stands for "Left to right scanning, Leftmost derivation," which means that the input is scanned from left to right and that a leftmost derivation is constructed. The number 1 means that we examine only one character at the beginning of the input.

Exercise 29. Is the grammar of the λ-calculusfrom exercise 26 LL(1)? Solution □

Exercise 30. Is the grammar of LISP from exercise 27 LL(1)? Solution □

We often need to transform grammars to make them LL(1). In particular, a *left-recursive* grammar, *i.e.*, containing a production of the form $X \to X\alpha$, will never be LL(1). We must then remove the left recursion (direct or indirect). Similarly, we must factor productions that start with the same terminal (left factorization).

In conclusion, LL(1) parsers are relatively simple to write but require writing unnatural grammars. We will turn to another solution.

4.4 Bottom-Up Parsing

The idea is still to read the input from left to right, but we now seek to recognize right-hand sides of productions to build the derivation tree from the bottom up, hence the name bottom-up parsing.

Like top-down parsing, bottom-up parsing reads the input from left to right and manipulates a stack that is a word in $(T \cup N)^\star$. At each moment, two actions are possible:

- a *shift* operation, which consists of reading the first terminal of the input and pushing it;
- a *reduce* operation, which consists of recognizing at the top of the stack the right-hand side β of a production $X \to \beta$ and replacing β with X at the top of the stack.

In the initial state, the stack is empty. When no more actions are possible, the input is recognized if it has been fully read and if the stack is reduced to the axiom S.

We illustrate on the right an example of bottom-up parsing with the grammar:

$$\begin{array}{rcl} E & \to & E \texttt{ + } T \\ & | & T \\ T & \to & T \texttt{ * } F \\ & | & F \\ F & \to & \texttt{(} E \texttt{)} \\ & | & \texttt{int} \end{array}$$

and the word `int+int*int`. The right column indicates the operation performed at each step. We have here a sequence of operations leading to success, and the word is therefore recognized by the grammar.

stack	input	action
ϵ	`int+int*int`	shift
`int`	`+int*int`	reduce $F \to$ `int`
F	`+int*int`	reduce $T \to F$
T	`+int*int`	reduce $E \to T$
E	`+int*int`	shift
E`+`	`int*int`	shift
E`+int`	`*int`	reduce $F \to$ `int`
E`+`F	`*int`	reduce $T \to F$
E`+`T	`*int`	shift
E`+`T`*`	`int`	shift
E`+`T`*int`		reduce $F \to$ `int`
E`+`T`*`F		reduce $T \to T$`*`F
E`+`T		reduce $E \to E$`+`T
E		success

Of course, we could have made other decisions, such as performing a second shift operation after the very first one and then failing in the analysis. To be able to use bottom-up parsing in practice, we need a way to mechanize this decision, *i.e.*, a way to choose at each step between shift and reduce. As for top-down parsing, we will make our decision by examining only the first character of the input (but the idea generalizes to an arbitrary number of characters).

The idea is to construct a finite automaton from the grammar and to interleave states of this automaton between the different symbols on the bottom-up parsing stack. The stack thus takes the form:

$$s_0\, x_1\, s_1\, \ldots\, x_n\, s_n$$

where s_i is a state of the automaton and $x_i \in T \cup N$ as before. We also have an action table indexed by a state of the automaton and a character. At each step, we consider the first character a of the input and the action indicated by the table for this character and the state s_n located at the top of the stack.

- if the table indicates success or failure, we stop;
- if the table indicates a shift, then there must exist a transition $s_n \xrightarrow{a} s'$ in the automaton and we push the terminal a and the state s' successively;
- if the table indicates a reduce $X \to \alpha$, with α of length p, then we must find α at the top of the stack:

  $$s_0\, x_1\, s_1\, \ldots\, x_{n-p}\, s_{n-p}\, |\alpha_1\, s_{n-p+1} \ldots\, \alpha_p\, s_n.$$

  Moreover, there must exist a transition $s_{n-p} \xrightarrow{X} s$ in the automaton. We then pop α and push $X\, s$, so the stack becomes:

  $$s_0\, x_1\, s_1\, \ldots\, x_{n-p}\, s_{n-p}\, X\, s.$$

Let's now show how to construct the automaton and the table. To explain the process, we start by constructing an *asynchronous* automaton, *i.e.*, an automaton containing spontaneous transitions called ϵ-transitions and denoted $s_1 \xrightarrow{\epsilon} s_2$. Such a transition means that one can go from state s_1 to state s_2 spontaneously, without reading a character. The *states* of this asynchronous automaton have the form:

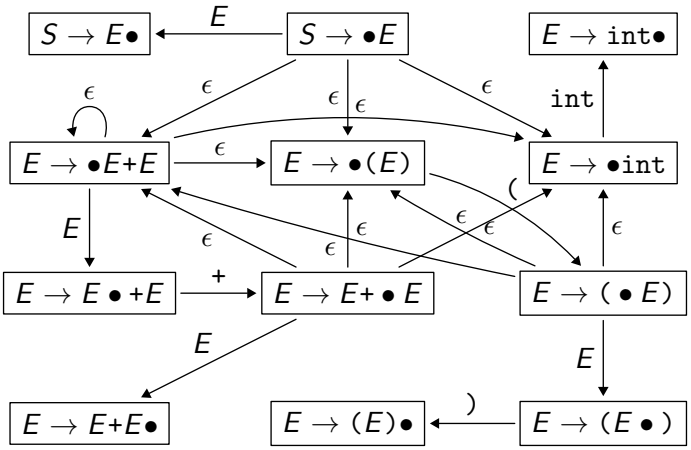
$$[X \to \alpha \bullet \beta]$$

where $X \to \alpha\beta$ is a production of the grammar. Such a state is called an *item*. The intuition of such a state is "I am trying to recognize X, I have already read α, and I still need to read β." The *transitions* of the automaton are labeled by $T \cup N$ and are of three forms:

$$\begin{array}{rcll} [Y \to \alpha \bullet a\beta] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta] & \text{for } a \in T, \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta] & \text{for } X \in N, \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{\epsilon} & [X \to \bullet\gamma] & \text{for every production } X \to \gamma. \end{array}$$

We start the construction with the states $[S \to \bullet\beta]$ for every production $S \to \beta$ of the grammar. Here is an example on a very simple (and ambiguous) grammar for arithmetic expressions:

$$\begin{array}{rcl} S & \to & E \\ E & \to & E\texttt{+}E \\ & | & \texttt{(}E\texttt{)} \\ & | & \texttt{int} \end{array}$$



The next step is to *determinize* this automaton. To do this, we group states connected (transitively) by ϵ-transitions, and the states thus become sets of *items*. Thus, the state $[S \to \bullet E]$, from which three spontaneous transitions departed, becomes the following set of four *items*:

$S \to \bullet E$
$E \to \bullet E$`+`E
$E \to \bullet$`(`E`)`
$E \to \bullet$`int`

In general, we obtain the states of the deterministic automaton by saturating them with the ϵ-transitions in the following sense: for a state s of the deterministic automaton and an item i belonging to s, if i is connected by ϵ to an item j in the non-deterministic automaton, then j also belongs to s.

Finally, we obtain the deterministic finite automaton given in the middle of figure 4.4. (We added at the end of the production $S \to E$ a new terminal symbol # to represent the end of input, as for top-down parsing.)

We can now construct the action table from this automaton. In practice, we work with two tables. On the one hand, we have an *action* table with rows for states and columns for terminals, the cell `action`(s, a) indicating:

- `shift` s' for a shift and a new state s';
- `reduce` $X \to \alpha$ for a reduce;
- success;
- failure.

On the other hand, we have a *goto* table with rows for states and columns for non-terminals, the cell `goto`(s, X) indicating the state resulting from a reduction of X. We thus construct these two tables. For the `action` table, we set:

- `action`$(s, \#)$ = success if $[S \to E \bullet \#] \in s$;
- `action`(s, a) = `shift` s' if there is a transition $s \xrightarrow{a} s'$;
- `action`(s, a) = `reduce` $X \to \beta$ if $[X \to \beta\bullet] \in s$, for all a;
- failure in all other cases.

For the `goto` table, we set:

- `goto`$(s, X) = s'$ if and only if there is a transition $s \xrightarrow{X} s'$.

In our example, we obtain the table given in figure 4.4 (at the bottom). The table thus constructed may contain several possible actions in the same cell. This is called a *conflict*. There are two types of conflicts:

- a *shift/reduce* conflict, if in a state s we can perform a shift but also a reduce;
- a *reduce/reduce* conflict, if in a state s two different reduces are possible.

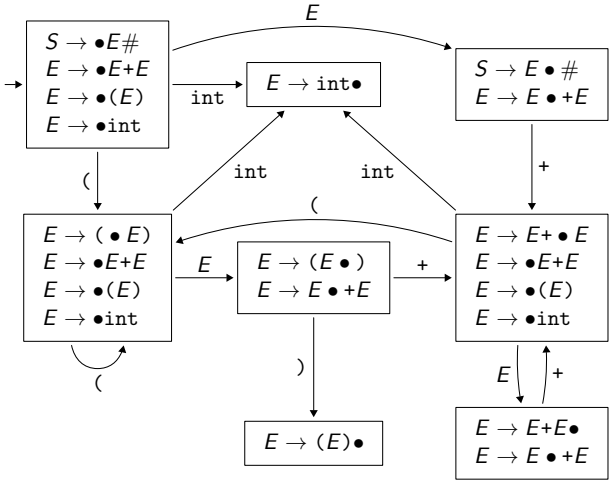
When there is no conflict, the grammar is said to be LR(0).

Definition 17 (LR(0) grammar)**.** A grammar is said to be LR(0) if the table thus constructed does not contain any conflict. (The acronym LR stands for "Left to right scanning, Rightmost derivation.")

a grammar

$$
\begin{array}{rcl}
S & \rightarrow & E \\
\\
E & \rightarrow & E\texttt{+}E \\
 & | & \texttt{(}E\texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

its LR(0) automaton



its LR(0) table

	action					*goto*
state	`(`	`)`	`+`	`int`	#	E
1	`shift` 4			`shift` 2		3
2	`reduce` $E \rightarrow$ `int`					
3			`shift` 6		success	
4	`shift` 4			`shift` 2		5
5		`shift` 7	`shift` 6			
6	`shift` 4			`shift` 2		8
7	`reduce` $E \rightarrow$ `(`E`)`					
8	`reduce` $E \rightarrow E$`+`E					

Figure 4.4: LR(0) parsing.

stack	input	action
1	`int+int+int`	s2
1 `int` 2	`+int+int`	$E \to$ `int`, g3
1 E 3	`+int+int`	s6
1 E 3 + 6	`int+int`	s2
1 E 3 + 6 `int` 2	`+int`	$E \to$ `int`, g8
1 E 3 + 6 E 8	`+int`	$E \to E$+E, g3
1 E 3	`+int`	s6
1 E 3 + 6	`int`	s2
1 E 3 + 6 `int` 2	#	$E \to$ `int`, g8
1 E 3 + 6 E 8	#	$E \to E$+E, g3
1 E 3	#	success

Figure 4.5: Analysis of the word `int+int+int`.

In our example, there is a conflict in state 8 (last line). If the input character is +, we can either shift this character or reduce the production $E \to E + E$. State 8 is the bottom right state of the automaton, *i.e.*, the one that contains both $E \to E$+$E\bullet$ and $E \to E \bullet$ +E. This conflict precisely illustrates the ambiguity of the grammar on a word such as `int+int+int`, after having read `int+int`. We can resolve the conflict in two ways: either we favor the *shift*, thus translating a right associativity; or we favor the *reduce*, thus translating a left associativity. Figure 4.5 illustrates the operation of the analysis on the word `int+int+int` assuming we have chosen the second option.

Exercise 31. Show that the grammar of the λ-calculus, given in exercise 26 page 70, is LR(0). Solution □

SLR(1) Parsing. The LR(0) construction very easily generates conflicts. We will therefore seek to limit reductions. A very simple idea is to set:

$$\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$$

if and only if:

$$[X \to \beta\bullet] \in s \quad \text{and} \quad a \in \textsc{follow}(X).$$

Definition 18 (SLR(1) class)**.** A grammar is said to be SLR(1) if the table thus constructed does not contain any conflict. (SLR stands for *Simple LR*.) □

Exercise 32. Show that the grammar:

$$\begin{array}{rcl} S & \to & E\# \\ E & \to & E\ \texttt{+}\ T \\ & | & T \\ T & \to & T\ \texttt{*}\ F \\ & | & F \\ F & \to & \texttt{(}\ E\ \texttt{)} \\ & | & \texttt{int} \end{array}$$

is SLR(1). (The automaton contains 12 states.) Solution □

Exercise 33. Is the grammar of LISP, given in exercise 27 page 70, SLR(1)? Solution □

Exercise 34. Show that the grammar:

$$\begin{array}{rcl} S & \to & E\# \\ E & \to & G = D \\ & | & D \\ G & \to & * D \\ & | & \texttt{id} \\ D & \to & G \end{array}$$

is not SLR(1). Solution □

LR(1) Parsing. In practice, the SLR(1) class remains too restrictive, as shown by the example in exercise 34. This is a realistic example, taken from the grammar of the C language. We therefore introduce an even larger class of grammars, LR(1), at the cost of even larger tables. The *items* now have the form:

$$[X \to \alpha \bullet \beta, a]$$

with the meaning "I am trying to recognize X, I have already read α and I still need to read β then check that the next character is a." The transitions of the non-deterministic LR(1) automaton are:

$$\begin{array}{rcll} [Y \to \alpha \bullet a\beta, b] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta, b] & \text{for } a \in T \\ [Y \to \alpha \bullet X\beta, b] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta, b] & \text{for } X \in N \\ [Y \to \alpha \bullet X\beta, b] & \xrightarrow{\epsilon} & [X \to \bullet\gamma, c] & \text{for } Y \to \beta \in R \text{ and } c \in \textsc{first}(\beta b) \end{array}$$

The initial state is the one that contains $[S \to \bullet\alpha, \#]$. As before, we can determinize the automaton and construct the corresponding table. We introduce a reduce action for (s, a) only when s contains an item of the form $[X \to \alpha\bullet, a]$.

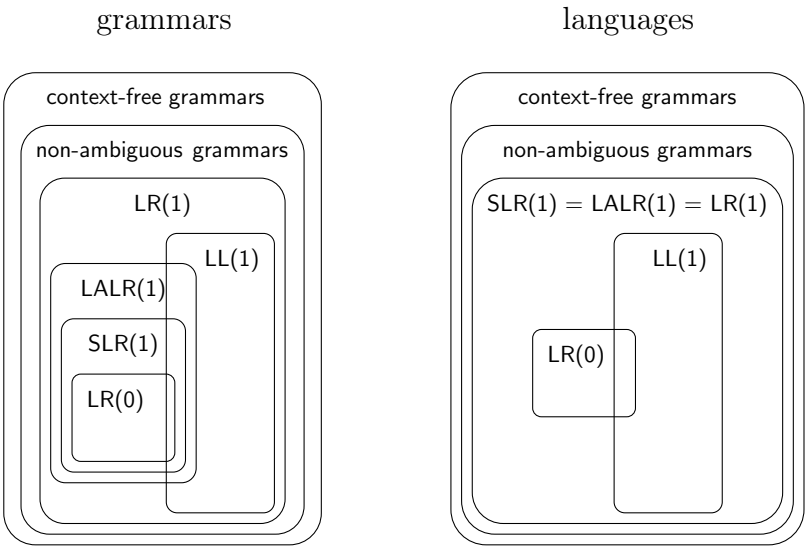
Definition 19 (LR(1) class)**.** A grammar is said to be LR(1) if the table thus constructed does not contain any conflict.

Exercise 35. Show that the grammar from exercise 34 is LR(1). Solution □

Other Classes. Since the LR(1) construction can be costly, there are approximations. The LALR(1) class (for *lookahead LR*) is one such approximation, used in particular in the tools of the `yacc` family, which we will discuss in the next section. We can seek to compare the different classes of grammars that we have introduced so far. We can also do this by interpreting a class of grammars as a class of languages, *i.e.*, by saying, for example, that a language is LL(1) if it is recognized by an LL(1) grammar. The comparisons of these classes of grammars and languages, in set-theoretic terms, are as follows:

```
%token PLUS TIMES LEFTPAR RIGHTPAR EOF
%token <int> CONST

%left PLUS
%left TIMES

%start <int> phrase

%%

phrase:
| e = expression; EOF  { e }

expression:
| e1 = expression; PLUS;  e2 = expression { e1 + e2 }
| e1 = expression; TIMES; e2 = expression { e1 * e2 }
| LEFTPAR; e = expression; RIGHTPAR       { e }
| i = CONST                               { i }
```

Figure 4.6: Syntactic analysis of arithmetic expressions with menhir.



These are strict inclusions. We note in particular that the LR(1) class is strictly more expressive than the LL(1) class, both in terms of grammars and in terms of languages.

4.5 The menhir Tool

Bottom-up parsing is powerful, but the computation of tables is complex. This is why it is automated by tools. This is the large family of yacc (YACC stands for *Yet Another Compiler Compiler*), which includes bison, ocamlyacc, cup, menhir, etc. Here, we choose to present the menhir tool, an LR(1) parser for OCaml. The menhir tool is used in conjunction with a lexical analyzer, typically built by ocamllex.

Figure 4.6 contains an example of a `menhir` source for the syntactic analysis of arithmetic expressions. The first two lines declare the tokens, namely `PLUS` (for `+`), `TIMES` (for `*`), `LEFTPAR` (for `(`), `RIGHTPAR` (for `)`), `CONST` (for a literal constant), and `EOF` for the end of input. The declaration of the token `CONST` is accompanied by the type of the value of this token, namely `int`. These tokens are identical in every respect to those of the lexical analyzer in figure 3.1.

The grammar rules appear after the line `%%`. There are two non-terminals, `phrase` and `expression`. The semicolons are optional; they are only there to aid reading. This grammar should be read as follows:

$$
\begin{array}{rcl}
\textit{phrase} & \rightarrow & \textit{expression}\ \texttt{EOF} \\
\textit{expression} & \rightarrow & \textit{expression}\ \texttt{PLUS}\ \textit{expression} \\
& | & \textit{expression}\ \texttt{TIMES}\ \textit{expression} \\
& | & \texttt{LEFTPAR}\ \textit{expression}\ \texttt{RIGHTPAR} \\
& | & \texttt{CONST}
\end{array}
$$

Each production is associated with an action, which is arbitrary OCaml code written between braces. Here, this code calculates the integer value of the expression that is recognized. In a more realistic example, this code would build an abstract syntax tree for the arithmetic expression. The syntax `e1 = ...` in the grammar rules allows retrieving the value of a terminal (in the case of `CONST` here) or the value recursively constructed by a non-terminal of the grammar (here `expression`). We can thus build the value of an arithmetic expression from the bottom up, retrieving the values of its sub-expressions.

If such a `menhir` source is contained in a file `arith.mly`, it is compiled with the command:

```
> menhir -v arith.mly
```

And we obtain OCaml code in two files `arith.ml` and `arith.mli`. This module exports the declaration of a type `token`,

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

and a function `phrase` of type:

```
val phrase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

The first is built from the `%token` declarations and the second from the `%start` declaration. Several `%start` declarations are possible and then give rise to several exported functions.

When the grammar is not LR(1), `menhir` reports the conflicts and presents them to the user in two files. The `.automaton` file contains a description of the LR(1) automaton and the conflicts are mentioned there. The `.conflicts` file contains an explanation of the conflicts, in the form of a sequence of tokens leading to two derivation trees. In the presence of conflicts, `menhir` makes an arbitrary choice between shift and reduce, but the user can indicate how to choose between shift and reduce. To do this, priorities are given to tokens and productions, and associativity rules are specified. By default, the priority of a production is that of its rightmost token (but it can be explicitly specified). If the priority of the production is higher than that of the token to be read, then the reduction is favored. Conversely, if the priority of the token is higher, then the shift is favored. In case of equality, the associativity is consulted: a left-associative token favors reduction and a right-associative token favors shift. In our example, we declared `PLUS`

and TIMES as left-associative with the %left declaration. Moreover, we declared TIMES as more prioritized than PLUS by placing %left TIMES lower than %left PLUS. From then on, all conflicts are resolved.

The menhir tool offers many advantages over traditional tools in the yacc family such as ocamlyacc. We refer to the menhir manual for more details [25].

Exercise 36. Write a menhir source for the following grammar:

$$\begin{array}{rcl} S & \to & E\ \texttt{EOF} \\ E & \to & \texttt{IF}\ E\ \texttt{THEN}\ E \\ & | & \texttt{IF}\ E\ \texttt{THEN}\ E\ \texttt{ELSE}\ E \\ & | & \texttt{CONST} \end{array}$$

This grammar has a conflict. Explain it. Propose a way to resolve it. Solution □

Bibliographical Notes. LR parsing was invented by Donald Knuth [15]. It is described in detail in section 4.7 of *Compilers: Principles, Techniques, and Tools* by A. Aho, R. Sethi, J. Ullman (known as the "dragon book") [1, 2].

5 Static Typing of Mini-ML

If you write an expression such as `"5" + 37` in a programming language, should you get a compilation error (as in OCaml), a runtime error (as in Python), the integer 42 (as in Visual Basic or PHP), the string `"537"` (as in Java), or something else? And if you add not just `"5"` and `37` but two arbitrary expressions, how do you determine if it is legal and what to do if it is? One answer is *typing*, an analysis that associates a *type* with each value or expression, with the aim of rejecting inconsistent programs.

Some languages are *dynamically typed*, meaning that types are associated with values and used during program execution, such as Lisp, PHP, Python, Julia, etc. Other languages are *statically typed*, meaning that types are associated with expressions and used during program compilation, such as C, C++, Java, OCaml, Rust, Go, etc. There are languages that use both static and dynamic typing. This is particularly the case for object-oriented languages, such as C++ or Java. We will return to this in chapter 6.

In this chapter, we focus solely on static typing. Static typing comes with a slogan attributed to Robin Milner:

well typed programs do not go wrong

This means that programs accepted by typing will execute without failure, in the sense of the operational semantics presented in section 2.2. For example, a well-typed program should not result in the use of an integer as a function. Beyond the safety property, typing must also have the property of being decidable. It would indeed not be reasonable for a compiler to not terminate due to typing, just as it would not be reasonable for it to respond "I don't know." Of course, it would be sufficient to reject all programs to automatically have the safety property, but this would not constitute a very interesting language. Typing must therefore also be *expressive*, not rejecting too many non-absurd programs. There is undeniable tension between the safety and expressiveness of typing. Thus, the OCaml language rejects a program such as:

```
(fun x -> x x) (fun x -> x x)
```

as being ill-typed, even though it actually poses no difficulty for the OCaml runtime engine.

In this chapter, we choose to illustrate static typing using the example of the Mini-ML language, already studied in chapter 2, whose abstract syntax we reproduce in figure 5.1.

$$
\begin{array}{rcll}
e & ::= & x & \text{identifier} \\
& | & c & \text{constant } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\
& | & \mathit{op} & \text{primitive } (+,\ \times,\ \mathit{fst},\ \ldots) \\
& | & \texttt{fun}\ x \to e & \text{anonymous function} \\
& | & e\ e & \text{application} \\
& | & (e,\ e) & \text{pair} \\
& | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{local binding}
\end{array}
$$

Figure 5.1: Abstract syntax of Mini-ML.

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ \text{etc.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}\ \text{etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Figure 5.2: Typing of Mini-ML with simple types.

5.1 Simple Typing

We will begin with a relatively simple typing of Mini-ML, with what are called *simple types*. A type is denoted by τ and its abstract syntax is:

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{base types} \\
& | & \tau \to \tau & \text{type of a function} \\
& | & \tau \times \tau & \text{type of a pair}
\end{array}
$$

Thus, $\texttt{int} \to \texttt{int} \times \texttt{int}$ is a type, namely the type of functions that take an integer as an argument and return a pair of integers. We do not specify here exactly what the base types are, as they depend on the constants and primitives chosen for Mini-ML.

To give a type to any expression, we will introduce the notion of a *typing judgment*. This is a ternary relation between a *typing environment* Γ, an expression e, and a type τ, which we denote as:

$$\Gamma \vdash e : \tau$$

and which is read as "in the environment Γ, the expression e has the type τ." The environment Γ is a function from variables to types. We can thus give a type to an expression e that is not necessarily closed, with each free variable x in e assumed to have the type $\Gamma(x)$ given by the environment.

We define the relation $\Gamma \vdash e : \tau$ by a set of inference rules, given in figure 5.2. In the typing rule for the `fun` and `let` constructions, we used the notation $\Gamma + x : \tau$ to denote an extension Γ' of the environment Γ defined by:

$$\Gamma'(y) = \begin{cases} \tau & \text{if } y = x, \\ \Gamma(y) & \text{otherwise.} \end{cases}$$

With these typing rules, we can show that the expression $\texttt{let}\ f = \texttt{fun}\ x \to +(x, 1)\ \texttt{in}\ f\ 2$ is well-typed with type `int` in an empty environment, with the following derivation (where some parts have been omitted due to space constraints):

$$\dfrac{\dfrac{\vdots\quad\dfrac{\vdots}{x : \texttt{int} \vdash (x, 1) : \texttt{int} \times \texttt{int}}}{x : \texttt{int} \vdash +(x, 1) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int} \to \texttt{int}} \qquad \dfrac{\dfrac{}{\cdots \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{}{\cdots \vdash 2 : \texttt{int}}}{f : \texttt{int} \to \texttt{int} \vdash f\ 2 : \texttt{int}}$$
$$\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}$$

On the other hand, we cannot type the program 1 2. Indeed, we would need to conclude by the application rule, *i.e.*,

$$\dfrac{\Gamma \vdash 1 : \tau' \to \tau \qquad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

and there is no rule to give the expression 1 a function type. Similarly, it is not possible to type the program $\texttt{fun}\ x \to x\ x$. For this, we would need a derivation of the form:

$$\dfrac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \to \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \texttt{fun}\ x \to x\ x : \tau_1 \to \tau_2}$$

with $\tau_3 = \tau_1$ and $\tau_1 = \tau_3 \to \tau_2$. However, there is no solution to these two equalities, as types are finite (which is implicit in their definition by an abstract syntax). We will see later a type system that allows typing this expression.

Some expressions admit multiple types. For instance, the expression $\texttt{fun}\ x \to x$ admits both the type $\texttt{int} \to \texttt{int}$ and the type $\texttt{bool} \to \texttt{bool}$. This is not, however, *polymorphism* (which we will discuss in detail later). Indeed, for a given occurrence of $\texttt{fun}\ x \to x$, a type must be *chosen*. Thus, the term $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ is not typable, as there is no type τ such that:

$$f : \tau \to \tau \vdash (f\ 1,\ f\ \texttt{true}) : \tau_1 \times \tau_2$$

Exercise 37. Show that a type can be given to $((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ 42$.

Solution □

In particular, we cannot give a satisfying type to a primitive such as `fst`; we would have to choose among an infinity of possible types:

$\texttt{int} \times \texttt{int} \to \texttt{int}$
$\texttt{int} \times \texttt{bool} \to \texttt{int}$
$\texttt{bool} \times \texttt{int} \to \texttt{bool}$
$(\texttt{int} \to \texttt{int}) \times \texttt{int} \to \texttt{int} \to \texttt{int}$
etc.

The same applies to the primitives *opif* and *opfix*. If we cannot give a satisfying type to *opfix*, we could, on the other hand, give a typing rule for a `let rec` construction that would be primitive:

$$\dfrac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

And if we wish to exclude recursive *values*, we can modify it as follows:

$$\frac{\Gamma + (f : \tau \to \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \qquad \Gamma + (f : \tau \to \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } f\ x = e_1 \texttt{ in } e_2 : \tau_2}$$

Exercise 38. Give the typing derivation of the expression:

$$\texttt{let rec } \mathit{fact}\ n\ =\ \texttt{if } =(n,\, 0) \texttt{ then } 1 \texttt{ else } \times (n,\, \mathit{fact}\ (+(n,\, -1))) \texttt{ in } \mathit{fact}\ 2.$$

Solution □

Typing Algorithm. So far, we have defined a *typing relation* between an expression and a type, but we do not have a *typing algorithm* that decides whether an expression is well-typed or not. In particular, a typing algorithm must choose a type to give to x when it encounters an expression of the form $\texttt{fun}\ x \to e$. Let's consider a simple approach where this type is provided by the user, for example in the form $\texttt{fun}\ x : \tau \to e$. This is how it is done in many statically typed languages, such as C or Java, where the formal parameters of a function are accompanied by their type. Then, it is sufficient to follow the rules of figure 5.2 to compute the type of an expression by recursion on the structure of this expression. By denoting $T(\Gamma, e)$ this type computation algorithm, we have:

$$\begin{array}{rcl}
T(\Gamma, x) & = & \Gamma(x) \\
T(\Gamma, n) & = & \texttt{int} \\
T(\Gamma, +) & = & \texttt{int} \times \texttt{int} \to \texttt{int} \\
T(\Gamma, \texttt{fun } x : \tau_1 \to e) & = & \tau_1 \to T(\Gamma + x : \tau_1, e) \\
T(\Gamma, (e_1,\, e_2)) & = & T(\Gamma, e_1) \times T(\Gamma, e_2) \\
T(\Gamma, \texttt{let } x = e_1 \texttt{ in } e_2) & = & T(\Gamma + x : T(\Gamma, e_1), e_2) \\
T(\Gamma, e_2\ e_1) & = & \tau_2 \quad \text{if } T(\Gamma, e_2) = T(\Gamma, e_1) \to \tau_2
\end{array}$$

Because the typing algorithm proceeds recursively on the syntax of expressions, without needing to make choices or backtrack, it is said to be *syntax-directed*. We note that only the last case involves a check, namely that the type of e_2 is indeed a function type expecting an argument of the type of e_1. In all other cases, the expression is well-typed as long as its sub-expressions are well-typed.

Exercise 39. Program the above typing algorithm in OCaml. Solution □

In practice, a compiler needs to keep the types of all expressions in the program for the needs of its subsequent phases. For this reason, we do not write a function that computes the type of an expression, as here, but a function that returns new abstract syntax trees, in which each sub-expression is decorated with its type. An efficient way to proceed in OCaml is to introduce two mutually recursive types, namely a record type for the decoration itself and an algebraic type for the different constructions of the abstract syntax.

```
type expression = { node: node; typ: typ; }
and node =
  | Var of string
  | App of expression * expression
  | ...
```

5.2 Type Safety

Now that we have defined the typing relation for Mini-ML, we can seek to verify whether Robin Milner's slogan, that well-typed programs do not go wrong, is valid for Mini-ML. To do this, we will use the small-step operational semantics defined in section 2.2.2 and show the following result:

> If $\emptyset \vdash e : \tau$, then the evaluation of e is infinite or terminates on a value.

The proof of this theorem relies on two lemmas, called *progress* and *preservation*. The first states that a well-typed expression that is not a value can progress in its computation, by performing at least one reduction step. The second states that the type of an expression is preserved by a reduction step. Let's start with the progress lemma.

Lemma 4 (progress)**.** If $\emptyset \vdash e : \tau$, then either e is a value, or there exists e' such that $e \to e'$.

Proof. We proceed by induction on the typing derivation $\emptyset \vdash e : \tau$. Suppose, for example, that $e = e_2\ e_1$; we therefore have:

$$\frac{\emptyset \vdash e_2 : \tau_1 \to \tau_2 \quad \emptyset \vdash e_1 : \tau_1}{\emptyset \vdash e_2\ e_1 : \tau_2}$$

We apply the induction hypothesis to e_2:

- if $e_2 \to e_2'$, then $e_2\ e_1 \to e_2'\ e_1$ by the context passage lemma 1 (page 35);
- if e_2 is a value, suppose $e_2 = \texttt{fun}\ x \to e_3$ (there is also $+$, etc.). We apply the induction hypothesis to e_1:
  - if $e_1 \to e_1'$, then $e_2\ e_1 \to e_2\ e_1'$ (same lemma);
  - if e_1 is a value, then $e_2\ e_1 \to e_3[x \leftarrow e_1]$.

The other cases are left as an exercise. □

Let's now move on to the preservation lemma. To show it, we will need several lemmas about typing derivations. The first states that the order of additions in Γ is not important.

Lemma 5 (permutation)**.** If $\Gamma + x : \tau_1 + y : \tau_2 \vdash e : \tau$ and $x \neq y$, then $\Gamma + y : \tau_2 + x : \tau_1 \vdash e : \tau$ and the derivation has the same height.

Proof. By immediate induction on the typing derivation. □

The following lemma states that we can add useless hypotheses to Γ without modifying the typing relation.

Lemma 6 (weakening)**.** If $\Gamma \vdash e : \tau$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma + x : \tau' \vdash e : \tau$ and the derivation has the same height.

Proof. Again, by immediate induction on the typing derivation. □

For these two lemmas, we took care to add "and the derivation has the same height" to the result, which will allow us later to apply induction hypotheses to derivations obtained thanks to these lemmas. We continue with a *key lemma* that shows that typing is preserved by certain substitutions.

Lemma 7 (preservation by substitution)**.** If $\Gamma + x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. We proceed by induction on the derivation $\Gamma + x : \tau' \vdash e : \tau$.

- case of a variable $e = y$:
  - if $x = y$, then $e[x \leftarrow e'] = e'$ and $\tau = \tau'$;
  - if $x \neq y$, then $e[x \leftarrow e'] = e$ and $\tau = \Gamma(y)$.
- case of an abstraction $e = \texttt{fun}\ y \to e_1$:

  We can assume $y \neq x$, $y \notin \mathrm{dom}(\Gamma)$, and y not free in e', possibly by renaming y. We have $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ and therefore $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$ (permutation). Moreover, $\Gamma \vdash e' : \tau'$ and therefore $\Gamma + y : \tau_2 \vdash e' : \tau'$ (weakening). By the induction hypothesis, we therefore have $\Gamma + y : \tau_2 \vdash e_1[x \leftarrow e'] : \tau_1$ and therefore $\Gamma \vdash (\texttt{fun}\ y \to e_1)[x \leftarrow e'] : \tau_2 \to \tau_1$, *i.e.*, $\Gamma \vdash e[x \leftarrow e'] : \tau$.

The other cases are left as an exercise. □

We are now able to show the preservation lemma.

Lemma 8 (preservation)**.** If $\emptyset \vdash e : \tau$ and $e \to e'$, then $\emptyset \vdash e' : \tau$.

Proof. We proceed by induction on the derivation $\emptyset \vdash e : \tau$.

- case $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$:

$$\frac{\emptyset \vdash e_1 : \tau_1 \qquad x : \tau_1 \vdash e_2 : \tau_2}{\emptyset \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

  - if $e_1 \to e_1'$, by the induction hypothesis we have $\emptyset \vdash e_1' : \tau_1$ and therefore $\emptyset \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau_2$;
  - if e_1 is a value and $e' = e_2[x \leftarrow e_1]$, then we apply the preservation by substitution lemma.
- case $e = e_1\ e_2$:
  - if $e_1 \to e_1'$ or if e_1 is a value and $e_2 \to e_2'$, then we use the induction hypothesis;
  - if $e_1 = \texttt{fun}\ x \to e_3$ and e_2 is a value, then $e' = e_3[x \leftarrow e_2]$ and we again apply the substitution lemma.

The other cases are left as an exercise. □

The type safety theorem is an easy consequence of the progress and preservation lemmas. Rather than stating it in the form " If $\emptyset \vdash e : \tau$, then the evaluation of e is infinite or terminates on a value" given above, we choose another equivalent formulation.

Theorem 2 (type safety)**.** If $\emptyset \vdash e : \tau$ and $e \xrightarrow{\star} e'$ with e' irreducible, then e' is a value.

Proof. We have $e \to e_1 \to \cdots \to e'$ and by repeated applications of the preservation lemma, we therefore have $\emptyset \vdash e' : \tau$. By the progress lemma, e' reduces or is a value. It is therefore a value. □

5.3 Parametric Polymorphism

It is restrictive to give a unique type to `fun` $x \to x$ in an expression like `let` $f =$ `fun` $x \to x$ `in` e because we would like to be able to use the function f several times in the expression e, with arguments of different types. Similarly, we would like to be able to give several types to primitives such as *fst* or *snd*. A solution to this problem is called *parametric polymorphism*. It consists in extending the algebra of types as follows:

$$\begin{array}{rcll} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{base types} \\ & \mid & \tau \to \tau & \text{type of a function} \\ & \mid & \tau \times \tau & \text{type of a pair} \\ & \mid & \alpha & \text{type variable} \\ & \mid & \forall \alpha.\tau & \text{polymorphic type} \end{array}$$

To the three existing type constructions, we have added two new constructions: type variables, denoted in the following by Greek letters, and universally quantified types. Intuitively, a type variable α represents any type, and a universally quantified type $\forall \alpha.\tau$ is the type of a value, called polymorphic, which can take the type τ whatever the type α. The construction $\forall \alpha.\tau$ is a binder, which binds the variable α in the type τ. In particular, it can be renamed at will, exactly like a program variable introduced by `fun` or `let`. Unsurprisingly, there is therefore a notion of free variable in a type, analogous to that of free variable in an expression.

Definition 20 (free variables of a type)**.** We denote $\mathcal{L}(\tau)$ the set of type variables *free* in τ, defined by:

$$\begin{array}{rcl} \mathcal{L}(\texttt{int}) & = & \emptyset \\ \mathcal{L}(\alpha) & = & \{\alpha\} \\ \mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\forall \alpha.\tau) & = & \mathcal{L}(\tau) \setminus \{\alpha\} \end{array}$$

For a typing environment Γ, we set:

$$\mathcal{L}(\Gamma) = \bigcup_{x \in \mathrm{dom}(\Gamma)} \mathcal{L}(\Gamma(x)).$$

□

If a type variable is free in a type, we can seek to substitute it with another type, in a manner analogous to the substitution of a variable by a value in an expression. As with the latter, we take care to stop the substitution when we encounter a bound variable with the same name as the variable to be replaced.

Definition 21 (type substitution)**.** We denote $\tau[\alpha \leftarrow \tau']$ the substitution of α by τ' in τ, defined by:

$$\begin{array}{rcl} \texttt{int}[\alpha \leftarrow \tau'] & = & \texttt{int} \\ \alpha[\alpha \leftarrow \tau'] & = & \tau' \\ \beta[\alpha \leftarrow \tau'] & = & \beta \quad \text{if } \beta \neq \alpha \\ (\tau_1 \to \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \to \tau_2[\alpha \leftarrow \tau'] \\ (\tau_1 \times \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \times \tau_2[\alpha \leftarrow \tau'] \\ (\forall \alpha.\tau)[\alpha \leftarrow \tau'] & = & \forall \alpha.\tau \\ (\forall \beta.\tau)[\alpha \leftarrow \tau'] & = & \forall \beta.\tau[\alpha \leftarrow \tau'] \quad \text{if } \beta \neq \alpha \end{array}$$

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ etc.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}} \text{ etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

Figure 5.3: Typing of Mini-ML with System F.

□

The typing rules in this new type system are given in figure 5.3. These are *exactly* the same rules as before (figure 5.2 page 84), to which we add two new rules (on the last line). The first states that the type of an expression can be *generalized* with respect to a type variable α as long as this variable does not appear in the context Γ.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau}$$

The second states that a polymorphic type can be *specialized* by any type.

$$\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

The system thus obtained is called *System F*. We note that this system is not syntax-directed. Indeed, three rules now potentially apply to each expression: the rule of the old system, the generalization rule, and the specialization rule.

We can now type the expression $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ which it was not possible to type before with simple types.

$$\dfrac{\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \to x : \forall\alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\cdots \vdash f : \forall\alpha.\alpha \to \alpha}{\cdots \vdash f : \texttt{int} \to \texttt{int}}}{\cdots \vdash f\ 1 : \texttt{int}} \quad \vdots \qquad \dfrac{\vdots}{\cdots \vdash f\ \texttt{true} : \texttt{bool}}}{f : \forall\alpha.\alpha \to \alpha \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

To achieve this, we gave the function $\texttt{fun}\ x \to x$ the polymorphic type $\forall\alpha.\alpha \to \alpha$, which allowed us to use it both on an integer, by specializing with the type `int`, and on a Boolean, by specializing with the type `bool`.

We can now give a satisfying type to the primitives of Mini-ML:

$$\begin{aligned} \mathit{fst} &: \ \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\ \mathit{snd} &: \ \forall\alpha.\forall\beta.\alpha \times \beta \to \beta \\ \mathit{opif} &: \ \forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha \\ \mathit{opfix} &: \ \forall\alpha.(\alpha \to \alpha) \to \alpha \end{aligned}$$

Exercise 40. Give a typing derivation for the expression $\texttt{fun}\ x \to x\ x$. Solution □

Remark. The condition $\alpha \notin \mathcal{L}(\Gamma)$ in the generalization rule is crucial. Without it, we could type $\texttt{fun}\ x \to x$ with the type $\alpha \to \forall\alpha.\alpha$, as follows:

$$\dfrac{\dfrac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall\alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \forall\alpha.\alpha}$$

and successfully type the expression $(\texttt{fun}\ x \to x)\ 1\ 2$, *i.e.*, a program whose execution results in the use of an integer as a function. Type safety would therefore not be guaranteed. □

We can show that System F is a safe type system for Mini-ML, as we did above for simple types, even if the proof is more complex. Unfortunately, we do not easily deduce a typing algorithm from it. Indeed, it turns out that the problem of *type inference* (given e, does there exist τ such that $\vdash e : \tau$?) as well as that of verification (given e and τ, do we have $\vdash e : \tau$?) are not decidable. To obtain a decidable type inference, we will restrict the power of System F.

Hindley-Milner System. A solution is provided by the Hindley-Milner system. It consists in limiting universal quantification $\forall$ to the head of types; this is called prenex quantification, as in logic. To do this, we distinguish types without quantification, still denoted as τ, from types that can be universally quantified, called *schemes* and denoted as σ.

$$\begin{array}{rcll} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{base types} \\ & \mid & \tau \to \tau & \text{type of a function} \\ & \mid & \tau \times \tau & \text{type of a pair} \\ & \mid & \alpha & \text{type variable} \\ \\ \sigma & ::= & \tau & \text{schemes} \\ & \mid & \forall\alpha.\sigma & \end{array}$$

In the Hindley-Milner system, the following types are always accepted:

$$\begin{array}{l} \forall\alpha.\alpha \to \alpha \\ \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\ \forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha \\ \forall\alpha.(\alpha \to \alpha) \to \alpha \end{array}$$

But no longer types such as:

$$(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha).$$

A typing environment Γ associates type schemes with variables, and the typing relation now has the form $\Gamma \vdash e : \sigma$. The rules of the Hindley-Milner system are given in figure 5.4.

We note that specialization replaces a type variable α with a type τ and not a scheme, otherwise the resulting scheme would be ill-formed. We also note that only the `let` construction allows introducing a polymorphic type into the environment. In particular, we can still type:

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ etc.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}} \text{ etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

Figure 5.4: Typing of Mini-ML with Hindley-Milner.

with $f : \forall \alpha.\alpha \to \alpha$ in the context to type $(f\ 1, f\ \texttt{true})$. On the other hand, the typing rule:

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2}$$

does not introduce a polymorphic type, because otherwise $\tau_1 \to \tau_2$ would be ill-formed. In particular, we can no longer type $\texttt{fun}\ x \to x\ x$.

Remark. In a language like OCaml, which implements the Hindley-Milner system, universal quantification is implicit because it is necessarily prenex. Thus, a type presented as:

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

must be read as:

$$\forall \alpha.\, \forall \beta.\, (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathit{list} \to \alpha$$

5.4 Type Inference

Let's try to program a type inference for the Hindley-Milner system. Naturally, we will try to proceed by recursion on the structure of the program. However, for a given expression, three rules can apply: the rule of the monomorphic system, the generalization rule:

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}$$

or the specialization rule:

$$\frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

It seems difficult to choose between the three, because we cannot predict the use that will be made of the expression once typed. And we do not want to proceed by trial and error, because typing might end up being too costly. To achieve our goal, we will modify

$$\frac{x \in dom(\Gamma) \quad \tau \leq \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ etc.} \qquad \frac{\tau \leq type(op)}{\Gamma \vdash op : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : Gen(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Figure 5.5: Typing of Mini-ML with Hindley-Milner, syntax-directed.

the presentation of the Hindley-Milner system so that it is *syntax-directed*, *i.e.*, so that, for any expression, at most one rule applies. The rules will have the same expressive power: any closed term is typable in one system if and only if it is typable in the other.

To define this new presentation, we need two new ingredients. On the one hand, we introduce the relation $\tau \leq \sigma$ whose meaning is "τ is a specialization of the scheme σ". Formally, this relation is defined by:

$$\tau \leq \forall\alpha_1 \ldots \alpha_n.\tau' \quad \text{if and only if} \quad \exists\tau_1 \ldots \exists\tau_n.\ \tau = \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n].$$

For example, we have:

$$\texttt{int} \times \texttt{bool} \to \texttt{int} \leq \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha.$$

On the other hand, we introduce a *generalization* operation, denoted as $Gen(\tau, \Gamma)$, which constructs a type scheme by quantifying τ with respect to all its free variables not appearing in Γ, *i.e.*,

$$Gen(\tau, \Gamma) \stackrel{\text{def}}{=} \forall\alpha_1 \ldots \forall\alpha_n.\tau \quad \text{where} \quad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau) \setminus \mathcal{L}(\Gamma).$$

Equipped with these two ingredients, we can define the syntax-directed Hindley-Milner system. Its rules are given in figure 5.5. The difference from the previous system lies in generalization and specialization only. Rather than proposing them as two new rules, they are placed at strategic points. On the one hand, specialization is done at the level of axioms, when using a variable or a primitive. We assume here that a function *type* gives the type of a primitive. Thus, we have:

$$type(fst) = \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

and therefore, in particular, $\Gamma \vdash fst : \texttt{int} \times \texttt{bool} \to \texttt{int}$ with this rule. On the other hand, generalization is done at the level of `let`, the only construction capable of introducing a scheme into the environment, by generalizing as much as possible.

We can still type the expression $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$, but this time the derivation is entirely syntax-directed:

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\texttt{int} \to \texttt{int} \leq \forall\alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\Gamma \vdash f\ 1 : \texttt{int}} \qquad \dfrac{\dfrac{\texttt{bool} \to \texttt{bool} \leq \forall\alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{bool} \to \texttt{bool}} \quad \vdots}{\Gamma \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

by setting $\Gamma = f : \mathit{Gen}(\alpha \to \alpha, \emptyset) = f : \forall\alpha.\alpha \to \alpha$. Though this derivation is syntax-directed, it still contains a bit of magic: to type $\texttt{fun}\ x \to x$ one must choose a type to give to x (here we chose α) and to type f one must choose a certain instance of $\forall\alpha.\alpha \to \alpha$ (here we chose $\texttt{int} \to \texttt{int}$ the first time and $\texttt{bool} \to \texttt{bool}$ the second). These two problems arise in general, for any `fun` construction on the one hand and for any use of a variable (or a polymorphic primitive) on the other. The problem is not trivial, because there is an arbitrary distance between the place where a function is typed and the place where it is used, just as between the place where a variable is typed and the place where the chosen type comes into play. Fortunately, there is a solution to these two problems, known as *Algorithm W*.

Algorithm W. The idea of Algorithm W is to defer the choice of types until the only moment when this choice has a real impact, *i.e.*, at the time of application, when we check that the argument passed to a function indeed has the type expected by this function. To do this, we use *new type variables* to represent types that are still unknown, namely the type of x in $\texttt{fun}\ x \to e$ on the one hand and the types with which to specialize the scheme $\Gamma(x)$ when using a variable x on the other. These are not type variables in the usual sense, but *unknowns*, likely to later receive values more precise than type variables. Their resolution is done at the time of typing applications, by unifying the type of the argument and the type expected by the function. Such a *unification* problem can be posed in the following terms: given two types τ_1 and τ_2 containing type variables $\alpha_1, \ldots, \alpha_n$, is there a specialization θ, *i.e.*, a function from the variables α_i to types, such that $\theta(\tau_1) = \theta(\tau_2)$? Thus, the unification problem:

$$\alpha \times \beta \to \texttt{int} \stackrel{?}{=} \texttt{int} \times \texttt{bool} \to \gamma$$

admits a solution, namely $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}, \gamma \mapsto \texttt{int}\}$. Similarly, the unification problem:

$$\alpha \times \texttt{int} \to \alpha \times \texttt{int} \stackrel{?}{=} \gamma \to \gamma$$

admits a solution, namely $\{\gamma \mapsto \alpha \times \texttt{int}\}$. On the other hand, the problem:

$$\alpha \to \texttt{int} \stackrel{?}{=} \beta \times \gamma$$

does not admit a solution, because a type $\to$ cannot be equal to a type $\times$, just as the problem:

$$\alpha \to \texttt{int} \stackrel{?}{=} \alpha$$

does not admit a solution, because types are finite.

The pseudo-code of a unification algorithm is given in figure 5.6. This code is deliberately written in an imperative style, in the sense that a call to $\mathit{unify}(\tau_1, \tau_2)$ leads to a global resolution, by side effect, of certain type variables. This is called *destructive unification*. Many other ways of writing it are possible, including in a purely applicative style. This algorithm gives us the *most general unifier* (or mgu), in the sense that any specialization θ such that $\theta(\tau_1) = \theta(\tau_2)$ is an instance of the result of $\mathit{unify}(\tau_1, \tau_2)$.

Before describing Algorithm W, let's start by illustrating the idea with an example, namely the expression $\texttt{fun}\ x \to +(\mathit{fst}\ x, 1)$. Since it is a function, we give x the type α_1, a new type variable. In the environment $x : \alpha_1$, we now seek to type the expression

$$
\begin{array}{rcl}
\mathit{unify}(\tau, \tau) & = & \text{success} \\
\mathit{unify}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') & = & \mathit{unify}(\tau_1, \tau_2)\ ;\ \mathit{unify}(\tau_1', \tau_2') \\
\mathit{unify}(\tau_1 \times \tau_1', \tau_2 \times \tau_2') & = & \mathit{unify}(\tau_1, \tau_2)\ ;\ \mathit{unify}(\tau_1', \tau_2') \\
\mathit{unify}(\alpha, \tau) & = & \text{if } \alpha \notin \mathcal{L}(\tau), \text{ replace } \alpha \text{ with } \tau \text{ everywhere} \\
 & & \text{otherwise, failure} \\
\mathit{unify}(\tau, \alpha) & = & \mathit{unify}(\alpha, \tau) \\
\mathit{unify}(\tau_1, \tau_2) & = & \text{failure in all other cases}
\end{array}
$$

Figure 5.6: Pseudo-code of the unification algorithm.

$+(\mathit{fst}\ x, 1)$. The primitive $+$ has the type $\texttt{int} \times \texttt{int} \to \texttt{int}$. Let's type its argument, namely the expression $(\mathit{fst}\ x, 1)$. The primitive *fst* has the scheme $\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$, which must therefore be specialized. We thus give it the type $\alpha_2 \times \beta_1 \to \alpha_2$, for two new type variables α_2 and β_1. The application $\mathit{fst}\ x$ requires unifying α_1 and $\alpha_2 \times \beta_1$, which gives $\{\alpha_1 \mapsto \alpha_2 \times \beta_1\}$. The expression $(\mathit{fst}\ x, 1)$ thus has the type $\alpha_2 \times \texttt{int}$. Then, the application $+(\mathit{fst}\ x, 1)$ unifies $\texttt{int} \times \texttt{int}$ and $\alpha_2 \times \texttt{int}$, which gives $\{\alpha_2 \mapsto \texttt{int}\}$. In the end, we obtain the type $\texttt{int} \times \beta_1 \to \texttt{int}$, *i.e.*,

$$\vdash \texttt{fun}\ x \to +(\mathit{fst}\ x, 1) : \texttt{int} \times \beta \to \texttt{int}$$

The remaining type variable, β_1, is no longer an unknown, but a true type variable, which we have renamed here as β. The pseudo-code of Algorithm W is given in figure 5.7.

We note that the case of a constant c involves a trivial instance. We can indeed have polymorphic constants, such as the empty list.

Theorem 3. Algorithm W is correct, in the sense that:

$$\text{if } W(\emptyset, e) = \tau \text{ then } \emptyset \vdash e : \tau,$$

and it determines the " most general possible" type, also called the *principal type*, in the sense that:

$$\text{if } \emptyset \vdash e : \tau \text{ then } \tau \leq \mathit{Gen}(W(\emptyset, e), \emptyset).$$

□

Moreover, we can show that the Hindley-Milner system is safe, in the sense of Theorem 2, *i.e.*, if $\emptyset \vdash e : \tau$, then the reduction of e is infinite or terminates on a value.

The Problem of Polymorphic References. Suppose we want to add references to Mini-ML, as in OCaml, with a construction *ref* e to create a new reference, a construction $!e$ to access the content of a reference, and a construction $e_1 := e_2$ to modify the content of a reference. One might think that it is sufficient to see these three constructions as so many new primitives, to which we would give the following polymorphic types:

$$
\begin{array}{rcl}
\mathit{ref} & : & \forall\alpha.\alpha \to \alpha\ \mathit{ref} \\
! & : & \forall\alpha.\alpha\ \mathit{ref} \to \alpha \\
:= & : & \forall\alpha.\alpha\ \mathit{ref} \to \alpha \to \mathit{unit}
\end{array}
$$

$W(\Gamma, e) \stackrel{\text{def}}{=}$

- if e is a variable x,
  return a trivial instance of $\Gamma(x)$
- if e is a constant c,
  return a trivial instance of its type
- if e is a primitive op,
  return a trivial instance of its type
- if e is a pair (e_1, e_2),
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  return $\tau_1 \times \tau_2$
- if e is a function `fun` $x \to e_1$,
  let α be a new variable
  compute $\tau = W(\Gamma + x : \alpha, e_1)$
  return $\alpha \to \tau$
- if e is an application $e_1\ e_2$,
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  let α be a new variable
  $\mathit{unify}(\tau_1, \tau_2 \to \alpha)$
  return α
- if e is `let` $x = e_1$ `in` e_2,
  compute $\tau_1 = W(\Gamma, e_1)$
  return $W(\Gamma + x : \mathit{Gen}(\tau_1, \Gamma), e_2)$

Figure 5.7: Pseudo-code of Algorithm W.

where τ *ref* is the type of a reference containing a value of type τ. However, it turns out that this is incorrect, in the sense that type safety is no longer guaranteed. A counterexample is the following program:

$$\begin{array}{l}\texttt{let } r = \mathit{ref}\ (\texttt{fun } x \to x) \texttt{ in}\\ \texttt{let } \_ = r := (\texttt{fun } x \to x + 1) \texttt{ in}\\ !r\ (\texttt{fun } y \to y)\end{array}$$

It is well-typed in the Hindley-Milner system. Indeed, the first line gives r the polymorphic type $\forall\alpha.(\alpha \to \alpha)$ *ref*. Then, the assignment is accepted, because the polymorphic type of r can be specialized to $(\texttt{int} \to \texttt{int})$ *ref*. Finally, the last line is also well-typed, because the type of r can be specialized again, this time with the type $((\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}))$ *ref*. But this program does not execute correctly, because it results in the addition of the function $\texttt{fun } y \to y$ and the integer 1.

This problem, called the *polymorphic references* problem, admits several solutions. One of the simplest consists in generalizing the type of e_1 in $\texttt{let } x = e_1 \texttt{ in } e_2$ only when e_1 is syntactically a value[1]. From then on, we can no longer write:

$$\texttt{let } r = \mathit{ref}\ (\texttt{fun } x \to x) \texttt{ in } \ldots$$

because *ref* $(\texttt{fun } x \to x)$ is not a value, but we can write instead:

$$(\texttt{fun } r \to \ldots)\ (\mathit{ref}\ (\texttt{fun } x \to x))$$

where the type of r is not generalized. In practice, we can continue to write $\texttt{let } r =$ *ref* $\ldots$ $\texttt{in}$ $\ldots$ but without generalizing the type of r in this case. This is notably what OCaml does.

```
# let x = ref (fun x -> x);;
val x : ('_a -> '_a) ref
```

Here, `'_a` denotes a type that we do not yet know and not a type variable like `'a` that would be quantified. If, for example, we apply `!x` to an integer, we then determine `'_a` as being `int` and we can no longer apply `!x` to anything other than integers.

Exercise 41. Explain why the following OCaml program is ill-typed:

```
let map_id = List.map (fun x -> x)
let l1 = map_id [1; 2; 3]
let l2 = map_id [true; false; true]
```

Propose a solution. Solution □

Bibliographical Notes. The type of proof carried out in section 5.2 was introduced in *A Syntactic Approach to Type Soundness* [31]. Benjamin Pierce's book *Types and Programming Languages* [23] contains several examples. System F is due independently to J.-Y. Girard and J. C. Reynolds. The result of undecidability of typing in System F was shown in 1994 by J. B. Wells [30]. Algorithm W is due to Damas and Milner [6]. The problem of polymorphic references was detected in 1990 [27] and required corrections of versions of SML that presented this problem. The solution implemented in OCaml is relatively sophisticated [8], notably to allow indicating which are the type variables of an abstract type that can be generalized.

[1] This is called the *value restriction*.

6 Object-Oriented Languages

Object-oriented languages emerged in the 1960s with Simula I and Simula 67, and further developed with Smalltalk (1972), followed by C++ (1983) and Java (1995). In this chapter, we explain what an object-oriented language is and how it can be compiled, particularly how to materialize an object in memory and how to perform a method call. We primarily use Java for illustration purposes, and then C++ at the end of the chapter to highlight some notable differences between the two.

6.1 Introduction to Object-Oriented Concepts with Java

The first object-oriented concept is that of a *class*. The declaration of a class introduces a new type. As a first approximation, a class can be seen as a record.

```
class Polar {
  double rho;
  double theta;
}
```

Here, `rho` and `theta` are the two *fields* of the `Polar` class, of type `double`. A particular *instance* of a class, called an *object*, is created using the `new` construct. For example,

```
  Polar p = new Polar();
```

declares a new local variable `p` of type `Polar`, whose value is a new instance of the `Polar` class. The object is allocated on the heap. Its fields are initialized here with default values (in this case, 0). The fields of `p` can be accessed and modified using the usual notation `p.rho` and `p.theta`:

```
  p.rho = 2;
  p.theta = 3.14159265;
  double x = p.rho * Math.cos(p.theta);
  p.theta = p.theta / 2;
```

One or more *constructors* can be introduced to initialize the object's fields. A constructor has the same name as the class.

```
class Polar {
  double rho, theta;
  Polar(double r, double t) {
    if (r < 0) throw new Error("Polar: negative length");
    rho = r;
    theta = t;
  }
}
```

We can then write[1] for example:

```
  Polar p = new Polar(2, 3.14159265);
```

Suppose we now want to maintain the following *invariant* for all objects of the `Polar` class:

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi.$$

To achieve this, we declare the fields `rho` and `theta` as *private*, so that they are no longer visible outside the `Polar` class.

```
class Polar {
  private double rho, theta;
  Polar(double r, double t) { /* ensures the invariant */ }
}
```

If we now attempt to access the `rho` field from another class:

```
  p.rho = 1;
```

we get a typing error:

```
 complex.java:19: rho has private access in Polar
```

However, the value of the `rho` field can still be provided through a *method*, *i.e.*, a function provided by the `Polar` class and applicable to any object of this class.

```
class Polar {
  private double rho, theta;
  ...
  double norm() { return rho; }
}
```

For an object `p` of type `Polar`, the `norm` method is called as follows:

```
p.norm()
```

This can be naively viewed as the call `norm(p)` of a function that takes the object as an argument, like this:

```
  double norm(Polar x) { return x.rho; }
```

Objects thus fulfill a first role of *encapsulation*.

It is possible to declare a field as *static*, and it is then linked to the class rather than to the instances of that class. In other words, it is akin to a global variable.

[1] Once a constructor is explicitly introduced, the default constructor that takes no arguments disappears. We could no longer write `new Polar()` as we did just before.

```
class Polar {
  double rho, theta;
  static double two_pi = 6.283185307179586;
}
```

Similarly, a *method* can be *static*, and it is then akin to a traditional function. Here is an example:

```
  static double normalize(double x) {
    while (x < 0) x += two_pi;
    while (x >= two_pi) x -= two_pi;
    return x;
  }
}
```

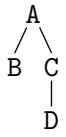
What is not static is called *dynamic*.

The second object-oriented concept is that of *inheritance*: a class `B` can be defined as inheriting from a class `A`, using the syntax:

```
class B extends A { ... }
```

Objects of class `B` then inherit all the fields and methods of class `A`, to which they can add new fields and new methods. The notion of inheritance is accompanied by a notion of *subtyping*: any value of type `B` can be seen as a value of type `A`. In Java, each class inherits from at most one class. This is called *single inheritance*, as opposed to multiple inheritance. The inheritance relation thus forms a tree structure.

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```



The class `A` is called the *superclass* of `B` and `C`. Similarly, `C` is the superclass of `D`. The subtyping relation is naturally transitive. Thus, any value of type `D` can be used as a value of type `A`.

Let's illustrate the usefulness of inheritance with the classic example of graphical elements (rectangles, circles, etc.). We start by introducing a `Graphical` class to represent any graphical element, with a central position and horizontal and vertical dimensions.

```
class Graphical {
  int x, y;          /* center */
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  void draw() { /* does nothing */ }
}
```

Two methods are declared in the `Graphical` class: `move` translates the element, and `draw` draws it. Since we do not yet know the nature of the element, the `draw` method does nothing for now.

To represent a specific element, for example, a rectangle, we inherit from the `Graphical` class.

```
class Rectangle extends Graphical {
```

In particular, `Rectangle` inherits the fields `x`, `y`, `width`, and `height`, and the methods `move` and `draw`. We can write a constructor that takes two corners of the rectangle as arguments:

```
  Rectangle(int x1, int y1, int x2, int y2) {
    x = (x1 + x2) / 2;
    y = (y1 + y2) / 2;
    width = Math.abs(x1 - x2);
    height = Math.abs(y1 - y2);
  }
```

We can directly use any method inherited from `Graphical`, such as `move`:

```
  Rectangle p = new Rectangle(0, 0, 100, 50);
  p.move(10, 5);
```

For drawing, however, we will *override* the `draw` method in the `Rectangle` class.

```
class Rectangle extends Graphical {
  ...
  void draw() { ... draws the rectangle ... }
}
```

The rectangle will then be effectively drawn when we call:

```
  p.draw();
```

We proceed similarly to introduce a `Circle` class for circles.

```
class Circle extends Graphical {
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx;
    y = cy;
    radius = r;
    width = height = 2 * radius;
  }
  void draw() { ... draws the circle ... }
}
```

Here, we added a `radius` field for the radius, to preserve it. An object of the `Circle` class thus has five fields, four of which are inherited from `Graphical`.

Static Type and Dynamic Type. The construct `new C(...)` creates an object of class `C`, and the class of this object cannot be modified afterward. It is called the *dynamic type* of the object. However, the *static type* of an expression, as calculated by the compiler during type checking, can be different from the dynamic type due to the subtyping relationship introduced by inheritance. During the compilation of the program:

```
  Graphical g = new Rectangle(0, 0, 100, 50);
  g.draw(); // draws the rectangle
```

The expression g has the type `Graphical`, but the rectangle is indeed drawn. It is the `draw` method of the `Rectangle` class that is executed. Of course, the variable `g` is initialized just above with `new Rectangle`, and the compiler thus knows the dynamic type of `g` if it wishes. But this is not always possible, as we will now show.

Let's introduce a third type of graphical element, which is simply the union of several graphical elements. We start by introducing a `GList` class for linked lists of `Graphical`.

```
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) {
    this.g = g;
    this.next = next;
  }
}
```

The keyword `this` refers to the object whose constructor or method is being called. It is used here to distinguish the formal parameter `g` from the field `g` of the same name[2]. A group of graphical elements inherits from `Graphical` and contains the list of these elements.

```
class Group extends Graphical {
  GList group;

  Group() { group = null; }

  void add(Graphical g) {
    group = new GList(g, group);
    // + update of x, y, width, height
  }
}
```

It remains to override the `draw` and `move` methods in the `Group` class:

```
  void draw() {
    for (GList l = group; l != null; l = l.next)
      l.g.draw();
  }

  void move(int dx, int dy) {
    x += dx; y += dy;
    for (GList l = group; l != null; l = l.next)
      l.g.move(dx, dy);
  }
```

It is clear that during the type checking of these two methods, the compiler cannot know the dynamic type of `l.g`. The list `l` can indeed contain `Rectangle` as well as `Circle`, arbitrarily mixed. We could even have in `l` objects of subclasses of `Graphical` that have not yet been defined.

[2]This avoids the need to introduce a different name for the formal parameter. Note that the assignment `g = g` would be accepted but would only assign the parameter `g` with its own value.

Abstract Class. Since there is never a need to create an instance of the `Graphical` class, we can make it an *abstract class*. We are then exempted from providing the code for certain methods, such as `draw`.

```
abstract class Graphical {
  int x, y;
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  abstract void draw();
}
```

It is then mandatory to define `draw` in every (non-abstract) subclass of `Graphical`.

Figure 6.1 contains the entirety of our Java program. Chapter 9 will explain how such a program is compiled.

Overloading. In Java, several methods in the same class can have the same name, provided they have different numbers and/or types of arguments. This is called *overloading*. We can define two `draw` methods in the `Rectangle` class:

```
class Rectangle extends Graphical {
  ...
  void draw() { ... }
  void draw(String color) { ... }
}
```

and then write:

```
  r.draw() ... r.draw("red") ...
```

Overloading is resolved during type checking. Everything happens as if we had written two methods with different names, for example:

```
class Rectangle extends Graphical {
  ...
  void draw() { ... }
  void draw_String(String color) { ... }
}
```

and then:

```
  r.draw() ... r.draw_String("red") ...
```

Constructors can also be overloaded. Thus, we can add a second constructor to the `Rectangle` class that takes only three arguments to construct a square:

```
class Rectangle extends Graphical {
  Rectangle(int x1, int y1, int x2, int y2) {
    ...
  }
  Rectangle(int x1, int y1, int w) {
    this(x1, y1, x1 + w, y1 + w); /* constructs a square */
  }
  ...
}
```

```
abstract class Graphical {
  int x, y;
  int width, height;
  void move(int dx, int dy) { x += dx; y += dy; }
  abstract void draw();
}
class Circle extends Graphical {
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx; y = cy; radius = r; width = height = 2 * radius; }
  void draw() { ... /* drawing */ ... }
}
class Rectangle extends Graphical {
  Rectangle(int x1, int y1, int x2, int y2) {
    this.x = (x1+x2)/2; this.y = (y1+y2)/2;
    width = Math.abs(x1-x2); height = Math.abs(y1-y2); }
  void draw() { ... /* drawing */ ... }
}
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) { this.g = g; this.next = next; }
}
class Group extends Graphical {
  private GList group;
  Group() { group = null; }
  void add(Graphical g) {
    group = new GList(g, group);
    ... // update x,y,width,height
  }
  void draw() {
    for (GList l = group; l != null; l = l.next) l.g.draw(); }
  void move(int dx, int dy) {
    super.move(dx, dy); // to update x,y
    for (GList l = group; l != null; l = l.next) l.g.move(dx, dy); }
}
class Main {
  public static void main(String arg[]) {
    Rectangle g1 = new Rectangle(0, 0, 100, 50);
    g1.move(10, 5); g1.draw();
    Graphical g2 = new Circle(10,10,2);
    Group g3 = new Group(); g3.add(g1); g3.add(g2);
    g3.draw(); g3.move(-5,-7); g3.draw();
  }
}
```

Figure 6.1: An example of a Java program.

On the first line of a constructor, the syntax `this(...)` allows calling another constructor of the same class. Instead, we can use the syntax `super(...)` to call a constructor of the superclass. In the absence of either of these constructions, a call to `super()` to the no-argument constructor of the superclass is implicitly inserted.

Overloading Resolution. Overloading is resolved during type checking, *i.e.*, using only static types. Once this resolution is done, we can consider that constructors and methods have distinct names. Thus, the program on the left will become the program on the right, and we can forget all considerations of overloading during the actual compilation.

```
class A {
  A() {...}
  A(int x) {...}

  void m() {...}
  void m(A a) {...}
  void m(A a, A b) {...}
```

```
class A {
  A() {...}
  A_int(int x) {...}

  void m() {...}
  void m_A(A a) {...}
  void m_A_A(A a, A b) {...}
```

Overloading is not without its subtleties. For example, if we introduce:

```
class A {...}
class B extends A {
  void m(A a) {...}
  void m(B b) {...}
}
```

then in the code snippet:

```
{ ... B b = new B(); b.m(b); ... }
```

both methods potentially apply. The method `m(B b)` is called because it is *more specific* from the argument's perspective. In some cases, there can be ambiguity. Here is an example:

```
class A {...}
class B extends A {
  void m(A a, B b) {...}
  void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b, b); ... }
```

On such a program, the Java compiler will fail during type checking with the message:

```
surcharge1.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

The overloading resolution algorithm works as follows. For each method defined:

$$\tau\ \mathtt{m}(\tau_1\ x_1, ..., \tau_n\ x_n)$$

in class `C`, we associate the profile:

$$(\mathtt{C}, \tau_1, \ldots, \tau_n).$$

We *order* the profiles by setting $(\tau_0, \tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \ldots, \tau'_n)$ if and only if τ_i is a subtype of τ'_i for all i. For a method call:

$$e.m(e_1, \ldots, e_n)$$

where e has the static type τ_0 and e_i has the static type τ_i, we consider the set of *minimal* elements in the set of compatible profiles. If this set is empty, no method applies. If this set contains multiple elements, there is ambiguity. Finally, if this set contains a single element, it is the method to call.

Overloading in Other Languages. Overloading is not a concept specific to object-oriented languages. Overloading is found in many languages, particularly for arithmetic operations. For example, the C language overloads the `+` operation, which can be used to add different types of integers, floats, or even perform pointer arithmetic (*i.e.*, add a pointer and an integer). It is the static types of the operands that allow the compiler to differentiate the operations. Similarly, the C language indistinctly uses `>>` for arithmetic and logical right shifts. It is the signed or unsigned nature of the operand that determines the machine operation in the compiled code.
There is no overloading in the OCaml language. Overloading does not mix well with type inference and polymorphism. For example, what would be the type of a function like `fun x -> x+x` if the `+` operation were overloaded on the types `int` and `float`? Continuing to ensure the most general possible type would require a much more complex system than Hindley-Milner. This is why we find the operations `+` and `+.` for the types `int` and `float` in OCaml.
There is also no overloading in Python, but this time for a different reason: the absence of static typing. If Python's `+` operation seems overloaded, it is only because x`+`y is syntactic sugar for the method call x`.__add__(`y`)` and the method `__add__` is defined differently in different Python classes.
In languages like Rust or Haskell, the problem of overloading is approached differently, with *traits* and *type classes* respectively, which are relatively similar concepts. This goes beyond the scope of this course, but one can read with interest the article *How to make ad-hoc polymorphism less ad hoc* by Wadler and Blott [28].

	horizontal extension = adding a case	vertical extension = adding a function
Java	*easy* (one file)	cumbersome (multiple files)
OCaml	cumbersome (multiple files)	*easy* (one file)

Figure 6.2: Duality of functional and object-oriented paradigms.

6.2 Comparison of Functional and Object-Oriented Paradigms

At this point, we can legitimately wonder what differentiates the four classes `Graphical`, `Rectangle`, `Circle`, and `Group` that we have just defined from an OCaml program such as:

```
type graphical = Circle of ... | Rectangle of ... | Group of ...
let move = function Circle _ -> ... | Rectangle _ -> ...
let draw = function Circle _ -> ... | Rectangle _ -> ...
```

A first, almost syntactic, difference concerns the organization of the code. In Java, the code related to each type of graphical object is isolated in a different class, possibly in a different file. Consequently, the code related to an operation (drawing, moving, etc.) is scattered across the different classes. In OCaml, it will be the opposite: the code related to an operation is isolated in a single function, but the different constructors of the `graphical` type are grouped in a single declaration. If we stop there, the difference is not so significant. In particular, Java's dynamic method call and OCaml's pattern matching play very similar roles and are compiled with comparable efficiency.

On the other hand, the difference between Java and OCaml will be more significant if we try to modify our code to add a new kind of graphical object (for example, triangles) or a new operation (for example, scaling). In the case of Java, it is easy to add a new kind of object. This can be done in a new file. It can even be done if the original code is only available in compiled form. With OCaml, it is less easy, because we need to modify the `graphical` type and all the functions already written. In particular, we cannot do this without access to the source code. If, on the other hand, we want to add a new operation, then it is the opposite: doing it with OCaml is easy (we declare a new function and do not need the source) but doing it with Java is difficult (we modify the existing classes and thus need the source). Figure 6.2 summarizes this duality.

6.3 A Few Words on C++

We revisit the example from figure 6.1. We provide a version in C++ in figure 6.3. Beyond just syntax, there are some notable differences with Java. The visibility of fields and methods, for example, is not the same by default. For the fields and methods of `Graphical` to be visible, they must be preceded by `public`. Similarly, `Circle` must publicly inherit from `Graphical` for the fields and methods to remain visible.

```
class Graphical {
public:
  int x, y, width, height;
  virtual void move(int dx, int dy) { x += dx; y += dy; }
  virtual void draw() = 0;
};
class Circle : public Graphical {
public:
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx; y = cy; radius = r; width = height = 2 * radius; }
  void draw() { ... /* drawing */ ... }
};
class Rectangle : public Graphical {
public:
  Rectangle(int x1, int y1, int x2, int y2) {
    this->x = (x1+x2)/2; this->y = (y1+y2)/2;
    width = abs(x1-x2); height = abs(y1-y2); }
  void draw() { ... /* drawing */ ... }
};
class GList {
public:
  Graphical* g;
  GList* next;
  GList(Graphical* g, GList* next) { this->g = g; this->next = next; }
};
class Group : public Graphical {
  GList* group;
public:
  Group() { group = NULL; }
  void add(Graphical* g) {
    group = new GList(g, group); ... /* update x,y,width,height */ }
  void draw() {
    for(GList* l = group; l != NULL; l = l->next) l->g->draw(); }
  void move(int dx, int dy) {
    Graphical::move(dx, dy); // to update x,y
    for(GList* l = group; l != NULL; l = l->next) l->g->move(dx, dy); }
};
int main() {
  Rectangle g1 = Rectangle(0, 0, 100, 50);
  g1.move(10, 5); g1.draw();
  Circle g2 = Circle(10, 10, 2);
  Group g3;
  g3.add(&g1); g3.add(&g2);
  g3.draw(); g3.move(-5,-7); g3.draw();
}
```

Figure 6.3: The program from figure 6.1 in C++.

Additionally, we must explicitly declare that the `move` method can be overridden by adding the qualifier `virtual` before its declaration. Again, the behavior is the opposite of Java, where we declare instead that a method cannot be overridden (with `final`).

We also note that the `add` method of `Group` receives its argument `g` as a pointer. Indeed, the default parameter passing mode in C++ is by value, as we explained in Section 7.2, and an object is no exception. In our case, we do not want the object to be copied. More generally, we have explicitly made a number of pointers visible here that were implicit in the Java code.

Another notable difference with Java is the ability to allocate objects on the stack. We do this, for example, in the `main` method when we write:

```
Rectangle g1(0, 0, 100, 50);
```

The constructor of `Rectangle` is called here for an object allocated on the stack. We can also allocate an object on the heap, with a `new` construction similar to that of Java. Thus, we could have written:

```
Rectangle* g1 = new Rectangle(0, 0, 100, 50);
```

We note that the result of `new` is a pointer to an object. Unlike Java, this pointer is explicit. Whether an object is allocated on the stack or on the heap, its representation in memory is identical.

Finally, let's note one last important difference between Java and C++. To obtain a second variable `v` designating the same object as `g1`, we can write in C++:

```
Rectangle* v = &g1;
```

Thus, `v` is a pointer to the object represented by `g1`. If we had written instead:

```
Rectangle v = g1;
```

as we would in Java, we would have *copied* the object `g1` into a new object `v`, with a completely different effect. In this regard, the semantics of C++ is consistent with the assignment of structures.

Part III

Back End of the Compiler

7 Parameter Passing

In this chapter, we focus on how function calls are executed, particularly on how parameters are passed from the *caller* to the *callee*. We illustrate various strategies in this context, using examples from existing languages such as C, OCaml, Java, C++, and Python.

Let's begin with some vocabulary. In the *declaration* of a function `f`[1]

```
function f(x1, ..., xn) =
  ...
```

the variables `x1`, ..., `xn` are called the *formal parameters* of `f`. In a *call* to this function

```
f(e1, ..., en)
```

the expressions `e1`, ..., `en` are called the *actual parameters* of `f`. If the language supports in-place modifications, an assignment

```
e1 := e2
```

modifies a memory location designated by the expression `e1`, assigning it the value of the expression `e2`. Most often, the expression `e1` is restricted to certain constructs, as assignments like `42 := 17` or `true := false` would be meaningless. The term *lvalue* (short for *left value*) is used to describe expressions that are legal on the left-hand side of an assignment.

7.1 Evaluation Strategy

The evaluation strategy of a language specifies the order in which computations are performed. It can be defined using formal semantics, as we did in Chapter 2, and the compiler must adhere to this strategy. In particular, the evaluation strategy *may* specify when the actual parameters of a call are evaluated, on the one hand, and the order of evaluation of operands and actual parameters, on the other. However, some aspects of evaluation may

[1] Regardless of the language for now.

intentionally remain *unspecified*. This gives the compiler flexibility to perform optimizations by strategically ordering computations.

Among the various evaluation strategies, we notably distinguish between *strict evaluation* and *lazy evaluation*. With strict evaluation, operands and actual parameters are evaluated *before* the operation or call. Languages such as C, C++, Java, OCaml, and Python have adopted strict evaluation. With lazy evaluation, on the other hand, operands and actual parameters are only evaluated when necessary. In particular, they are not evaluated before the call. Languages like Haskell or Clojure have made this choice.

An imperative language systematically adopts strict evaluation to ensure that the sequencing of side effects aligns with the source code. For example, the following OCaml code:

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

prints `42` because both arguments of `f` are evaluated, even though the function `f` does not use either of its two arguments. It would be difficult to understand the effect of the expression `f (id 40) (id 2)` if one first had to determine which of its arguments `f` actually uses.

In imperative languages, an exception is made for the logical connectives `\&\&` and `||`, which evaluate their second operand only when necessary. Specifically, the connective `\&\&` (respectively `||`) does not evaluate its second operand if the first is false (respectively true), even if the second operand has side effects. This allows writing code snippets such as

```
while 0 < !j && v < a.(!j-1)
```

which ensure that `a.(!j-1)` is not evaluated when `!j` is zero.

It is important to understand that non-termination is also a side effect. For example, the following OCaml program:

```
let rec loop () = loop ()
let f x y = x + 1
let v = f 41 (loop ())
```

does not terminate, even though the argument `y` of `f` is not used.

A purely applicative language, on the other hand, can adopt the evaluation strategy of its choice, since an expression will always yield the same value. This property is known as *referential transparency*. In particular, it can choose lazy evaluation. For example, the following Haskell program:

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

terminates (after printing `a`).

Parameter Passing. The evaluation strategy of a language also specifies the *passing mode* of parameters during a call. We notably distinguish between *call by value*, *call by*

reference, *call by name*, and *call by need*. These are sometimes referred to as *passing* by value, by reference, etc.

In call by value, *new* variables representing the formal parameters receive the *values* of the actual parameters[2]. Thus, a program like:

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)
```

prints `41`, because the parameter `x` of `f` is a new variable that receives the value `41`, and it is this variable that is incremented. The variable `v`, on the other hand, remains equal to `41`, hence the result.

In call by reference, however, the formal parameters refer to the *same* lvalues as the actual parameters. Thus, the same program above prints `42` in call by reference, because `x` refers to the same variable as `v`, which is therefore incremented.

In call by name, the actual parameters are *substituted* for the formal parameters textually and are therefore evaluated only if necessary. Thus, the program:

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0))
```

prints `25`, evaluating the expression `1+2` twice and the expression `2+2` twice. However, the expression `1/0` is never evaluated, which is why the program does not fail.

Finally, in call by need, the actual parameters are evaluated only if necessary, but *at most once*. Thus, the same program above still prints `25`, but evaluates the expression `1+2` only once and the expression `2+2` only once.

7.2 Comparison of Java, OCaml, C, C++, and Python

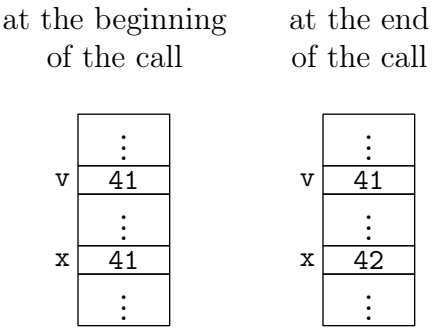
A thorough understanding of a programming language, especially for effective use, requires a clear grasp of its evaluation strategy. In this section, we detail the parameter passing modes of five languages: Java, OCaml, C, C++, and Python.

Java. Java uses a strict evaluation strategy with *call by value*. The evaluation order is specified as left-to-right, meaning parameters are evaluated in the order they are written[3]. A value is either of a primitive type (boolean, character, machine integer, etc.) or a pointer to an object allocated on the heap. If an integer is passed to a function that increments its argument, this increment will not be observed by the caller. This can be understood by visualizing the call by value as illustrated below:

[2]We already discussed call by value in Chapter 2

[3]Interestingly, *The Java Language Specification* states, "It is recommended that code not rely crucially on this specification."

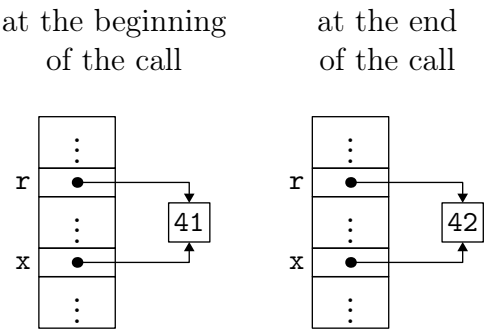
```
void f(int x) {
  x += 1;
}
int main() {
  int v = 41;
  f(v);
  // v is still 41
}
```



Here, the variables `v` and `x` are represented as allocated on the stack, in the stack frames of the `main` and `f` functions, respectively[4]. These variables could also be allocated in registers, which would not change the outcome of this example.

When an *object* is passed as an argument to a function, it is still call by value, but the value is now a pointer, even if it is not explicit in Java. Here is an illustration:

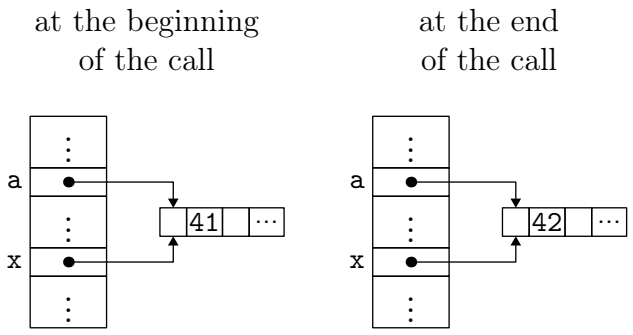
```
class C { int f; }

void incr(C x) {
  x.f += 1;
}
void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f is now 42
}
```



This time, the increment is observed by the caller because, even though a new variable `x` was created, it only contained a pointer to the same object as the one referenced by `r`, and it is in this object that the modification was made.

In Java, an array is an object (almost) like any other, leading to a similar situation:

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] is now 42
}
```



[4]Since the stack grows downward, the variable `x` appears lower than the variable `v`.

In Java, one can *simulate call by name* by replacing arguments with argumentless functions[5]. Thus, the function:

```
int f(int x, int y) {
  if (x == 0) return 42; else return y + y;
}
```

can be rewritten as:

```
int f(Supplier<Integer> x, Supplier<Integer> y) {
  if (x.get() == 0)
    return 42;
  else
    return y.get() + y.get();
}
```

and called like this:

```
int v = f(() -> 0, () -> { throw new Error(); });
```

This code will not fail, because the function `y` is not applied in the case where `x.get()` returns zero. This effectively simulates call by name.

More subtly, one can also *simulate call by need* in Java. This can be done using an object that caches the value returned by the function the first time it is called.

```
class Lazy<T> implements Supplier<T> {
  private T cache = null;
  private Supplier<T> f;

  Lazy(Supplier<T> f) { this.f = f; }

  public T get() {
    if (this.cache == null) {
      this.cache = this.f.get();
      this.f = null; // allows the GC to reclaim f
    }
    return this.cache;
  }
}
```

This is simply memoization applied to a function with only one value. The `Lazy` class can then be used as follows:

```
int w = f(new Lazy<Integer>(() -> 1),
          new Lazy<Integer>(() -> { ...expensive computation... }));
```

OCaml. OCaml uses a strict evaluation strategy with *call by value*. The evaluation order is not specified. A value is either of a primitive type (Boolean, character, machine integer, etc.) or a pointer to a memory block (array, record, non-constant constructor, etc.) generally allocated on the heap. Left values in OCaml are array elements and fields

[5]Here, an argumentless function is simply an object of type `Supplier`, with a `get` method to apply it.

of records declared as mutable. A mutable variable, called a *reference* in OCaml, is simply a value of a predefined record type `ref` defined as:

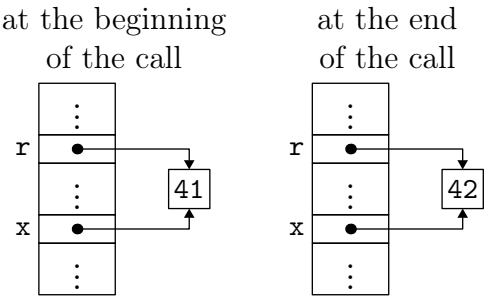
```
type 'a ref = { mutable contents: 'a }
```

Access and assignment operations ! and := are defined as follows:

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

When a reference is passed as an argument to a function, the value passed is a pointer, and it is passed *by value*. Here is an illustration:

```
let incr x =
  x := !x + 1

let main () =
  let r = ref 41 in
  incr r
  (* !r is now 42 *)
```



This situation is entirely analogous to the case of passing an object in Java, as seen earlier. The same applies to passing an array in OCaml, where the value of an array is a pointer to a memory block containing the array elements.

As we did in Java, one can *simulate call by name* in OCaml by replacing arguments with functions. Thus, the function:

```
let f x y =
  if x = 0 then 42 else y + y
```

can be rewritten as:

```
let f x y =
  if x () = 0 then 42 else y () + y ()
```

and called like this:

```
let v = f (fun () -> 0) (fun () -> failwith "oops")
```

One can also *simulate call by need* in OCaml by first introducing a type to represent lazy computations:

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

and a function that evaluates such a computation if it has not already been done:

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

The function `f` is then defined slightly differently to force the evaluation of its arguments:

```
let f x y =
  if force x = 0 then 42 else force y + force y
```

and called as follows:

```
let v = f (ref (Frozen (fun () -> 1)))
          (ref (Frozen (fun () -> ...expensive computation...)))
```

The lazy construct in OCaml does something similar, but in a slightly more sophisticated way.

C. The C language is a relatively low-level imperative language, notably because the notion of pointers and pointer arithmetic is explicit. Conversely, it can be considered a high-level assembly language, which is an excellent definition of C. The C language uses a strict evaluation strategy with *call by value*. The evaluation order is not specified.

Among C's types, there are basic types such as `char`, `int`, `float`, `bool`, etc. In addition to Booleans, any scalar value can be used in a condition, and it is true if and only if it is non-zero. There is also a type τ`*` for pointers to values of type τ. If p is a pointer of type τ`*`, then `*`p denotes the value pointed to by p, of type τ. Conversely, if e is an lvalue of type τ, then `&`e is a pointer to the corresponding memory location, of type τ`*`. Finally, there are records, called *structures*, such as:

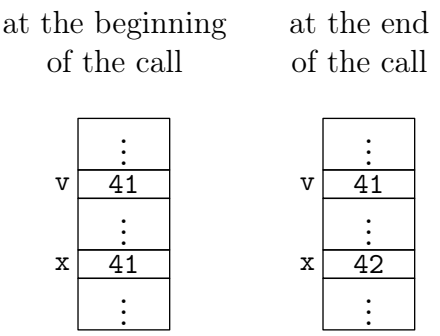
```
struct L { int head; struct L *next; };
```

If `e` has the type `struct L`, `e.head` denotes access to the field.

In C, a left value can take three possible forms: a variable `x`, the dereferencing of a pointer `*e`, or access to a structure field `e.x` if `e` is itself an left value. Furthermore, `t[e]` is syntactic sugar for `*(t+e)`, and `e->x` is syntactic sugar for `(*e).x`. These are, at least syntactically, two other forms of left values.

Passing an integer by value does not differ from Java:

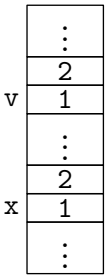
```
void f(int x) {
  x += 1;
}

int main() {
  int v = 41;
  f(v);
  // v is still 41
}
```



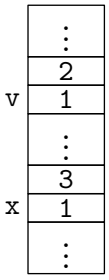
Passing a structure, however, presents a new situation, as a C structure *is a value* and is therefore copied during a call. Here is an illustration:

```
struct S { int a; int b; };

void f(struct S x) {
  x.b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b is still 2
}
```

at the beginning of the call



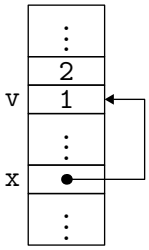
at the end of the call



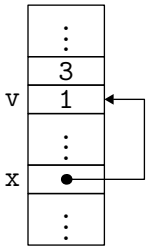
Structures are also copied when returned with `return` and during structure assignments, *i.e.*, assignments of the form `x = y`, where `x` and `y` have the type `struct S`. To avoid the cost of such copies, pointers to structures are most often used, as follows:

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b is now 3
}
```

at the beginning of the call



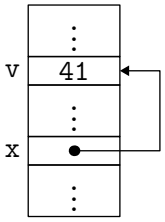
at the end of the call



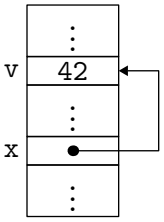
This is a way to *simulate* call by reference, but it remains a call by value of a pointer. The same can be done with an integer:

```
void incr(int *x) {
  *x += 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v is now 42
}
```

at the beginning of the call



at the end of the call



Explicit pointer manipulation can be dangerous. Consider, for example, the program:

```
int* p() {
  int x;
  ...
  return &x;
}
```

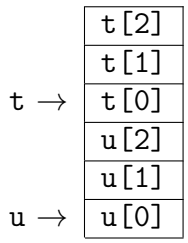
It returns a pointer obtained by taking the address of the local variable `x`, *i.e.*, a pointer that corresponds to a location on the stack that has just disappeared (namely, the stack frame of `p`) and will most likely be reused quickly by another stack frame. This is known as a *dangling reference*.

In C, arrays are not really a type in their own right. If you declare an array of ten integers with

```
int t[10];
```

the notation `t[i]` is just syntactic sugar for `*(t+i)`, where `t` denotes a pointer to the start of an area containing 10 integers, and `+` denotes a *pointer arithmetic* operation (which consists of adding `4i` to `t` for a 32-bit integer array). The first element of the array is therefore `t[0]`, *i.e.*, `*t`. When passing an array as a parameter, only the pointer is passed, always by value. You cannot assign arrays, only pointers. Thus, you cannot write:

```
void p() {
  int t[3];
  int u[3];
  t = u; // refused
}
```



because `t` and `u` are arrays allocated on the stack, and array assignment is not allowed.

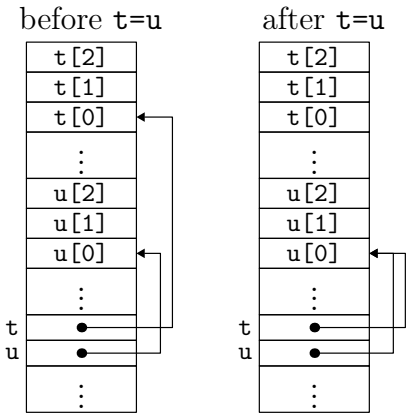
However, you can write:

```
void q(int t[3], int u[3]) {
  t = u;
}
```

because it is exactly the same as:

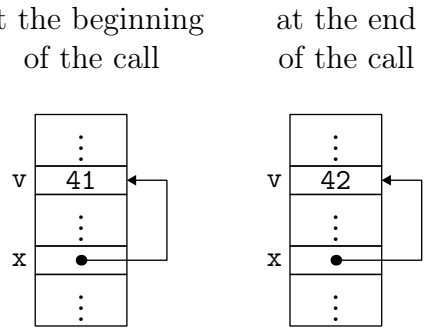
```
void q(int *t, int *u) {
  t = u;
}
```

and pointer assignment is allowed.



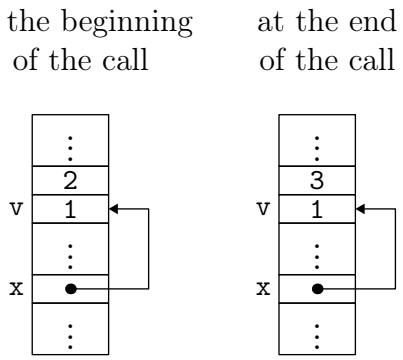
C++. In C++, you will find (among others) the types and constructs of C, with a strict evaluation strategy. The default passing mode is *by value*. However, there is also a *call by reference*, indicated by the symbol `&` at the formal argument level. Thus, you can write:

```
void f(int &x) {
  x += 1;
}

int main() {
  int v = 41;
  f(v);
  // v is now 42
}
```



In particular, it is the compiler that takes the address of `v` at the time of the call, on the one hand, and dereferences the address `x` in the function `f`, on the other. Similarly, you can pass a structure by reference, which is an alternative to the explicit use of pointers as in C:

```
struct S { int a; int b; };

void f(struct S &x) {
  x.b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b is now 3
}
```



You can also pass a pointer by reference. A relevant example is inserting an element into a tree:

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if      (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

This elegantly handles the case of insertion into an empty tree.

Exercise 42. Using *Compiler Explorer* (`godbolt.org`), observe and explain the assembly code produced by `gcc -Og` for these two C++ functions:

```
int f(int x) { return -x; }
int g(int &x) { return -x; }
```

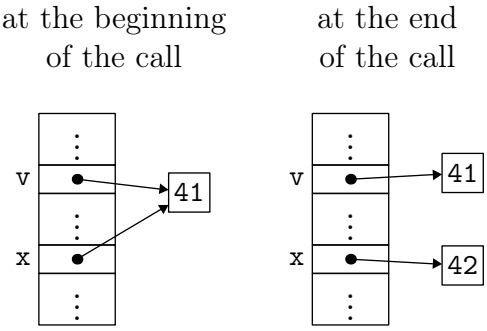
Then add a third function containing a call to `g(v)` on a local variable `v` and observe:

- where the variable `v` is allocated;
- how the call is compiled.

Solution □

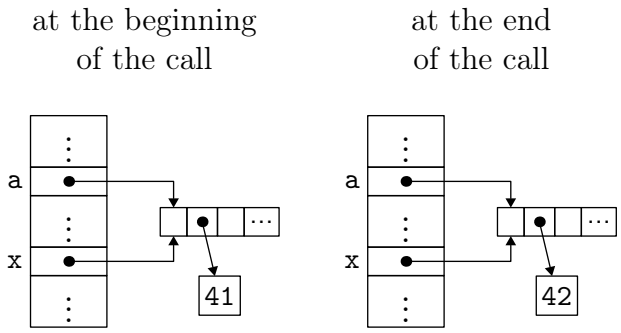
Python. The Python language uses a strict evaluation strategy with *call by value*. The evaluation order is specified as left-to-right (but right-to-left for assignments). A value is a pointer to an object allocated on the heap. An integer is an immutable object. When passing an integer, it is always a *call by value* of a value that is an (implicit) pointer to an object.

```
def f(x):
    x += 1

v = 41
f(v)
print(v) # prints 41
```

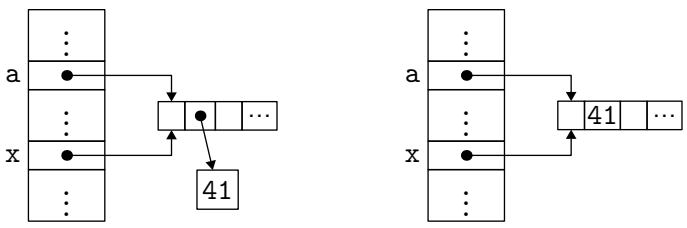


An array is a mutable object. The modification of an array element by a function can be visualized as follows:

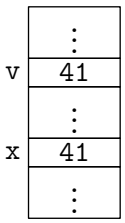
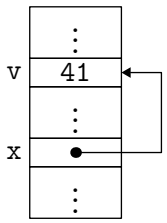
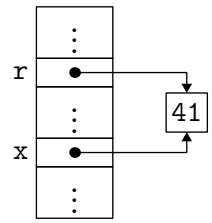
```
def incr(x):
    x[1] += 1

a = [0] * 17
a[1] = 41
incr(a)
# a[1] is now 42
```



Since integers are *immutable* objects, we can abstract away their representation as objects. In other words, we can simplify our representation of the situation above by identifying the following two representations as equivalent:



Summary. Let's summarize the different situations we have encountered with these five languages regarding the passing of an integer to a function:

			
C	integer by value	pointer by value	pointer by value
C++	integer by value	pointer by value integer by reference	pointer by value or by reference
OCaml	integer by value	—	pointer by value (e.g., type `ref`)
Java	integer by value	—	pointer by value (object)
Python	—	—	pointer by value (object)

In particular, we note that the situation illustrated in the middle column, corresponding to passing a pointer to the location of a variable, has no equivalent in OCaml, Java, and Python.

It is striking to note that the execution models of Java, OCaml, and Python are *very similar*—only passing by value of atomic values (a memory word)—even though their surface languages are very different.

Bibliographical Notes. The book *The C Programming Language* by Brian Kernighan and Dennis Ritchie [14, 13] remains a reference for learning this language.

7.3 Compilation of a Micro C++

To fully understand parameter passing from a compilation perspective, let's write a compiler for a small fragment of the C++ language. The interest of this language here is that it supports both pass-by-value and pass-by-reference mechanisms. The abstract syntax of this micro C++ is given in Figure 7.1.

Variables are denoted by x and functions by f. Expressions E are limited to integer expressions of the single type `int`. Boolean conditions C are limited to comparisons of integer expressions and the three connectives `&&` (and), `||` (or), and `!` (not). A program consists of a sequence of functions, ending with a `main` function, which serves as the entry point. A variable is introduced either as a function parameter or as a local variable within a block. Variables are mutable. Local variables are not initialized.

The evaluation strategy is strict, except for the `&&` and `||` constructs. Parameters are passed by value when declared with the syntax `int` x and by reference when declared with the syntax `int &`x. Functions do not return a value, but pass-by-reference can be used to return a result. We assume the existence of a predefined `printf` function to display an integer. Figure 7.2 contains a micro C++ program that computes the Fibonacci sequence.

The *scope* defines the portions of the program where a variable is visible. Here, if the body of a function f mentions a variable x, then:

E	::=	n	arithmetic expression
	|	x	
	|	E `+` E | E `-` E	
	|	E `*` E | E `/` E	
	|	`-` E	
C	::=	E `==` E | E `!=` E	Boolean expression
	|	E `<` E | E `<=` E	
	|	E `>` E | E `>=` E	
	|	C `&&` C | C `||` C	
	|	`!` C	
S	::=	x `=` E`;`	statement
	|	`if (` C `)` S	
	|	`if (` C `)` S `else` S	
	|	`while (` C `)` S	
	|	f`(`E`,...,`E`);`	
	|	`printf("%d\n",` E`);`	
	|	`int` x`,...,`x`;`	
	|	B	
B	::=	`{` S `...` S `}`	block
F	::=	`void` f`(`X`,...,`X`)` B	function
X	::=	`int` x | `int &`x	formal parameter
P	::=	F `...` F	program
		`int main()` B	

Figure 7.1: Abstract syntax of micro C++.

```
void fib(int n, int &r) {
  if (n <= 1)
    r = n;
  else {
    int tmp;
    fib(n - 2, tmp);
    fib(n - 1, r);
    r = r + tmp;
  }
}

int main() {
  int f;
  fib(10, f);
  printf("%d\n", f);
}
```

Figure 7.2: A micro C++ program.

- either x is a parameter of f;
- or x is declared earlier in an enclosing block (including the current block).

Additionally, a variable can *shadow* another variable. Thus, in the following code snippet, there are two distinct variables named `n`, with the second temporarily shadowing the first:

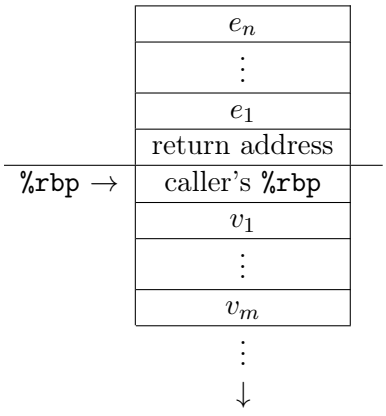
```
  int n; n = 34; printf("%d\n", n); // prints 34
  if (n > 0) {
    int n; n = 89; printf("%d\n", n); // prints 89
  }
  printf("%d\n", n); // prints 34
```

Here, the scope depends solely on the source text. This is referred to as *lexical scoping*[6]. It can be resolved either before or during type checking. The abstract syntax preserves a trace of this analysis by uniquely identifying each variable. In other words, the abstract syntax reflects the above program exactly as if we had written the following program instead:

```
  int n1; n1 = 34; printf("%d\n", n1);
  if (n1 > 0) { int n2; n2 = 89; printf("%d\n", n2); }
  printf("%d\n", n1);
```

In practice, distinguishing between different variables can be done using unique names or, better yet, by identifying each variable with a pointer to a metadata block: name, type, declaration position in the source, mutability, passing mode, etc.

[6]Some languages, such as `bash`, use *dynamic scoping*, where the scope depends on the program's execution.



Let us now turn to the compilation of micro C++. We adopt a simple compilation scheme here where all variables are stored on the stack. (Chapter 10 will explain in detail how to allocate processor registers to variables instead.) Parameters are located at the top of the *stack frame* (see Chapter 1), placed there by the caller, with local variables lower down in the section allocated by the callee. The stack frame corresponding to a call $f(e_1, \ldots, e_n)$ is illustrated on the right. The upper part is constructed by the caller and contains the values of the arguments $e_1, \ldots, e_n$. This is followed by the return address, placed by the `call` instruction. Finally, the lower part is constructed by the callee and contains the saved `%rbp` register and the local variables $v_1, \ldots, v_m$ of f. Recall that positioning the `%rbp` register in this way allows us to easily locate any variable with a fixed offset (+16, +24, etc., for parameters and −8, −16, etc., for local variables). Indeed, the stack pointer will move if we store intermediate calculations or if we are preparing a function call.

Let us now explain how to compile the various constructs of micro C++ into x86-64 assembly. We assume that the relative position x_o to `%rbp` where a variable x is located in its stack frame is known. For now, we assume that all arguments are passed by value. Let us start with expressions, whose values we will compute as 64-bit signed integers. (In a C++ compiler, the choice for the `int` type would typically be 32 bits, but the standard does not prevent us from choosing 64 bits.) The compilation of an arithmetic expression E is denoted by *Compile*(E). This is a piece of assembly code whose execution must place the value of E in the `%rdi` register (an arbitrary choice). The case of a constant is straightforward:

$$\mathit{Compile}(n) = \texttt{movq \$}n\texttt{, \%rdi}$$

For a variable x, we use indirect addressing with the known relative position x_o:

$$\mathit{Compile}(x) = \texttt{movq } x_o\texttt{(\%rbp), \%rdi}$$

For now, we are limiting ourselves to pass-by-value.

The remaining arithmetic operations are handled next. We will not attempt to perform register allocation here. Instead, we use the stack to store intermediate results. For example, addition can be compiled as follows:

$$\begin{aligned}\mathit{Compile}(E_1 + E_2) = \;& \mathit{Compile}(E_1)\\ & \texttt{pushq \%rdi}\\ & \mathit{Compile}(E_2)\\ & \texttt{popq \%rsi}\\ & \texttt{addq \%rsi, \%rdi}\end{aligned}$$

Of course, this is extremely naive. The code for `1+2` will require five instructions and use the stack, even though we have 16 registers available. However, this is not the focus here.

Boolean expressions are compiled similarly, with a piece of assembly code *Compile*(C) that places the value 0 (false) or 1 (true) of the expression C in the `%rdi` register. Thus,

the equality of two expressions can be compiled as follows:

$$\begin{aligned} \mathit{Compile}(E_1\texttt{==}E_2) = \; & \mathit{Compile}(E_1) \\ & \texttt{pushq \%rdi} \\ & \mathit{Compile}(E_2) \\ & \texttt{popq \%rsi} \\ & \texttt{cmpq \%rdi, \%rsi} \\ & \texttt{sete \%dil} \\ & \texttt{movzbq \%dil, \%rdi} \end{aligned}$$

The same approach is used for other comparisons.

Note that the operators `&&` and `||` must be evaluated lazily, i.e., e_2 is not evaluated in e_1 `&&` e_2 (or e_1 `||` e_2) if e_1 evaluates to `false` (or `true`, respectively). For example, the `&&` operator can be compiled as follows:

$$\begin{aligned} \mathit{Compile}(C_1 \texttt{ \&\& } C_2) = \; & \mathit{Compile}(C_1) \\ & \texttt{testq \%rdi, \%rdi} \\ & \texttt{jz } L \\ & \mathit{Compile}(C_2) \\ & L\texttt{:} \end{aligned}$$

where L denotes a fresh label.

Now, let us address the compilation of statements. A statement S is compiled into a piece of assembly code $\mathit{Compile}(S)$ that realizes the effect of the statement S. Some statements, such as conditionals, `while` loops, or blocks of statements, do not pose any particular difficulty and are not detailed here. For an assignment, the value of the variable is written to the stack:

$$\begin{aligned} \mathit{Compile}(x \texttt{ = } E\texttt{;}) = \; & \mathit{Compile}(E) \\ & \texttt{movq \%rdi, } x_o\texttt{(\%rbp)} \end{aligned}$$

For a function call f, the parameter values must be pushed onto the stack, the code at label f is called, and then the parameters are popped from the stack:

$$\begin{aligned} \mathit{Compile}(f(E_1, \ldots, E_n)) = \; & \mathit{Compile}(E_n) \texttt{ pushq \%rdi} \\ & \vdots \\ & \mathit{Compile}(E_1) \texttt{ pushq \%rdi} \\ & \texttt{call } f \\ & \texttt{addq \$8}n\texttt{, \%rsp} \end{aligned}$$

Note that the parameters are evaluated from right to left (RTL, for *Right To Left*), so that their values are pushed in the order $E_n, \ldots, E_1$, consistent with the structure of the stack frame described earlier. In the C++ standard, the order of parameter evaluation is unspecified, meaning it is left to the compiler's discretion.

Finally, we need to explain how to compile the *declaration* of a function. Each function

is compiled independently as follows:

$$\begin{array}{rl} \mathit{Compile}(f(x_1,\ldots,x_n)\ldots B) = & f\texttt{:} \\ & \texttt{pushq \%rbp} \\ & \texttt{movq \%rsp, \%rbp} \\ & \texttt{subq \$}8m\texttt{, \%rsp} \\ & \mathit{Compile}(B) \\ & \texttt{movq \%rbp, \%rsp} \\ & \texttt{popq \%rbp} \\ & \texttt{ret} \end{array}$$

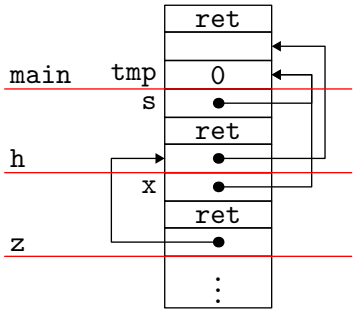
Here, the integer m denotes the number of slots required for the local variables of the function f. The `main` function is compiled like the others, except that the `%rax` register is set to zero just before exiting, as the return value of `main` will be the program's exit code.

Pass-by-Reference. So far, we have assumed that all parameters are passed *by value*. However, the `&` qualifier for a formal parameter specifies a *pass-by-reference*, and the compilation must be adjusted accordingly. In the case of pass-by-reference, the actual parameter must be an left value. In micro C++, this is limited to a variable[7]. When calling a function $f(e_1,\ldots,e_n)$, if parameter i is passed by reference, we must ensure that e_i is indeed a variable and compile this argument differently during the call to pass the *address* of this variable rather than its value. An elegant way to proceed is to add a "compute left value" construct to the expression syntax. Let us denote it as $\&x$, by analogy with the C++ construct, which is not part of our fragment. We assume that typing has introduced this new construct in function calls for each parameter passed by reference. Let us first explain how to compile this new construct:

$$\begin{array}{rll} \mathit{Compile}(\&x) = & \texttt{leaq } x_0\texttt{(\%rbp), \%rdi} & \\ & \texttt{movq (\%rdi), \%rdi} & \text{if } x \text{ is passed by reference} \end{array}$$

The first instruction computes the address where the variable x is stored in `%rdi`. The second instruction is added only when the variable x itself is passed by reference. Here is an example of such a situation:

```
void z(int &x) { x = 0; }
void h(int &s) { z(s); while (s < 100) s = 2*s+1; }
int main() { int tmp; h(tmp); printf("%d\n", tmp); }
```



In the call `z(s)`, the variable `s` is a variable passed to `h` by reference, which is in turn passed by reference to the function `z`. The goal is to ensure that the variable `tmp` in `main` is ultimately set to zero by the function `z`. We have drawn the stack at the moment of the call to `z`.

We must now modify how we compile access to a variable x in the compilation of expressions:

$$\begin{array}{rll} \mathit{Compile}(x) = & \texttt{movq } x_0\texttt{(\%rbp), \%rdi} & \\ & \texttt{movq (\%rdi), \%rdi} & \text{if } x \text{ is passed by reference} \end{array}$$

[7] In a more complete language, this could be an array cell, a structure field, etc.

Similarly, we must modify how we compile an assignment in the compilation of statements:

$$
\begin{array}{lll}
\mathit{Compile}(x \texttt{ = } E\texttt{;}) = & \mathit{Compile}(E) & \\
& \texttt{movq } x_o\texttt{(\%rdi), \%rsi} & \text{if } x \text{ is passed by reference} \\
& \texttt{leaq } x_o\texttt{(\%rdi), \%rsi} & \text{otherwise} \\
& \texttt{movq \%rdi, (\%rsi)} &
\end{array}
$$

Note that there is a single memory access (write) when the variable is not passed by reference (it is local or passed by value), but two memory accesses (one read and one write) when the variable is passed by reference. On the other hand, there is nothing to modify in the function call, thanks to the new address computation construct introduced in the expressions. Similarly, there is nothing to modify in the compilation of the function declaration.

Exercise 43. Consider the following C++ function:

```
void swap(int &x, int &y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```

1. Draw its stack frame.
2. Provide the result of its compilation.

Solution □

8 Compilation of Functional Languages

In this chapter, we focus on various aspects of compilation related to functional languages. Today, many ideas originating from functional languages have found their way into languages with seemingly very different paradigms. For example, first-class functions are now found in Java and C++. Throughout this chapter, we use fragments of the OCaml language for illustration purposes, but it is important to keep in mind that the concepts are not tied to any particular language.

8.1 Functions as First-Class Values

What characterizes functional programming languages is the ability to manipulate functions as first-class values, meaning exactly like values of any other type, such as integers. Thus, one can receive a function as an argument, return it as a result, store it in a list or an array, propagate it through an exception, etc.

Consider the simple OCaml fragment given in figure 8.1, in which expressions contain anonymous functions introduced by `fun`, local declarations (possibly recursive) introduced by `let`, and conditionals. A program is a sequence of declarations, possibly recursive[1]. This OCaml fragment is very close to the Mini-ML language we considered in chapters 2 and 5.

In this fragment, one can write programs where functions are arbitrarily nested, such as:

```
let sum n =
  let f x = x * x in
  let rec loop i = if i = n then 0 else f i + loop (i+1) in
  loop 0
```

This is not yet about first-class functions. Moreover, nesting functions in this way is not a trait exclusive to functional languages and has existed for a long time in languages like Algol, Pascal, or Ada[2]. In this example, it is important to note that local functions can

[1] We do not consider mutually recursive functions here, but that would not add any real difficulty.

[2] The `gcc` compiler offers a non-standard extension where functions can be nested.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x \rightarrow e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

Figure 8.1: A functional core of OCaml.

refer to variables introduced earlier, in outer scopes. Thus, the function `loop` refers to `n`, the argument of the function `sum`, and to the function `f` introduced just before.

It is also possible to take functions as arguments, as in this example:

```
let square f x =
  f (f x)
```

and to return them, as in this other example:

```
let f x =
  if x < 0 then fun y -> y - x else fun y -> y + x
```

In the latter case, the value returned by `f` is a function that uses `x`, but the stack frame of `f` has just disappeared. Therefore, we need to compile such a function in a way that the value of `x` survives this call.

The solution is to use a *closure*. This is a data structure allocated on the heap (to survive function calls) containing, on the one hand, a *pointer to the code* of the function to be called and, on the other hand, the values of the variables that might be used by this code. This second part of the closure is called the *environment*. For a closure representing `fun` $x \rightarrow e$, the variables whose values need to be stored in the environment are exactly the *free variables* of `fun` $x \rightarrow e$ (see definition 1 on page 25).

Let's consider the example of this recursive function `pow`, which calculates $\texttt{x}^{\texttt{i}}$ for a floating-point number `x` and an integer `i`.

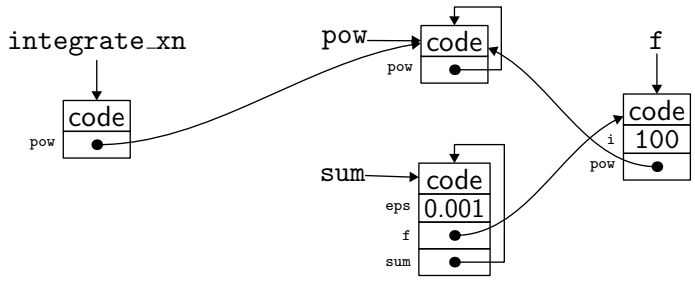
```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x
```

There are two `fun` constructs, which correspond to two closures. In the first closure, `fun i ->`, the environment is reduced to {`pow`}, since the variables `i` and `x` are bound. In the second closure, `fun x ->`, the environment is {`i`, `pow`}. Let us consider a second function, `integrate_xn`, which uses the `pow` function to compute an approximation of $\int_0^1 x^n dx$.

```
let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

Again, there are two closures here. For the first one, the environment is {pow}, and for the second one, the environment is {eps, f, sum}.

A closure can be materialized by a single block on the heap[3], containing the address of the code in its first field and the values of the environment in the subsequent fields. With this representation, the various closures involved during the execution of a call to `integrate_xn 100` look like this:



The two functions `integrate\_xn` and `pow` are closures. Once `integrate\_xn` is called, the application of `pow` to `100` returns a function, and thus a closure, stored in the variable `f`. Finally, the function `sum` is a fourth closure. It is particularly noteworthy that the closures of the functions `pow` and `sum` contain their own values in their environments because they are recursive functions. Thus, the execution of the code of a closure retrieves the value of any variable from the environment, without making any special case for a recursive function. It is during the construction of the closure that care must be taken to handle recursion.

Let us now explain how to mechanize the construction of closures and how to use them. A relatively simple way to compile closures is to proceed in two steps. First, we search the code for all constructs of the form `fun` $x \to e$ and replace them with an explicit closure construction operation:

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

where the y_i are the free variables of `fun` $x \to e$ and f is the name given to a global function declaration of the form:

$$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

where e' is obtained from e by recursively removing the `fun` constructs. This process of making closures explicit is called *closure conversion*.

For our example, the closure conversion yields the following program:

[3]Other solutions are possible, such as an environment allocated in a second block or even as a linked list. However, the solution using a single block is more efficient in practice, as it places less demand on the GC

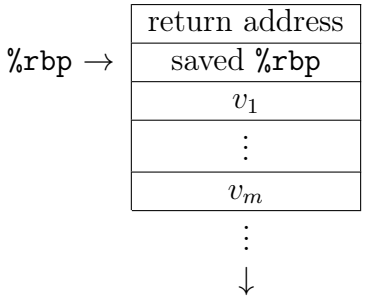
$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{clos}\ f\ [x, \ldots, x] \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
& | & \texttt{letfun}\ f\ [x, \ldots, x]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

Figure 8.2: Explicit closures.

```
letfun fun2 [i,pow] x =
  if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i =
  clos fun2 [i,pow]
let rec pow =
  clos fun1 [pow]
letfun fun3 [eps,f,sum] x =
  if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum = clos fun3 [eps,f,sum] in
  sum 0. *. eps
let integrate_xn =
  clos fun4 [pow]
```

We note that we still have the two global declarations of `pow` and `integrate\_xn`, whose values are closures. The four functions `fun1`, `fun2`, `fun3`, and `fun4` introduced by this translation can be placed arbitrarily in the code, as they are independent of each other. They are also not recursive. Indeed, a closure has in its environment all the values it needs. Thus, the function `fun2` finds the value of `pow` in its environment to perform a recursive call. The target language of this transformation is given in figure 8.2.

In a second step, we compile the obtained code. Let's adopt the compilation scheme where each function introduced by `letfun` receives its single argument in `%rdi` and its closure in `%rsi`. There are no other arguments, and in particular, no arguments passed on the stack. The stack is, however, used to allocate local variables introduced by `let`. We thus have a stack frame of the form:



where $v_1, \ldots, v_m$ are the local variables. Let's detail the compilation of the various constructs. To compile the construct

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

we proceed as follows. We allocate a block of size $n + 1$ on the heap using a function of type `malloc`. We store the address of f in the first field. This address is known, as it is the address of the function f in the compiled program. We store the values of the variables $y_1, \ldots, y_n$ in fields 1 to n of the block. We will explain a bit later how access to the value of a variable is done. We return the address of the block as the value of the expression. We rely on a garbage collector (GC) to free this block when possible. It is indeed difficult, and in general impossible, to decide at what moment a closure can be freed.

Let us now explain how to compile a call, that is, an expression of the form $e_1\ e_2$. We compile e_1 into the register `%rsi`. Since it is a function (as guaranteed by static typing), we know that its value is the address of a closure. We compile e_2 into the register `%rdi`. Finally, we call the function whose address is contained in the first field of the closure, using `call *(%rsi)`. This is therefore a jump to a *computed address*.

Let us now explain how to access the value of a variable x. This is more subtle than it appears, as we must distinguish four cases. If it is a global variable, introduced by a `let` at the global level, its value is located, for example, in the data segment. If it is a local variable, introduced by a `let-in`, its value is on the stack, and we access it via n(`%rbp`) for some n. If it is a variable contained in the closure, we access it via n(`%rsi`) for some n. Finally, if it is the argument of a `letfun` function, its value is in `%rdi`. This case distinction can already be made in the abstract syntax tree after closure conversion.

Finally, we compile a function declaration `letfun` $f\ [y_1, \ldots, y_n]\ x = e$ in the usual way. We allocate the stack frame, in which we save `%rbp` before setting it. We evaluate the expression e into `%rax`. Then we restore `%rbp` and deallocate the stack frame before executing `ret`.

The compilation of the other constructs, namely a constant, a local declaration, and a conditional, is quite standard and independent of the rest.

Optimizations. It is unnecessarily expensive to create intermediate closures in a call where all arguments are provided. A "traditional" call can be made, where all arguments are passed at once. On the other hand, a partial application of the same function must produce a closure. For example, the OCaml compiler performs this optimization. On code that does not use functions as first-class values, the same efficiency as with a non-functional language is thus achieved.

Another optimization is possible when it is certain that a closure will not outlive the function in which it is created. It can then be allocated on the stack rather than on the heap. This is the case, for example, with the closure for `f` in:

```
let integrate_xn n =
  let f = ... in
```

However, to ensure that this optimization is possible, a non-trivial static analysis, known as escape analysis, must be performed.

Closures in Other Languages. Closures are also found in languages such as Java (since 2014 and Java 8) or C++ (since 2011 and C++11). In these languages, anonymous functions are called *lambdas*, echoing the notion of λ-abstraction.

In Java, a function is an object like any other, with an `apply` method. For example, one can write:

```
LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
  ... f.apply(x) ...
}
```

where `Function` is a predefined interface for such an object. An anonymous function is introduced with `->`:

```
map(l, x -> { System.out.print(x); return x+y; })
```

The compiler constructs a closure object, which captures `y` here, with an `apply` method. The variables captured in a closure cannot be modified.

In C++, an anonymous function is introduced with `[]`:

```
for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

The variables captured in the closure are specified, here `y`. By default, variables are captured by value, but a capture by reference can be specified (here for `s`):

```
for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

Variables captured by value cannot be modified.

If a language does not provide closures, they can be manually constructed once the principle is understood. For example, before Java 8, it was entirely possible to occasionally construct a class representing a closure, with environment values in fields and an `apply` method for the function code. Constructing such an object is less convenient than using the syntax provided by Java 8, but the result is identical. Similarly, the C language does not provide closures, but it is entirely possible to introduce, on a case-by-case basis, a structure containing a function pointer and an environment, accompanied by an `apply` function.

The Python language also includes a notion of closure. Thus, a function can return a locally defined function:

```
def f(x):
  def g():
      nonlocal x
      x += 1
      return x
  return g
```

or an anonymous function (introduced by the keyword `lambda`):

```
def f(x):
  return lambda y: x+y
```

In both cases, the returned function indeed has access to the value of `x`, as expected. However, it is not the *value* of `x` that is captured in the closure, but rather the *mutable* variable space that contains `x`. It is therefore still possible to modify the value of `x` afterward, thereby changing the behavior of the returned function. Thus, the following code:

```
def f(x):
  def incr():
      nonlocal x
      x += 1
  return (incr, lambda y: x+y)
```

returns two functions, where the first allows modifying the value of the variable `x` used by the second. Such a "by-reference" capture can be surprising in a language that only has pass-by-value.

8.2 Tail Call Optimization

Functional languages encourage a recursive programming style. When the language is purely applicative, recursion is even the only way to write loops. Consequently, special care is taken in the compilers of functional languages to ensure that recursion is as efficient as using a loop, and in particular that it does not cause stack overflow, as much as possible. To achieve this, an optimization of certain function calls, known as *tail calls*, is performed by the compiler. Today, such optimization is done in a majority of compilers, regardless of the nature of the language considered.

Definition 22. A call to a function f that appears in the body of a function g is said to be a *tail call* if it is the last thing that g computes before returning its result.

By extension, a function is said to be *tail recursive* if it is a recursive function where all recursive calls are tail calls. □

Here is an example of a tail call to a function `f` within a function `g`:

```
let g x =
  let y = x * x in f y
```

The call `f y` is indeed the last thing that `g` computes. Once the result of `f y` is obtained, it is directly returned as the result of `g`.

In a recursive function, there can be tail recursive calls and others that are not, as in the famous McCarthy's function 91:

```
let rec f91 n =
  if n <= 100 then f91 (f91 (n + 11)) else n - 10
```

The advantage of a tail call from a compilation perspective is that the stack frame of the function g containing the call can be destroyed *before* making the call to f, since it will no longer be needed afterward. Even better, it can be reused for the tail call that

needs to be made. In particular, the return address it contains is the correct one. In other words, a jump with `jump` can be made instead of a call with `call`.

Consider, for example, the following program that calculates the factorial of `n` multiplied by `acc`:

```
let rec fact acc n =
  if n <= 1 then acc else fact (acc * n) (n - 1)
```

A standard compilation produces the assembly program on the left, whereas optimizing the tail call results in the program on the right:

```
fact:
        cmpq    $1, %rsi
        jle     L0
        imulq   %rsi, %rdi
        decq    %rsi
        call    fact
        ret
L0:     movq    %rdi, %rax
        ret
```

```
fact:
        cmpq    $1, %rsi
        jle     L0
        imulq   %rsi, %rdi
        decq    %rsi
        jmp     fact

L0:     movq    %rdi, %rax
        ret
```

The resulting program is a *loop*. The code is indeed identical to what would have been produced by compiling a C program such as:

```
while (n > 1) {
  acc = acc * n;
  n   = n - 1;
}
```

This is true even if the language considered does not necessarily have imperative features. We could very well be compiling a purely applicative language.

The program obtained with this optimization is more efficient. On one hand, there is less memory access, since `call` and `ret` instructions that manipulate the stack are no longer used. But more importantly, the stack space used becomes constant. In particular, this avoids any stack overflow that would be due to a large number of nested calls.

It is important to note that the concept of a tail call has nothing to do with functional languages. Its compilation can be optimized in all languages. For example, `gcc` performs this optimization if given the command-line option `-foptimize-sibling-calls` (which is included in the `-O2` option). It is also important to remember that the concept of a tail call is not linked to recursion, even though it is most often a recursive function that will cause a stack overflow and thus for which such optimization is desired.

Application. Suppose we want to write an OCaml function to calculate the height of a tree, for a binary tree type defined as follows:

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

It is natural to write code in the following form:

```
let rec height = function
| Empty          -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

However, this will cause a stack overflow for a tree of great height. Indeed, the two recursive calls to `height` are not tail calls, as the maximum must still be calculated and one must be added once these two calls are completed.

To avoid stack overflow, let's try to write the `height` function using only tail calls. Instead of calculating the height h of the tree, let's calculate $k(h)$ for an arbitrary function k, called a *continuation*. The `height` function will therefore have the following type:

```
val height: 'a tree -> (int -> 'b) -> 'b
```

This is called *continuation-passing style* (CPS). The desired program can be derived using the identity continuation, *i.e.*, `height t (fun h -> h)`. The code then takes the following form:

```
let rec height t k = match t with
| Empty ->
    k 0
| Node (l, _, r) ->
    height l (fun hl ->
    height r (fun hr ->
    k (1 + max hl hr)))
```

We observe that both calls to `height` and both calls to `k` are *tail calls*. The calculation of `height` is therefore done in constant stack space. We have replaced the space on the stack with space *on the heap*, occupied by closures. The first closure captures `r` and `k`, and the second captures `hl` and `k`.

Of course, there are other ad hoc solutions to calculate the height of a tree without causing a stack overflow, such as a breadth-first traversal. Similarly, there are other solutions if the tree type is more complex: mutable trees, height stored in the node, parent pointers, etc. However, the CPS-based solution has the advantage of being mechanical. More details on this example can be found in the article *Mesurer la hauteur d'un arbre* [7].

Exercise 44. What can be done if the language optimizes tail calls but does not offer anonymous functions (for example, the C language)? Solution □

Exercise 45. In the *quicksort* (Hoare, 1961), the segment to be sorted is partitioned into three parts according to a pivot value (elements smaller than the pivot, equal elements, and larger elements), and then the two end portions are recursively sorted.

```
void quicksort(int a[], int l, int r) {
  if (r - l <= 1) return;
  ... // partition a[l..r[ into three parts a[l..lo..hi..r[
  quicksort(a, l, lo);
  quicksort(a, hi, r);
}
```

There are two recursive calls, only the second of which is a tail call.

1. Assuming that tail calls are optimized, explain how to guarantee logarithmic stack space (without making or modifying any assumptions about the partitioning).

2. Now assuming that tail calls are not optimized (e.g., in Java, Python, etc.), explain how to modify the `quicksort` function above to continue guaranteeing logarithmic stack space.

Solution □

Tail Calls and Exceptions. In OCaml, a call located in an exception handler is not a tail call. Thus, the following code can cause a stack overflow:

```
let rec loop n = try loop (f n) with Exit -> ()
```

However, a tail call can be recovered by using a different construction that combines pattern matching and exception handling as follows:

```
let rec loop n =
  match f n with
  | m -> loop m
  | exception Exit -> ()
```

This pattern can be advantageously used, for example, to read the lines of a file, processing them one by one, and stopping when the `End_of_file` exception is raised.

8.3 Pattern Matching

In functional languages, there is generally a construct called *pattern matching*, used in function definitions, such as

$$\texttt{function}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n,$$

generalized conditionals, such as

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n,$$

and exception handlers, such as

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n.$$

The compiler transforms these high-level constructs into sequences of *elementary tests* (constructor tests and comparisons of constant values) and accesses to fields of structured values. We will explain this compilation process. In what follows, we consider the construct

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n,$$

to which it is easy to reduce using a `let`. Let's start by defining what the p_i above represent.

Definition 23. A *pattern* is defined by the abstract syntax

$$p ::= x \mid C(p, \ldots, p)$$

where x is a variable and C is a *constructor*. □

If we take the example of the OCaml language, a constructor can be a constant (such as `false`, `true`, 0, 1, `"hello"`, etc.), a constant constructor of an algebraic type (such as `[]` or, for example, `Empty` declared by `type t = Empty | ...`), a constructor of arity $n \geq 1$ (such as `::` or, for example, `Node` declared by `type t = Node of t * t | ...`), or a constructor of an n-tuple, with $n \geq 2$.

Definition 24 (linear pattern)**.** A pattern p is said to be *linear* if every variable appears at most once in p. □

Thus, the pattern (x, y) is linear, but (x, x) is not. In what follows, we only consider linear patterns[4]. The values being matched are constructed from the same set of constants and constructors as in the definition of patterns, that is,

$$v ::= C(v, \ldots, v)$$

We begin by defining the notion of matching a value with a single pattern.

Definition 25 (matching)**.** A value v is said to *match* a pattern p if there exists a substitution σ of variables by values such that $v = \sigma(p)$. □

We can further assume that the domain of σ, that is, the set of variables x such that $\sigma(x) \neq x$, is included in the set of variables of p. It is clear that any value matches $p = x$. On the other hand, we have the following result.

Proposition 5. A value v matches $p = C(p_1, \ldots, p_n)$ if and only if v is of the form $v = C(v_1, \ldots, v_n)$ with v_i matching p_i for all $i = 1, \ldots, n$.

Proof. Let v match p. We therefore have $v = \sigma(p)$ for some σ, so $v = C(\sigma(p_1), \ldots, \sigma(p_n))$, and it suffices to set $v_i = \sigma(p_i)$.

Conversely, if v_i matches p_i for all i, then there exist σ_i such that $v_i = \sigma_i(p_i)$. Since p is linear, the domains of σ_i are pairwise disjoint, and we therefore have $\sigma_i(p_j) = p_j$ if $i \neq j$. By setting $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$, we have

$$\begin{aligned} \sigma(p_i) &= \sigma_1(\sigma_2(\ldots \sigma_n(p_i) \ldots)) \\ &= \sigma_1(\sigma_2(\ldots \sigma_i(p_i) \ldots)) \\ &= \sigma_1(\sigma_2(\ldots v_i \ldots)) \\ &= v_i \end{aligned}$$

and thus $\sigma(p) = C(\sigma(p_1), \ldots, \sigma(p_n)) = C(v_1, \ldots, v_n) = v$. □

We now define the semantics of the `match` construct.

Definition 26 (multi-case matching)**.** In the matching

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

if v is the value of x, we say that v matches the case p_i if v matches p_i and if v does not match any p_j for all $j < i$. The result of the matching is then $\sigma(e_i)$, where σ is the substitution such that $\sigma(p_i) = v$. If, on the other hand, v does not match any p_i, the matching results in an error[5]. □

[4] OCaml only allows non-linear patterns in OR patterns such as `let x,0 | 0,x = ...`. We do not consider OR patterns here.

[5] the exception `Match_failure` in the case of OCaml

A First Algorithm. To write a pattern matching compilation algorithm, we assume the availability of a function $\mathit{constr}(e)$ that returns the constructor of the value e and a function $\texttt{\#}_i(e)$ that returns the i-th component of the value e. In other words, if $e = C(v_1, \ldots, v_n)$, then $\mathit{constr}(e) = C$ and $\texttt{\#}_i(e) = v_i$.

We start with the compilation of a single pattern matching line:

$$\mathit{code}(\texttt{match } e \texttt{ with } p \to \mathit{action}) = F(p, e, \mathit{action})$$

where the compilation function F is defined as follows:

$$\begin{array}{l}
F(x, e, \mathit{action}) = \\
\quad \texttt{let } x = e \texttt{ in } \mathit{action} \\
F(C, e, \mathit{action}) = \\
\quad \texttt{if } \mathit{constr}(e) = C \texttt{ then } \mathit{action} \texttt{ else } \mathit{error} \\
F(C(p), e, \mathit{action}) = \\
\quad \texttt{if } \mathit{constr}(e) = C \texttt{ then } F(p, \texttt{\#}_1(e), \mathit{action}) \texttt{ else } \mathit{error} \\
F(C(p_1, \ldots, p_n), e, \mathit{action}) = \\
\quad \texttt{if } \mathit{constr}(e) = C \texttt{ then} \\
\qquad F(p_1, \texttt{\#}_1(e), F(p_2, \texttt{\#}_2(e), \ldots F(p_n, \texttt{\#}_n(e), \mathit{action}) \ldots)) \\
\quad \texttt{else } \mathit{error}
\end{array}$$

Consider, for example:

```
match x with 1 :: y :: z -> y + length z
```

Its compilation yields the following:

```
if constr(x) = :: then
  if constr(#1(x)) = 1 then
    if constr(#2(x)) = :: then
      let y = #1(#2(x)) in
      let z = #2(#2(x)) in
      y + length(z)
    else error
  else error
else error
```

Note that $\texttt{\#}_2\texttt{(x)}$ is computed multiple times. We could introduce `let` bindings in the definition of F to remedy this.

We can prove the correctness of this algorithm.

Proposition 6. If $e \xrightarrow{\star} v$, then

$$\begin{array}{ll}
F(p, e, \mathit{action}) \xrightarrow{\star} \sigma(\mathit{action}) & \text{if there exists } \sigma \text{ such that } v = \sigma(p), \\
F(p, e, \mathit{action}) \xrightarrow{\star} \mathit{error} & \text{otherwise.}
\end{array}$$

Proof. We proceed by induction on p.

- If $p = x$ or $p = C$, it is immediate.
- If $p = C(p_1, \ldots, p_n)$:

- If $constr(v) \neq C$, there is no σ such that $v = \sigma(p)$, and $F(C(p_1, \ldots, p_n), e, action) = error$.
- If $constr(v) = C$, then $v = C(v_1, \ldots, v_n)$, and σ such that $v = \sigma(p)$ exists if and only if there exist σ_i such that $v_i = \sigma_i(p_i)$. If one of the σ_i does not exist, then the call $F(p_i, \#_i(e), \ldots)$ reduces to $error$, and so does $F(p, e, action)$. If, on the other hand, all σ_i exist, then by the induction hypothesis:

$$\begin{aligned} F(p, e, action) &= F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), action) \ldots)) \\ &= F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots \sigma_n(action) \ldots)) \\ &= \sigma_1(\sigma_2(\ldots \sigma_n(action) \ldots)) \\ &= \sigma(action) \end{aligned}$$

□

To match multiple lines, we replace *error* with the transition to the next line, that is:

$$\begin{aligned} &code(\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n) = \\ &\quad F(p_1, x, e_1, F(p_2, x, e_2, \ldots F(p_n, x, e_n, error) \ldots)) \end{aligned}$$

where the compilation function F now has four arguments and is defined by:

$$\begin{aligned} &F(x, e, success, failure) = \\ &\quad \texttt{let}\ x = e\ \texttt{in}\ success \\ &F(C, e, success, failure) = \\ &\quad \texttt{if}\ constr(e) = C\ \texttt{then}\ success\ \texttt{else}\ failure \\ &F(C(p_1, \ldots, p_n), e, success, failure) = \\ &\quad \texttt{if}\ constr(e) = C\ \texttt{then} \\ &\qquad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), success, failure) \ldots, failure)) \\ &\quad \texttt{else}\ failure \end{aligned}$$

The compilation of:

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

yields the following code:

```
if constr(x) = [] then
  1
else
  if constr(x) = :: then
    if constr(#1(x)) = 1 then
      let y = #2(x) in 2
    else
      if constr(x) = :: then
        let z = #1(x) in let y = #2(x) in z
      else error
  else
    if constr(x) = :: then
      let z = #1(x) in let y = #2(x) in z
    else error
```

As we can see, this algorithm is inefficient because it performs the same tests multiple times (from one line to another) and performs redundant tests (if $constr(e) \neq$ `[]`, then necessarily $constr(e) =$ `::`). We will now turn to a more efficient algorithm.

A Better Algorithm. To achieve a better result, we will consider the problem of simultaneously matching m values against n pattern lines in its entirety. We represent it as a *matrix*:

$$\left|\begin{array}{cccccc} e_1 & e_2 & \dots & e_m & & \\ p_{1,1} & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} & \to & action_n \end{array}\right|$$

This represents:

$$\begin{array}{l} \texttt{match}\ (e_1, e_2, \dots, e_m)\ \texttt{with} \\ \mid (p_{1,1}, p_{1,2}, \dots, p_{1,m}) \to action_1 \\ \mid \dots \\ \mid (p_{n,1}, p_{n,2}, \dots, p_{n,m}) \to action_n \end{array}$$

The compilation algorithm, denoted as F, proceeds recursively on the matrix.

A base case corresponds to matching with no patterns, *i.e.*, $n = 0$. In this case:

$$F \left|\begin{array}{ccc} e_1 & \dots & e_m \end{array}\right| = error$$

Another base case corresponds to matching with no values to match, *i.e.*, $m = 0$. In this case:

$$F \left|\begin{array}{cc} \to & action_1 \\ & \vdots \\ \to & action_n \end{array}\right| = action_1$$

When $n > 0$ and $m > 0$, we reduce the problem to smaller matrices. If the entire left column consists of *variables* $x_{i,1}$, *i.e.*,

$$M = \left|\begin{array}{ccccccc} e_1 & e_2 & \dots & e_m & & \\ x_{1,1} & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ \vdots & & & & & \\ x_{n,1} & p_{n,2} & \dots & p_{n,m} & \to & action_n \end{array}\right|$$

We eliminate this column by introducing `let` bindings:

$$F(M) = F \left|\begin{array}{cccccc} e_2 & \dots & e_m & & & \\ p_{1,2} & \dots & p_{1,m} & \to & \texttt{let}\ x_{1,1} = e_1\ \texttt{in}\ action_1 \\ \vdots & & & & & \\ p_{n,2} & \dots & p_{n,m} & \to & \texttt{let}\ x_{n,1} = e_1\ \texttt{in}\ action_n \end{array}\right|$$

Otherwise, the left column contains at least one constructed pattern. Suppose, for example, that this column contains three different constructors: C of arity 1, D of arity

0, and E of arity 2:

$$M = \left|\begin{array}{llllll} e_1 & e_2 & \ldots & e_m & & \\ C(q) & p_{1,2} & \ldots & p_{1,m} & \rightarrow & action_1 \\ D & p_{2,2} & & p_{2,m} & \rightarrow & action_2 \\ x & p_{3,2} & & p_{3,m} & \rightarrow & action_3 \\ E(r,s) & p_{4,2} & & p_{4,m} & \rightarrow & action_4 \\ y & p_{5,2} & & p_{5,m} & \rightarrow & action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \rightarrow & action_6 \\ E(u,v) & p_{7,2} & \ldots & p_{7,m} & \rightarrow & action_7 \end{array}\right|$$

For each constructor C, D, and E, we construct the submatrix corresponding to matching a value for that constructor. For constructor C, we construct the matrix:

$$M_C = \left|\begin{array}{llllll} \#_1(e_1) & e_2 & \ldots & e_m & & \\ q & p_{1,2} & \ldots & p_{1,m} & \rightarrow & action_1 \\ \_ & p_{3,2} & & p_{3,m} & \rightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ \_ & p_{5,2} & & p_{5,m} & \rightarrow & \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \\ t & p_{6,2} & \ldots & p_{6,m} & \rightarrow & action_6 \end{array}\right|$$

For constructor D, we construct the matrix:

$$M_D = \left|\begin{array}{llllll} e_2 & \ldots & e_m & & & \\ p_{2,2} & & p_{2,m} & \rightarrow & action_2 & \\ p_{3,2} & & p_{3,m} & \rightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ action_3 & \\ p_{5,2} & \ldots & p_{5,m} & \rightarrow & \texttt{let}\ y = e_1\ \texttt{in}\ action_5 & \end{array}\right|$$

For constructor E, we construct the matrix:

$$M_E = \left|\begin{array}{llllllll} \#_1(e_1) & \#_2(e_1) & e_2 & \ldots & e_m & & & \\ \_ & \_ & p_{3,2} & & p_{3,m} & \rightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ action_3 & \\ r & s & p_{4,2} & & p_{4,m} & \rightarrow & action_4 & \\ \_ & \_ & p_{5,2} & & p_{5,m} & \rightarrow & \texttt{let}\ y = e_1\ \texttt{in}\ action_5 & \\ u & v & p_{7,2} & \ldots & p_{7,m} & \rightarrow & action_7 & \end{array}\right|$$

Finally, we define a submatrix for other values (constructors different from C, D, and E), *i.e.*, for the variables appearing in the first column:

$$M_R = \left|\begin{array}{llllll} e_2 & \ldots & e_m & & & \\ p_{3,2} & & p_{3,m} & \rightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ action_3 & \\ p_{5,2} & \ldots & p_{5,m} & \rightarrow & \texttt{let}\ y = e_1\ \texttt{in}\ action_5 & \end{array}\right|$$

With these four matrices, we then define:

$$\begin{array}{rl} F(M) = & \texttt{case}\ constr(e_1)\ \texttt{in} \\ & \quad C \Rightarrow F(M_C) \\ & \quad D \Rightarrow F(M_D) \\ & \quad E \Rightarrow F(M_E) \\ & \quad \texttt{otherwise} \Rightarrow F(M_R) \end{array}$$

Here, `case` is a basic construct used to compare $constr(e_1)$ with C, D, and E. When there are only two constructors in the type of e_1, a simple `if then else` can be used.

When there is a finite number of constructors, a jump table can be used. When there are infinitely many constructors (e.g., strings), a binary tree or hash table can be used to implement `case`. Finally, sometimes there is only one constructor (e.g., an n-tuple), and then $F(M) = F(M_C)$ directly.

It is important to realize that this algorithm terminates. This is indeed the case because the quantity:

$$\sum_{i,j} size(p_{i,j})$$

strictly decreases with each recursive call to F, if we define the size of a pattern as follows:

$$\begin{array}{rcl} size(x) & = & 1 \\ size(C(p_1, \ldots, p_n)) & = & 1 + \sum_{i=1}^{n} size(p_i) \end{array}$$

Let's reconsider the example:

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

This corresponds to the matrix:

$$M = \left| \begin{array}{lcl} x & & \\ [] & \rightarrow & 1 \\ 1::y & \rightarrow & 2 \\ z::y & \rightarrow & z \end{array} \right|$$

We finally obtain:

```
case constr(x) in
  [] -> 1
  :: -> case constr(#1(x)) in
          1 -> let y = #2(x) in 2
          otherwise -> let z = #1(x) in let y = #2(x) in z
```

This is much better than with the previous algorithm. In particular, there are no more redundant computations here.

This new algorithm has additional advantages. For example, it allows detecting *redundant cases*. It is sufficient to note that an action does not appear in the produced code to determine that the corresponding matching case is redundant and issue a warning. This algorithm also allows detecting *non-exhaustive matches*. It is sufficient to check if *error* appears in the produced code.

Bibliographical Notes. The notion of closure was invented by P. J. Landin in 1964 [16]. To explore continuations in more detail, we highly recommend the book by Xavier Leroy, *Control Structures in Programming Languages* [19]. Luc Maranget has written numerous articles on pattern matching [17, 21, 22].

9 Compilation of Object-Oriented Languages

9.1 Compilation of Java

A Java object is a block allocated on the heap containing, on the one hand, its class and, on the other hand, the values of its various fields. If we revisit the example of our graphical elements (figure 6.1 page 105) and construct these two objects:

```
new Rectangle(0, 0, 100, 50)
new Circle(50, 50, 10)
```

We will have two blocks allocated in memory, respectively of the following form:

	Rectangle
x	0
y	0
width	100
height	50

	Circle
x	50
y	50
width	20
height	20
radius	10

We do not detail here in what form the class is stored in the object, but it is important to understand that this information is present and immutable. The value of an object is the address of the block that represents it (as already explained in Section 7.2).

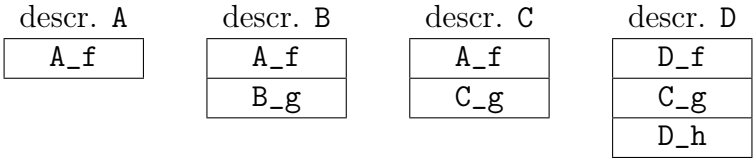
Note that Java's single inheritance allows storing the value of a field at a constant location in the block, with the class's own fields following the inherited fields. Thus, the value of `width` will always be stored in the third field of the object, whether it is a `Rectangle`, a `Circle`, or any other subclass of `Graphical`. This organization, called a *prefix*, allows compiling field access with only the static type information, without knowing the dynamic type. For each field, the compiler knows the position where this field is stored, *i.e.*, the offset to add to the object pointer. For example, if the field `width` is stored at position `+32`, then the expression `e.width` is compiled as:

```
...                  # compile e into %rcx
movq 32(%rcx), %rax  # field width
```

Method Call. The entire subtlety of compiling object-oriented languages lies in the *dynamic method call*. For this, a *class descriptor* is constructed for each class, containing the addresses of the dynamic method codes of that class. As with fields, single inheritance allows storing the address of the code of method m at a constant location in the descriptor. Class descriptors can be constructed in the data segment. Each object contains in its first field a pointer to its class descriptor. For the following four classes,

```
class A           { void f() {...} }
class B extends A { void g() {...} }
class C extends B { void g() {...} }
class D extends C { void f() {...} void h() {...} }
```

we construct four class descriptors of the following form[1]



where `A_f`, `B_g`, etc., are the addresses of the codes of the different methods. As with the arrangement of fields in objects, we have adopted a prefix organization for the arrangement of these addresses in the descriptors. Consequently, to compile a method call $e.m(e_1, \ldots, e_n)$, it is sufficient to:

1. Calculate the value of e, which is an address a pointing to an object;
2. Access the descriptor of the class of this object;
3. Find the address of the code of method m in this descriptor, with an offset that depends only on m;
4. Call this function in the traditional way, passing it the value of the object in addition to the values of the arguments $e_1, \ldots, e_n$.

We start by constructing the three class descriptors in the data segment, as illustrated in figure 9.1. Here, `Graphical_move`, `Circle_draw`, etc., correspond to the addresses of the methods.

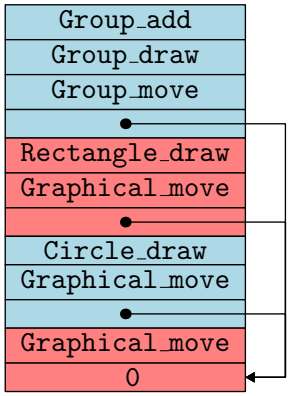
The code of a constructor is a function that assumes the object is already allocated and its address is in `%rdi`, the first field is already filled (with the class descriptor), and the constructor arguments are in `%rsi`, `%rdx`, etc., and the stack if needed. We therefore compile the constructor of the `GList` class as follows:

```
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) {
    this.g = g; this.next = next;
  }
}
```

```
new_GList:
        movq %rsi, 8(%rdi)
        movq %rdx, 16(%rdi)
        ret
```

[1] In practice, the descriptor of class C also contains an indication of the superclass of C, such as a pointer to its descriptor.

```
        .data
descr_Graphical:
        .quad 0
        .quad Graphical_move
descr_Circle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Circle_draw
descr_Rectangle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Rectangle_draw
descr_Group:
        .quad descr_Graphical
        .quad Group_move
        .quad Group_draw
        .quad Group_add
```



(addresses increase upwards)

Figure 9.1: Class descriptors for the graphical Java classes.

The constructors of the `Circle`, `Rectangle`, and `Group` classes are compiled similarly. For methods, we adopt the same convention: the object is in `%rdi` and the method arguments are in `%rsi`, `%rdx`, etc., and the stack if needed.

```
abstract class Graphical {
  void move(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

```
Graphical_move:
        addq %rsi, 8(%rdi)
        addq %rdx, 16(%rdi)
        ret
```

The `draw` method of the `Group` class is more interesting because it contains a dynamic call:

```
class Group extends Graphical {
  void draw() {
    GList l = group;
    while (l != null) {
      l.g.draw();
      l = l.next;
    }
  }
}
```

```
Group_draw:
    pushq %rbx
    movq  8(%rdi), %rbx
    jmp   2f
1:  movq  8(%rbx), %rdi
    movq  (%rdi), %rcx
    call  *16(%rcx)
    movq  16(%rbx), %rbx
2:  testq %rbx, %rbx
    jnz   1b
    popq  %rbx
    ret
```

The dynamic call is compiled by a jump to a computed address with `call *`. We had already used this instruction to compile the call to a first-class function in the previous chapter (see page 135).

Finally, let's explain how to construct an object, *i.e.*, how to compile the `new` construction, taking the first line of the main program:

```
public static void main(String arg[]){
  Rectangle g1 =
    new Rectangle(0, 0, 100, 50);
  ...
```

```
main:
  movq $40, %rdi
  call malloc
  movq %rax, %r12
  movq $descr_Rectangle,(%r12)
  movq %r12, %rdi
  movq $0, %rsi
  movq $0, %rdx
  movq $100, %rcx
  movq $50, %r8
  call new_Rectangle
  ...
```

Here, we assume the variable `g1` is allocated in register `%r12`. We start by allocating a 40-byte block on the heap with `malloc`, which is divided into 8 bytes for the pointer to the class descriptor and 8 bytes for each of the fields of the `Rectangle` class[2]. Once the result of `malloc` is obtained and copied into `%r12`, we store the class descriptor, *i.e.*, the address represented by `descr_Rectangle`, in the first field of the object. Finally, we call the constructor `new_Rectangle` by passing the object in `%rdi` and the arguments in `%rsi`, `%rdx`, `%rcx`, and `%r8`.

Exercise 46. Write the rest of the assembly code for the `main` function. Solution □

Call Optimization. For greater efficiency, we can attempt to replace a dynamic call (*i.e.*, computed during execution) with a static call (*i.e.*, known at compile time). For a call $e.m(\ldots)$, where e has a static type C, this is possible when the method m is not overridden in any subclass of C. Another, more complex possibility is to propagate types known at compile time (known as *type propagation*). In code such as:

```
B b = new B();
A a = b;
a.m();
```

we know that the dynamic type of `a` is `B`, and we can therefore deduce which method `a.m` refers to. We can then compile this call as a traditional call.

Type Casting. As we have seen, the static type and the dynamic type of an expression designating an object can differ due to subtyping. It is sometimes necessary to "force the

[2]The `int` type in Java is specified as representing 32-bit signed integers. We could therefore use only 4 bytes for each field. However, we choose here the simplicity of a uniform representation where each field occupies exactly 8 bytes, whether it is a pointer or an integer.

hand" of the compiler by pretending that an object e belongs to a certain class C, or more precisely, to one of the superclasses of C. This is called *type casting* (or *cast*). The Java construct for type casting is:

$$(C)e$$

The static type of such an expression is C. If D is the static type of the expression e and E is the dynamic type of (the object designated by) e, there are three possible situations:

- If C is a superclass of D, it is called an *upcast*, and the code produced for $(C)e$ is the same as for e. However, the *cast* influences typing since the type of $(C)e$ is C.
- If C is a subclass of D, it is called a *downcast*, and the code contains a *dynamic test* to verify that E is indeed a subclass of C.
- Finally, if C is neither a subclass nor a superclass of D, the compiler rejects the expression.

Upcasting is sometimes necessary to refer to a field that has been shadowed by another. Consider, for example, the following two classes:

```
class A           { int x = 1; }
class B extends A { int x = 2; }
```

An object of class `B` has two fields `x`, one inherited from `A` and one of its own. If `b` is an object of class `B`, then `b.x` refers to the latter. To access the former, one can write `((A)b).x`. Indeed, the expression `(A)b` has the static type `A`, which affects the compilation of the access to the field `x` (the offset is different).

Let's now give an example of downcasting, with a method `m` that attempts to call the `add` method of a `Graphical`:

```
void m(Graphical g, Graphical x) {
  ((Group)g).add(x);
}
```

Nothing guarantees that the object `g` passed to `m` will indeed be a `Group`. In particular, it might not even have an `add` method. The dynamic test is therefore necessary. The exception `ClassCastException` is raised if the test fails.

To allow for slightly more defensive programming, there is a Boolean construct:

$$e \texttt{ instanceof } C$$

which determines whether the class of e is indeed a subclass of C. As a result, one often finds the pattern:

```
if (e instanceof C) {
  C c = (C)e;
  ...
}
```

In this case, the compiler typically performs an optimization by not generating a second test for the type cast. Compiling the `e instanceof C` construct is not difficult. It is a simple loop that traverses the class hierarchy, starting from the dynamic class of `e` until reaching `C` or `Object`, the class at the top of the hierarchy.

The compiler can optimize the constructs $(C)e$ and e `instanceof` C in certain cases. If C is the only subclass of D, then a single equality test can be performed instead of a loop. If D is a subclass of C, then e `instanceof` C directly evaluates to `true`.

Exercise 47. Compile the `instanceof` construct in assembly. Solution □

9.2 Compilation of C++

Let's now illustrate the compilation of C++. Using the same example (figure 6.3 page 109), the representation of an object is not different from Java.

descr. Circle
x
y
width
height
radius

descr. Rectangle
x
y
width
height

descr. Group
x
y
width
height
group

However, in C++, we also have *multiple inheritance*. Consequently, we can no longer always use the principle that the representation of an object of a superclass of C is a prefix of the representation of an object of class C (and the same applies to class descriptors). Here is an example of multiple inheritance, where the class `FlexibleArray` inherits from both the class `Array` and the class `List`.

```
class Collection {
  int cardinal;
};
class Array : public Collection {
  int nth(int i) ...
};
class List {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray : public Array, public List {
};
```

descr. FlexibleArray
cardinal
...
descr. List
...
...

On the right, we have illustrated the representation of an object of the class `FlexibleArray`. The object representations of the two classes `Array` and `List` are juxtaposed, followed by the fields specific to the class `FlexibleArray`[3]. In particular, a cast such as:

```
FlexibleArray fa;
List l = (List)fa;
```

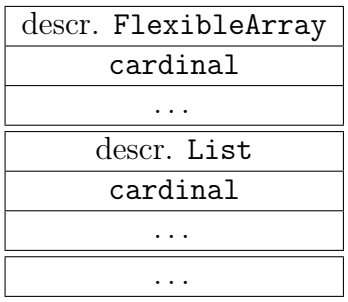
is not translated by identity but by pointer arithmetic, specifically an assignment of the form:

[3]The option `-fdump-lang-class` (formerly `-fdump-class-hierarchy`) of the GNU C++ compiler provides complete access to the representation of objects and class descriptors.

```
  l = fa + 24;
```

Now suppose that `List` also inherits from `Collection`. There will now be *two* `cardinal` fields in an object of the class `FlexibleArray`, one inherited from `Array` and the other from `List`.

```
...
class List : public Collection {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray : public Array, public List {
};
```



In particular, there is now ambiguity if we refer to the `cardinal` field of an object of the class `FlexibleArray`.

```
int main() {
  FlexibleArray fa;
  cout << fa.cardinal << endl;
};
```

```
test.cc: In function ‘int main()’:
test.cc:42:14: error: request for member ‘cardinal’ is ambiguous
```

We must specify which `cardinal` field is intended, using this notation:

```
  cout << fa.Array::cardinal << endl;
```

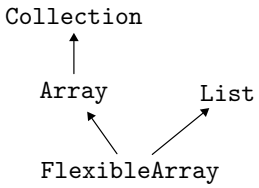
To have only one instance of `Collection` inside `FlexibleArray`, we need to modify how `Array` and `List` inherit from `Collection`, using *virtual inheritance*.

```
class Collection                                { ... };
class Array : public virtual Collection         { ... };
class List : public virtual Collection          { ... };
class FlexibleArray : public Array, public List { ... };
```

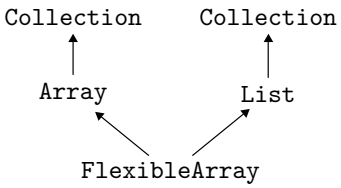
There is then only one instance of the class `Collection` in the class `FlexibleArray`, and thus in particular only one `cardinal` field.

We can summarize the three situations above by illustrating the corresponding class diagram each time.

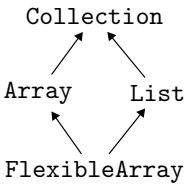
```
class Collection
class Array : Collection
class List
class FlexibleArray : Array, List
```




```
class Collection
class Array : Collection
class List : Collection
class FlexibleArray : Array, List
```




```
class Collection
class Array : virtual Collection
class List : virtual Collection
class FlexibleArray : Array, List
```



This last situation is called the *diamond* or the *rhombus*. It sometimes appears as a problem when it is accidental and has a notorious reputation — this is called the *deadly diamond of death*, which says it all!

As anticipated, multiple inheritance complicates not only the representation of objects but also that of class descriptors. For exactly the same reason, we can no longer use a prefix representation where the descriptor of the subclass is an extension of that of the superclass. As with the representation of objects, we must now juxtapose several class descriptors and perform arithmetic adjustment if necessary.

Multiple Inheritance That Doesn't Say Its Name. In the Java language, we certainly have only single inheritance, but we also have *interfaces*. A class, in addition to inheriting from another class, can implement as many interfaces as it wants.

```
interface I { ... }
interface J { int foo(); }
interface K { ... }
class B extends A implements I, J, K {
  ...
}
```

Moreover, an interface is a type, and we can write a piece of code like this that receives as a parameter any object that implements the `J` interface:

```
int f(J x) { return x.foo(); }
```

and that calls its `foo` method. From then on, we can no longer compile the method call as simply as we did in Section 9.1, because we cannot easily know where the code for the `foo` method is located in the descriptor of the class of `x`. We must resort to more complex, and notably more time-consuming, solutions. On the other hand, the representation of Java objects can remain simple, as there is only single inheritance.

10 Optimizing Compiler

In this chapter, we will write a compiler for a simple fragment of the C language to x86-64 assembly, aiming to produce reasonably efficient code. Specifically, we will seek to effectively utilize the sixteen registers and numerous instructions of the x86-64 architecture.

The fragment of the C language that we will compile includes integers (only the `int` type), pointers to structures (allocated only on the heap), functions, and control structures `if`, `while`, and `return`. The abstract syntax of this fragment is given in figure 10.1.

This fragment may seem modest, but by using a few C library functions, we can allocate structures on the heap (with `sbrk` or `malloc`) or perform input/output operations (for example, with `putchar`). To simplify our compiler a bit, we choose to use 64-bit signed integers for the `int` type, which the C standard allows us to do, even if it is not the common practice[1]. In this way, both integers and pointers occupy 64 bits.

It is unrealistic to aim for producing efficient code in a single pass. Code production is therefore be broken down into several phases. The number and nature of these phases depend on the compilers. We choose here the architecture of the CompCert compiler[2] by Xavier Leroy. This architecture is not tied to the C language. It could just as well be used to compile a functional or object-oriented language.

Our starting point is the abstract syntax tree resulting from type checking. In particular, we assume that all identifiers have been distinguished, that local variables have been distinguished from global variables, and that the type of each sub-expression is known.

10.1 Instruction Selection

The first phase of our compiler is called *instruction selection*. It involves replacing the arithmetic operations of C with those of x86-64 and replacing accesses to structure fields with explicit memory access operations.

This can be done naively by translating each C arithmetic operation into the corresponding x86-64 instruction. However, x86-64 provides instructions that allow for greater

[1] On a 64-bit architecture, a C compiler typically chooses to represent the `int` type with 32 bits, reserving the `long int` type for 64-bit integers. To make our compiler more standard, it would suffice to replace `int` with `long int`.

[2] see `http://compcert.inria.fr/`

E	::=	n	expression
		| L	
		| L `=` E	
		| E *op* E | `-` E | `!` E	
		| x`(`$E, \ldots, E$`)`	
		| `sizeof(struct` x`)`	
L	::=	x	left value
		| E`->`x	
op	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`	binary operators
		| `&&` | `||` | `+` | `-` | `*` | `/`	
S	::=	`;`	statement
		| E`;`	
		| `if` (E) S `else` S	
		| `while` (E) S	
		| `return` E`;`	
		| B	
B	::=	`{` $V \ldots V$ $S \ldots S$ `}`	block
V	::=	`int` $x, \ldots, x$`;`	variables or fields
		| `struct` x `*`$x, \ldots,$ `*`x`;`	
T	::=	`int` | `struct` x `*`	type
D	::=	V	global variable declaration
		| $T\ x(T\ x, \ldots, T\ x)\ B$	function declaration
		| `struct` x `{`$V \ldots V$`};`	structure declaration
P	::=	$D \ldots D$	program

Figure 10.1: Abstract Syntax of Mini-C.

efficiency, such as adding a register and a constant or shifting the bits of an integer to the left or right, which can be interpreted as multiplication or division by a power of two.

Additionally, it is possible, and desirable, to evaluate as many expressions as possible during compilation. This is called *partial evaluation*. For example, we can simplify the expression $(1 + e_1) + (2 + e_2)$ to $e_1 + e_2 + 3$, thus saving an addition. Similarly, we can simplify the expression $!(e_1 < e_2)$ to $e_1 \geq e_2$, replacing two operations with one. Of course, such simplifications must preserve the semantics of programs. If, for example, a left-to-right evaluation order were specified, we could only simplify $(0 - e_1) + e_2$ to $e_2 - e_1$ if the expressions e_1 and e_2 do not interfere. This is the case, for example, if e_1 and e_2 are pure, *i.e.*, without any side effects. In C, the evaluation order is not specified, and we can therefore always replace $(0 - e_1) + e_2$ with $e_2 - e_1$ if we wish, even if the expressions e_1 and e_2 interfere. It is up to the programmer not to write expressions such as `(0-x++) + ++x`.

Another example is the simplification of $e + 1 < 10$ to $e < 9$. This may seem clever, but the semantics are not always preserved. Indeed, if e is the largest integer, $e + 1$ will cause an arithmetic overflow and end up being less than 9. In *unsigned* arithmetic in C, the behavior of arithmetic overflow is specified as a calculation modulo 2^n with n being the size of the integers, and such a simplification could therefore not be performed. In *signed* arithmetic, however, overflow is undefined behavior in C (see page 41). We can therefore make this simplification for the `int` type. With `gcc -O2`, it is indeed done, and when e is `INT_MAX`, we get a different result from that obtained with `gcc -O1`, which does not optimize $e + 1 < 10$ to $e < 9$.

A final example is the simplification of $0 \times e$ to 0. This is only possible if the expression e has no side effects. Since our expressions include function calls, determining if e has no side effects is undecidable. But we can settle for a correct approximation, such as:

$$\begin{aligned}
\mathit{pure}(n) &= \mathit{true} \\
\mathit{pure}(x) &= \mathit{true} \\
\mathit{pure}(e_1 \texttt{ + } e_2) &= \mathit{pure}(e_1) \wedge \mathit{pure}(e_2) \\
&\vdots \\
\mathit{pure}(e_1 \texttt{ = } e_2) &= \mathit{false} \\
\mathit{pure}(f(e_1, \ldots, e_n)) &= \mathit{false} \quad \text{(we don't know)}
\end{aligned}$$

When e is not pure, it is still possible to simplify $0 \times e$ to 0 after evaluating e for its effects. This saves an unnecessary multiplication. In the case of a division $0/e$, we must be even more careful: evaluate e for its effects, test if $e = 0$ and signal a division by zero if necessary, and otherwise give the value 0 to the expression without performing a division. We might wonder why the programmer would have written expressions such as $0 \times e$ or $0/e$ in the first place, but we must keep in mind the use of macros (even if it is discouraged) or the compilation of automatically generated C code.

Instruction selection will transform the abstract syntax trees into new trees in a slightly different abstract syntax, where the operations are now those of x86-64. This new abstract syntax is given in figure 10.2. The main difference lies in the expressions. The rest consists solely of forgetting the types and grouping local variables at the beginning of the function and global variables at the beginning of the program.

To perform instruction selection while incorporating partial evaluation, we can use *smart constructors*. These are functions that behave like constructors of the abstract syntax, while performing simplifications on the fly. For example, we can define the following

E	::=	n	expression
	|	x	
	|	x `=` E	
	|	`load` n`(`E`)`	
	|	`store` n`(`E`)` E	
	|	*binop* E E	
	|	*unop* E	
	|	x`(`$E, \ldots, E$`)`	
binop	::=	`add` | `sub` | `imul` | `div`	x86-64 binary operators
	|	`mov` | `setl` | `setle` | ...	
unop	::=	`add` n | `sete` n | ...	x86-64 unary operators
S	::=	`;`	statement
	|	E`;`	
	|	`if` (E) S `else` S	
	|	`while` (E) S	
	|	`return` E`;`	
	|	`{` $S \ldots S$ `}`	
F	::=	$x(x, \ldots, x)$ `{` $x \ldots x$ $S \ldots S$ `}`	function declaration
P	::=	$x \ldots x$ $F \ldots F$	program

Figure 10.2: Abstract Syntax for Instruction Selection.

$IS(e_1$`+`$e_2)$	=	`mkAdd`$(IS(e_1),\ IS(e_2))$	
$IS(e_1$`-`$e_2)$	=	`mkSub`$(IS(e_1),\ IS(e_2))$	
		$\vdots$	
$IS($`!`$e_1)$	=	`mkNot`$(IS(e_1))$	
$IS($`-`$e_1)$	=	`mkNeg`$(IS(e_1))$	
$IS(e_1$`->`$x)$	=	`load` $d(IS(e_1))$	(d calculated based on x)
$IS(e_1$`->`x `=` $e_2)$	=	`store` $d(IS(e_1))$ $IS(e_2)$	(d calculated based on x)
$IS(x$`(`$e_1, \ldots, e_n$`)`$)$	=	x`(`$IS(e_1), \ldots, IS(e_n)$`)`	
$IS($`sizeof(struct` x`)`$)$	=	n	(size of the structure x)

Figure 10.3: Instruction Selection for Mini-C.

constructor for addition:

$$\begin{aligned}
\texttt{mkAdd}(n_1,\ n_2) &= n_1 + n_2 \\
\texttt{mkAdd}(0,\ e) &= e \\
\texttt{mkAdd}(e,\ 0) &= e \\
\texttt{mkAdd}((\texttt{add}\ n_1)\ e,\ n_2) &= \texttt{mkAdd}(n_1 + n_2,\ e) \\
\texttt{mkAdd}(n,\ e) &= (\texttt{add}\ n)\ e \\
\texttt{mkAdd}(e,\ n) &= (\texttt{add}\ n)\ e \\
\texttt{mkAdd}(e_1,\ e_2) &= \texttt{add}\ e_1\ e_2 \quad \text{otherwise}
\end{aligned}$$

(Be careful not to confuse the unary operator `add` n, which adds the immediate operand n to its argument, with the binary addition operator `add`.) We could do even more simplifications, for example by intelligently using the `lea` instruction. Whatever we do, we must ensure that the simplification function terminates. Here, the size of the arguments of `mkAdd` strictly decreases during the recursive call, which guarantees termination.

Once such smart constructors are defined, the translation can be done word for word. We denote $IS(e)$ as the translation of an expression e. Its definition is given in figure 10.3. Memory accesses through the `->` construct are translated into indirect accesses with an offset. This offset can be easily calculated because each structure field occupies the same size, namely 8 bytes. If x is the third field of a structure, for example, its offset will be $d = 16$. Similarly, we know the total size occupied by a structure and can therefore translate `sizeof(struct` x`)` directly to an integer.

For the rest of the abstract syntax, *i.e.*, function calls, statements, and declarations, instruction selection is a morphism. Here is an example of instruction selection:

```
struct list { int val; struct list *next; };

int print(struct list *l) {
  struct list *p;
  p = l;
  while (p) {
    int c;
    c = p->val;
    putchar(c);
    p = p->next;
  }
  return 0;
}
```

```
// no need for the list type
// anymore
print(l) {
  locals p, c;
  p = l;
  while (p) {

    c = load 0(p);
    putchar(c);
    p = load 8(p);
  }
  return 0;
}
```

And here is another example with the factorial function:

```
int fact(int x) {

  if (x <= 1) return 1;
  return x * fact(x-1);
}
```

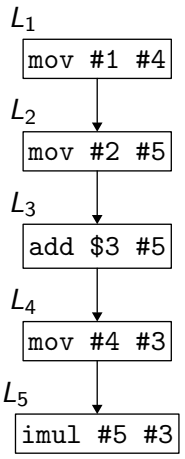
```
fact(x) {
  locals:
  if (setle x 1) return 1;
  return imul x fact((add -1) x);
}
```

This example of the factorial function will serve as our guiding thread throughout the following phases.

10.2 Generating RTL Code

The second phase involves transforming the code into a language called RTL, which stands for *Register Transfer Language*, with several objectives. First, we will break down the tree structure of expressions and statements into a *control flow graph* (CFG for short), to facilitate subsequent phases. In particular, this will eliminate the distinction between expressions and statements. Second, we will introduce the concept of *pseudo-registers*. Unlike machine registers, these pseudo-registers are unlimited in number. It is only later, in the following phases, that these pseudo-registers will become x86-64 registers as much as possible, or stack locations otherwise. Finally, RTL instructions will closely resemble x86-64 assembly instructions.

Let's illustrate the concept of the RTL language with an example. Consider the expression `b * (3 + d)`, where `b` and `d` are two local variables. We allocate pseudo-registers for these two variables, for example, `#1` for `b` and `#2` for `d`. (We denote `#`n as the n-th pseudo-register.) We assign a pseudo-register to hold the result of the expression, for example, `#3`. The RTL control flow graph that assigns the value of the expression to `#3` is shown on the right. It consists of five sequential RTL instructions. Each instruction is identified by a label, denoted as L_1, L_2, etc. The first instruction, at label L_1, copies the value of `b` into a new pseudo-register `#4`. The next two instructions calculate the value of `3+d` in another pseudo-register `#5`. The fourth instruction copies `#4` into `#3`, and finally, the last instruction multiplies `#3` by `#5`. RTL instructions take their destination operand as the last argument, following the AT&Tassembly convention.



Of course, there were other ways to proceed here. For example, we could have avoided using the pseudo-register `#4` and directly copied `#1` into `#3` in the second-to-last instruction.

The control flow graph can be represented by a dictionary associating an RTL instruction with each label. Conversely, each RTL instruction indicates the next label or labels for a branching instruction. Thus, the RTL instruction:

$$\texttt{mov}\ n\ r\ \rightarrow L$$

means "load the constant n into the pseudo-register r and transfer control to label L."

The set of RTL instructions is given in figure 10.4.

Translating an Expression. We translate an expression by defining a function $RTL(e, r_d, L_d)$, where e is the expression to translate, r_d is a pseudo-register that should receive the value of the expression, and L_d is a label L_d where control should be transferred afterward. The function RTL returns the label corresponding to the entry point in the graph for the calculation of e. Thus, we translate an expression "from the end."

To translate an addition e_1+e_2, for example, we proceed as follows:

$$\begin{array}{rcll} RTL(e_1\texttt{+}e_2, r_d, L_d) & = & \text{add } L_3 : \texttt{add}\ r_2\ r_d\ \rightarrow L_d & (r_2, L_3 \text{ fresh}) \\ & & L_2 \leftarrow RTL(e_2, r_2, L_3) & \\ & & L_1 \leftarrow RTL(e_1, r_d, L_2) & \\ & & \text{return } L_1 & \end{array}$$

This code should be read backward: we start by evaluating e_1 in r_d, then e_2 in a new pseudo-register r_2, and finally perform the addition in r_d.

A base case, reduced to a single instruction, is, for example, loading a constant n into r_d:

$$\begin{array}{rcll} RTL(n, r_d, L_d) & = & \text{add } L_1 : \texttt{mov } n\ r_d \rightarrow L_d & (L_1 \text{ fresh}) \\ & & \text{return } L_1 & \end{array}$$

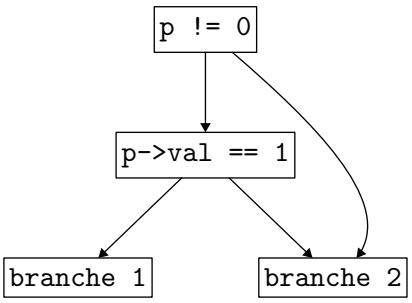
We proceed similarly with global variables and memory accesses. For a function call, the principle is the same as for an addition, with only more arguments to evaluate in pseudo-registers.

$$\begin{array}{rcll} RTL(x(e_1, \ldots, e_n), r_d, L_d) & = & \text{add } L_n : r_d \leftarrow \texttt{call } x(r_1, \ldots, r_n) \rightarrow L_d & \\ & & L_{n-1} \leftarrow RTL(e_n, r_n, L_n) & \\ & & \ldots & \\ & & L_1 \leftarrow RTL(e_1, r_1, L_2) & \\ & & \text{return } L_1 & (r_1, \ldots, r_n, L_1, \ldots, L_n \text{ fresh}) \end{array}$$

For local variables, we use a table where each variable is associated with a pseudo-register. Reading or writing a local variable is then translated with the `mov` instruction (binary operator).

Branches. To translate the operators `&&` and `||` and the statements `if` and `while`, we will use RTL *branch* instructions. The idea is to translate a program like:

```
if (p != 0 && p->val == 1)
  ...branch 1...
else
  ...branch 2...
```



into a control flow graph such as the one shown opposite (where the blocks are schematic). In particular, there are two ways to end up in the second branch, either because `p != 0` is false, or because `p != 0` is true but `p->val == 1` is false, and we want to construct the graph fragment corresponding to this second branch only once.

To achieve this, we introduce a new translation function, $RTL_c(e, L_t, L_f)$, where e is an expression to translate, interpreted as a condition, and L_t and L_f are two labels. The idea is to evaluate the expression e and then transfer control to label L_t if the value is

true and to label L_f otherwise. Here is a possible definition of the function RTL_c.

$$\begin{aligned}
RTL_c(e_1 \texttt{ \&\& } e_2, L_t, L_f) &= RTL_c(e_1,\ RTL_c(e_2, L_t, L_f),\ L_f) \\
RTL_c(e_1 \texttt{ || } e_2, L_t, L_f) &= RTL_c(e_1,\ L_t,\ RTL_c(e_2, L_t, L_f)) \\
RTL_c(e_1 \texttt{ <= } e_2, L_t, L_f) &= \text{add } L_3 : \texttt{jle } r_2\ r_1\ \rightarrow L_t, L_f \\
&\quad\ L_2 \leftarrow RTL(e_2, r_2, L_3) \\
&\quad\ L_1 \leftarrow RTL(e_1, r_1, L_2) \\
&\quad\ \text{return } L_1 \\
RTL_c(e, L_t, L_f) &= \text{add } L_2 : \texttt{jz } r\ \rightarrow L_f, L_t \\
&\quad\ L_1 \leftarrow RTL(e, r, L_2) \\
&\quad\ \text{return } L_1
\end{aligned}$$

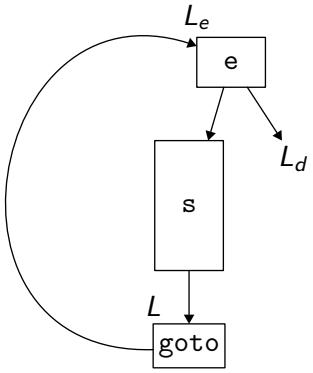
The last case corresponds to any expression e, the semantics of C being that a value is true as long as it is non-zero, regardless of its type. Of course, we can handle more specific cases before resorting to this general case.

Translating a Statement. Translating statements poses the problem of the `return` statement, which must transfer control to the function exit. We assign a pseudo-register r_{ret} to receive the function result and a label L_{ret} corresponding to the function exit. The translation of a statement s then takes the form $RTL(s, L_d)$, where L_d is the label to which control should be transferred if the statement s did not lead to `return`.

$$\begin{aligned}
RTL(\texttt{;},\ L_d) &= \text{return } L_d \\
RTL(e\texttt{;},\ L_d) &= \text{return } RTL(e, r_1,\ L_d) \\
RTL(\texttt{return } e\texttt{;},\ L_d) &= RTL(e, r_{ret},\ L_{ret}) \\
RTL(\texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2,\ L_d) &= RTL_c(e,\ RTL(s_1,\ L_d),\ RTL(s_2,\ L_d)) \\
RTL(\texttt{\{}s_1 \ldots s_n\texttt{\}},\ L_d) &= L_n \leftarrow RTL(s_n,\ L_d) \\
&\quad \ldots \\
&\quad L_1 \leftarrow RTL(s_1,\ L_2) \\
&\quad \text{return } L_1
\end{aligned}$$

To translate a `while(e)s`, we need to build a loop in the control flow graph. Since we build the code from bottom to top, this poses a small difficulty. We solve it by using the RTL instruction `goto` to close the loop *a posteriori*, like this:

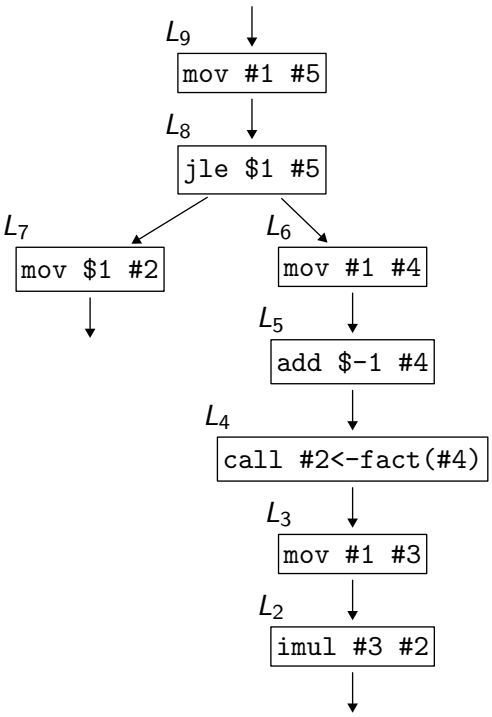
$$\begin{aligned}
RTL(\texttt{while(}e\texttt{)}s,\ L_d) &= L_e \leftarrow RTL_c(e, RTL(s,\ L),\ L_d) \\
&\quad \text{add } L : \texttt{goto } L_e \\
&\quad \text{return } L_e
\end{aligned}$$



Translating a Function. To translate a function, we start by allocating fresh pseudo-registers for its arguments, its result, and its local variables. We create a fresh label L_d for the function *exit*. Finally, we translate the body s of the function with $RTL(s, L_d)$ and note the result as the function's *entry* label.

For the `fact` function, we end up with this:

```
#2 fact(#1)
  entry : L9
  exit  : L1
  locals:
  L9: mov #1 #5            -> L8
  L8: jle $1 #5        -> L7, L6
  L7: mov $1 #2            -> L1
  L6: mov #1 #4            -> L5
  L5: add $-1 #4           -> L4
  L4: #2 <- call fact(#4) -> L3
  L3: mov #1 #3            -> L2
  L2: imul #3 #2           -> L1
```



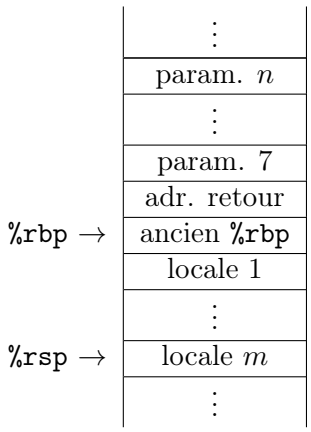
Each function is translated independently, with its own control flow graph. In this sense, our translation is *intraprocedural*, *i.e.*, without knowledge of other functions and, in particular, without knowledge of the call points of this function. In contrast, an *interprocedural* analysis could take into account the calling context of functions.

10.3 Generating ERTL Code

The third phase involves translating into a new language called ERTL, which stands for *Explicit Register Transfer Language*, with the goal of making the *calling conventions* explicit. Specifically, we will make it explicit that in a call, the first six parameters are passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, and the subsequent ones on the stack, and that the result is returned in `%rax`. We will apply these calling conventions both for our own functions and for any library functions used in our programs, such as `putchar` or `sbrk`, without distinction. Additionally, we will make it explicit that the `idivq` division

instruction requires the use of registers `%rax` and `%rdx`. Finally, we will ensure that the callee-saved registers (`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`, `%rbp`) are preserved by the callee.

The stack frame is organized as illustrated on the right. Beyond the sixth, arguments are passed on the stack. This is followed by the return address placed by `call`, then the saved `%rbp` register. Finally, there is the space allocated for local data, *i.e.*, everything that could not be allocated in registers. Thus, register allocation will determine m. Here, we use the `%rbp` register to easily retrieve both the parameters and the local variables. Indeed, the stack may be used for a function call with more than six arguments, and the value of `%rsp` varies accordingly. It would then be more complex to retrieve the various elements of the stack frame using only `%rsp`.



The ERTL instructions are given in figure 10.5. The first ten are identical to the RTL instructions, except that r can now also refer to a physical register. The `call` instruction is simplified because only the function name to be called remains. Indeed, new instructions will be inserted to load the parameters into registers and onto the stack, and to retrieve the result in `%rax`. However, we still keep the information about the number of parameters passed in registers (which will be used in phase 4). Thus, we write `call` *fact*(1) to signify a call to *fact* with a single argument passed by register.

Finally, new instructions are added to manipulate the stack. The two instructions `alloc_frame` and `delete_frame` allocate and deallocate, respectively, the local part of the activation record. These instructions have no arguments because the size is not yet known. It will only be known after register allocation, which is performed in the next phase. The instructions `push_param` and `get_param` allow the caller to place a parameter on the stack and the callee to retrieve it, respectively. The argument n of `get_param` is a relative position with respect to `%rbp`. For example, the last argument placed on the stack, if any, is located at the address `%rbp` + 16.

To translate RTL code into ERTL code, we do not change the structure of the control flow graph; we simply insert *new instructions*. We do this mainly in three places:

- at the beginning of each function, to allocate the activation record, save the callee-saved registers, and copy the parameters into the corresponding pseudo-registers;
- at the end of each function, to copy the pseudo-register containing the result into `%rax`, restore the callee-saved registers, and deallocate the activation record;
- at each call, to copy the pseudo-registers containing the parameters into `%rdi`, ..., `%r9` and onto the stack before the call, and copy `%rax` into the pseudo-register containing the result after the call.

Translating Instructions. The instructions that are not modified are loading a constant, reading and writing a global variable, reading and writing to memory, a unary operation, a binary operation other than division, and branching operations. These instructions remain placed at the same labels and transfer control to the same labels as before.

instr	::=	`mov` $n\ r \rightarrow L$	loading a constant
		`mov` $x\ r \rightarrow L$	reading a global variable
		`mov` $r\ x \rightarrow L$	writing a global variable
		`load` $n(r)\ r \rightarrow L$	reading from memory
		`store` $r\ n(r) \rightarrow L$	writing to memory
		unop $r \rightarrow L$	unary operation
		binop $r\ r \rightarrow L$	binary operation
		ubranch $r \rightarrow L, L$	unary branch
		bbranch $r\ r \rightarrow L, L$	binary branch
		`call` $r \leftarrow x(r, \ldots, r) \rightarrow L$	function call
		`goto` $\rightarrow L$	unconditional branch
ubranch	::=	`jz` | `jnz` | `jl` n | `jle` n | ...	unary branch
bbranch	::=	`jl` | `jle` | `jg` | `jge` | ...	binary branch

Figure 10.4: RTL Language.

instr	::=	`mov` $n\ r \rightarrow L$	loading a constant
		`mov` $x\ r \rightarrow L$	reading a global variable
		`mov` $r\ x \rightarrow L$	writing a global variable
		`load` $n(r)\ r \rightarrow L$	reading from memory
		`store` $r\ n(r) \rightarrow L$	writing to memory
		unop $r \rightarrow L$	unary operation
		binop $r\ r \rightarrow L$	binary operation
		ubranch $r \rightarrow L, L$	unary branch
		bbranch $r\ r \rightarrow L, L$	binary branch
		`goto` $\rightarrow L$	unconditional branch
		`call` $x(n) \rightarrow L$	function call
		`return`	explicit return instruction
		`alloc_frame` $\rightarrow L$	allocate the stack frame
		`delete_frame` $\rightarrow L$	deallocate the stack frame
		`push_param` $r \rightarrow L$	push the value of r
		`get_param` $n\ r \rightarrow L$	access a parameter on the stack

Figure 10.5: ERTL Language.

Let's now show the changes made in ERTL. We start with the case of division. In RTL, we currently have an instruction:

$$L_1 : \texttt{div}\ r_1\ r_2 \to L$$

where r_1 and r_2 are two pseudo-registers. Be careful with the direction: here, we divide r_2 by r_1. In ERTL, this becomes three instructions[3]:

$$\begin{aligned} L_1 &: \texttt{mov}\ r_2\ \texttt{\%rax} \to L_2 \\ L_2 &: \texttt{div}\ r_1\ \texttt{\%rax} \to L_3 \\ L_3 &: \texttt{mov}\ \texttt{\%rax}\ r_2 \to L \end{aligned}$$

with L_2 and L_3 being fresh labels. Even though we have added new instructions, the entry point is still L_1 and the exit point is still L.

Now consider a function call, *i.e.*, an RTL instruction:

$$L_1 : \texttt{call}\ r \leftarrow f(r_1, \ldots, r_n) \to L$$

where $r, r_1, r_2, \ldots, r_n$ are pseudo-registers. It is translated into a sequence of ERTL instructions to perform, in this order, the following tasks:

1. copy $\min(n, 6)$ arguments $r_1, r_2, \ldots$ into `%rdi`, `%rsi`, ..., `%r9`;
2. if $n > 6$, pass the others on the stack with `push_param`;
3. execute `call` $f(\min(n, 6))$;
4. copy `%rax` into r;
5. if $n > 6$, pop $8 \times (n - 6)$ bytes.

This sequence of instructions starts at the same label L_1 and transfers control to the same label L as before. For example, the RTL code:

```
L4 : #2 <- call fact(#4) -> L3
```

is translated into ERTL by the following three instructions:

```
L4 : goto          -> L12
L12: mov #4 %rdi   -> L11
L11: call fact(1)  -> L10
L10: mov %rax #2   -> L3
```

Note that we enter and exit at the same points.

[3]The x86-64 instruction `idiv` r actually divides the integer represented by the concatenation of `%rdx` and `%rax` by r, then places the quotient in `%rax` and the remainder in `%rdx`. We will make this explicit a little later, in the next phase.

Translating a Function. To translate an RTL function into an ERTL function, we translate its control flow graph by translating each RTL instruction as explained above. Additionally, we add new ERTL instructions at the entry and exit of the function. At the function entry, we start by allocating the stack frame, adding the `alloc_frame` instruction. Then, we save the callee-saved registers. For this, we assign as many fresh pseudo-registers as there are callee-saved registers and copy the values of the latter into these pseudo-registers. Finally, we copy the parameters into their pseudo-registers. If these are parameters passed in registers, we copy them with `mov`. Otherwise, we fetch them from the stack with `get_param`. If we take the example of the `fact` function and assume for simplicity that the only callee-saved registers are `%rbx` and `%r12`, we get this:

RTL

```
#2 fact(#1)
  entry : L9
  ...
  L9 : mov #1 #5    -> L8
  ...
```

ERTL

```
fact(1)
  entry : L16
  L16: alloc_frame  -> L15
  L15: mov %rbx #6  -> L14
  L14: mov %r12 #7  -> L13
  L13: mov %rdi #1  -> L9
  L9 : mov #1 #5    -> L8
  ...
```

The registers `%rbx` and `%r12` are saved in `#6` and `#7`, respectively. Note that the entry point of the control flow graph has changed; it is now `L16` instead of `L9`. We have added four instructions, which are then followed by what was previously the RTL code, starting from `L9`.

At the function exit, we add a `mov` instruction to copy the pseudo-register containing the result into `%rax`. Then, we restore the callee-saved registers by copying the values from the pseudo-registers used for their saving. Finally, we deallocate the stack frame with `delete_frame` before performing `return`. For the `fact` function, we get this:

RTL

```
#2 fact(#1)
  entry : L9
  exit  : L1
  ...
  L7 : mov $1 #2    -> L1
  ...
  L2 : imul #3 #2   -> L1
```

ERTL

```
fact(1)
  entry : L16
  ...
  L7 : mov $1 #2    -> L1
  ...
  L2 : imul #3 #2   -> L1
  L1 : goto         -> L21
  L21: mov #2 %rax  -> L20
  L20: mov #6 %rbx  -> L19
  L19: mov #7 %r12  -> L18
  L18: delete_frame -> L17
  L17: return
```

In the RTL code, there were two jumps to the exit label `L1`, corresponding to the two `return` instructions in the C code. In the ERTL code, we still have these two jumps to `L1`, but ERTL instructions have now been added at this point. In particular, the result,

contained in `#2`, is copied into `%rax` and the callee-saved registers are restored by taking the values from `#6` and `#7`. Note that there is no longer a notion of an exit label in the ERTL code, as we now have an explicit `return` instruction.

The complete ERTL code for the `fact` function is given in figure 10.6 page 172. This is still far from what we imagine to be good x86-64 code for this function. At this point, several things must be understood. On the one hand, register allocation (phase 4) will attempt to associate physical registers with pseudo-registers in order to limit the use of the stack and also to remove certain instructions. For example, if we implement `#7` with `%r12`, we simply remove the two instructions `L15` and `L20`. On the other hand, the code is not yet organized linearly (the graph is only displayed arbitrarily). This will be the task of phase 5, which will attempt, among other things, to minimize jumps.

Tail Call Optimization. It is at the RTL to ERTL translation level that we must perform the optimization of *tail calls* (see section 8.2). Indeed, the instructions to be produced are not the same, and this change will have an influence in the next phase of register allocation.

However, there is a difficulty if the function called by a tail call does not have the same number of arguments passed on the stack or local variables. Indeed, the structure of the stack frame must then be modified. There are at least two solutions. For example, we can limit the optimization of the tail call to cases where the structure of the stack frame does not change. This is particularly the case if it is a tail call of a recursive function to itself. Another solution is for the caller to modify the stack frame and transfer control to the callee *after* the instruction to create its frame.

10.4 Generating LTL Code

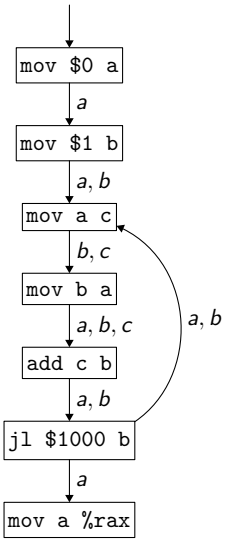
The fourth phase involves translating into a language called *LTL* for *Location Transfer Language*. The goal is to eliminate pseudo-registers in favor of physical registers, preferably, or stack locations, otherwise. This is known as *register allocation*. This is a complex phase of the compiler, which we will break down into several steps. We will start by performing a *liveness analysis*. This involves determining the exact moments when the value of a pseudo-register is needed for the continuation of the computation. With this information, we will construct an *interference graph*. This is a graph indicating which pseudo-registers cannot be realized by the same location. Finally, we will perform the actual register allocation by *coloring this graph*.

10.4.1 Liveness Analysis

In what follows, we call a *variable* a pseudo-register or a physical register. Liveness analysis concerns all variables, with the meaning given by the following definition.

Definition 27 (live variable)**.** At a point in the program, a variable is said to be *live* if the value it contains is likely to be used in the subsequent execution. □

We use the phrase "is likely to be used" because the property "is used" is not decidable. We will therefore settle for an approximation. This approximation must be correct, in the sense that a negative answer, *i.e.*, the variable v is not live, means that it is guaranteed that the value of v is no longer used in the subsequent computation.



Let's take the example of the ERTL code whose control flow graph is represented on the right. The variables here are `a`, `b`, `c`, and `%rax`. On each edge of the graph, we have indicated the live variables. For example, `a` is live on the second edge from the top because its value is used in the third instruction (`mov a c`) and `a` has not been assigned in the meantime. The variable `b`, on the other hand, is not live on this edge because its value is defined by the second instruction (`mov $1 b`). However, it is live on the third edge because its value is used by the instruction `mov b a`, which copies it into `a`. The variable `%rax` is never live because its value is never used, but only defined by the very last instruction. It is important to fully understand this example.

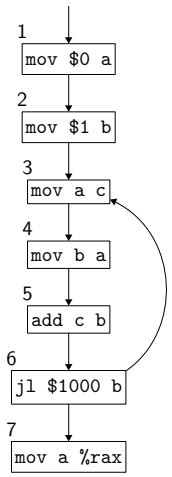
The notion of a live variable is derived from the *definitions* and *uses* of variables performed by each instruction. For instance, the instruction `add` r_1 r_2 uses the variables r_1 and r_2, and defines the variable r_1. For an instruction located at label l in the graph, we denote $def(l)$ as the set of variables defined by this instruction and $use(l)$ as the set of variables used by this instruction.

To compute the live variables, it is convenient to associate them not with the edges but rather with the *nodes* of the control flow graph, *i.e.*, with each instruction. However, we must then distinguish between variables that are *live at the entry* of an instruction and those that are *live at the exit*. For an instruction located at label l in the graph, we denote $in(l)$ as the set of live variables on all edges arriving at l and $out(l)$ as the set of live variables on all edges leaving l. The equations that define $in(l)$ and $out(l)$ are as follows:

$$\begin{cases} in(l) &= use(l) \cup (out(l) \setminus def(l)) \\ out(l) &= \bigcup_{s \in succ(l)} in(s) \end{cases}$$

These are recursive equations whose smallest solution is the one that interests us. We are dealing with a monotonic function on a finite domain (the subsets of the finite set of variables), and we can therefore apply the Knaster–Tarski theorem (see page 68). This means that we start with empty sets $in(l)$ and $out(l)$ for all instructions, then we apply the above equations until we reach a fixed point. If we take the example above, numbering the instructions from 1 to 7 from top to bottom, we converge after seven iterations[4]:

[4]There is no relation, *a priori*, between the number of instructions and the number of iterations.



			iteration 1		iteration 2			iteration 7	
	use	*def*	*in*	*out*	*in*	*out*		*in*	*out*
1		a					...		a
2		b				a	...	a	a, b
3	a	c	a		a	b	...	a, b	b, c
4	b	a	b		b	b, c	...	b, c	a, b, c
5	b, c	b	b, c		b, c	b	...	a, b, c	a, b
6	b		b		b	a	...	a, b	a, b
7	a		a		a		...	a	

Assuming the control-flow graph has N nodes and N variables, a brute force computation has complexity $O(N^3)$ in the worst case. Our control flow graph for the `fact` function already contains 22 nodes and more than 20 variables, while the C code is only two lines long. Fortunately, we can improve the efficiency of the above calculation in several ways. First, we can perform the calculations in the "reverse order" of the control flow graph and calculate *out* before *in*. In the previous example, we converge in three iterations instead of seven. Next, we can merge nodes that have only one predecessor and one successor[5]. Finally, we can use a more subtle algorithm that only recalculates the values of *in* and *out* that may have changed; this is Kildall's algorithm. The idea is as follows: if $in(l)$ changes, then we need to recalculate only for the predecessors of l. It can be sketched as follows:

```
let WS be a set containing all nodes
while WS is not empty
  extract a node l from WS
  old_in <- in(l)
  out(l) <- ...
  in(l)  <- ...
  if in(l) is different from old_in(l) then
    add all predecessors of l to WS
```

Calculating *def* and *use*. The calculation of the sets $def(l)$ (definitions) and $use(l)$ (uses) is immediate for most ERTL instructions. Here are all the simple cases:

	def	*use*
`mov` n r	$\{r\}$	$\emptyset$
`mov` x r	$\{r\}$	$\emptyset$
`mov` r x	$\emptyset$	$\{r\}$
`load` $n(r_1)$ r_2	$\{r_2\}$	$\{r_1\}$
`store` r_1 $n(r_2)$	$\emptyset$	$\{r_1, r_2\}$
unop r	$\{r\}$	$\{r\}$
binop r_1 r_2	$\{r_2\}$	$\{r_1, r_2\}$

	def	*use*
ubranch r	$\emptyset$	$\{r\}$
bbranch r_1 r_2	$\emptyset$	$\{r_1, r_2\}$
`goto`	$\emptyset$	$\emptyset$
`alloc_frame`	$\emptyset$	$\emptyset$
`delete_frame`	$\emptyset$	$\emptyset$
`push_param` r	$\emptyset$	$\{r\}$
`get_param` n r	$\{r\}$	$\emptyset$

[5]This is called a *basic block*.

We make a special case for division, which takes the dividend in the set `%rdx` : `%rax` and places the quotient and remainder in `%rax` and `%rdx`, respectively.

	def	*use*
`div` r `%rax`	{`%rax`, `%rdx`}	{`%rax`, `%rdx`, r}

Finally, there are the `call` and `return` instructions. For an instruction `call` $f(k)$, the integer k indicates the number of parameters passed in registers, which defines the registers used by the call. We also express that all caller-saved registers can be overwritten by the call.

	def	*use*
`call` $f(k)$	*caller-saved*	the first k of `%rdi`,`%rsi`,...,`%r9`

Finally, for the `return` instruction, we express the fact that `%rax` and all *callee-saved* registers are likely to be used.

	def	*use*
`return`	$\emptyset$	{`%rax`} $\cup$ *callee-saved*

The figure 10.7 gives the result of the liveness analysis for the `fact` function.

10.4.2 Interference Graph

We will now use the result of the liveness analysis to construct an *interference graph* that expresses the constraints on the possible locations for the pseudo-registers.

Definition 28 (interference)**.** Two variables v_1 and v_2 are said to *interfere* if they cannot be realized by the same location (physical register or memory location). □

Since interference is not decidable, we will settle for sufficient conditions. Consider an instruction that defines a variable v: any other variable w live at the exit of this instruction may interfere with v. However, in the special case of an instruction

`mov` w v

we do not want to declare that v and w interfere because it may be precisely interesting to realize v and w by the same location and thus eliminate one or more instructions. We therefore adopt the following definition.

Definition 29 (interference graph)**.** The *interference graph* of a function is an undirected graph whose vertices are the variables of this function and whose edges are of two types: interference or preference. For each instruction that defines a variable v and whose live variables at the exit, other than v, are $w_1, \ldots, w_n$, we proceed as follows:

- if the instruction is not a `mov` w v instruction, we add the n interference edges $v - w_i$;
- if it is a `mov` w v instruction, we add the interference edges $v - w_i$ for all w_i different from w and we add the preference edge $v - w$.

If an edge $v - w$ is both a preference and an interference edge, we keep only the interference edge. □

```
fact(1)
  entry : L16
  locals: #6,#7
  L16: alloc_frame  -> L15
  L15: mov %rbx #6  -> L14
  L14: mov %r12 #7  -> L13
  L13: mov %rdi #1  -> L9
  L9: mov #1 #5     -> L8
  L8: jle $1 #5  -> L7, L6
  L7: mov $1 #2     -> L1
  L1: goto          -> L21
  L21: mov #2 %rax  -> L20
  L20: mov #6 %rbx  -> L19
  L19: mov #7 %r12  -> L18
  L18: delete_frame -> L17
  L17: return
  L6: mov #1 #4     -> L5
  L5: add $-1 #4    -> L4
  L4: goto          -> L12
  L12: mov #4 %rdi  -> L11
  L11: call fact(1) -> L10
  L10: mov %rax #2  -> L3
  L3: mov #1 #3     -> L2
  L2: imul #3 #2    -> L1
```

Figure 10.6: ERTL Code for the `fact` Function.

	in	*out*
L16: alloc_frame --> L15	%r12,%rbx,%rdi	%r12,%rbx,%rdi
L15: mov %rbx #6 --> L14	%r12,%rbx,%rdi	#6,%r12,%rdi
L14: mov %r12 #7 --> L13	#6,%r12,%rdi	#6,#7,%rdi
L13: mov %rdi #1 --> L9	#6,#7,%rdi	#1,#6,#7
L9 : mov #1 #5 --> L8	#1,#6,#7	#1,#5,#6,#7
L8 : jle $1 #5 -> L7, L6	#1,#5,#6,#7	#1,#6,#7
L7 : mov $1 #2 --> L1	#6,#7	#2,#6,#7
L1 : goto --> L21	#2,#6,#7	#2,#6,#7
L21: mov #2 %rax --> L20	#2,#6,#7	#6,#7,%rax
L20: mov #6 %rbx --> L19	#6,#7,%rax	#7,%rax,%rbx
L19: mov #7 %r12 --> L18	#7,%rax,%rbx	%r12,%rax,%rbx
L18: delete_frame --> L17	%r12,%rax,%rbx	%r12,%rax,%rbx
L17: return	%r12,%rax,%rbx	
L6 : mov #1 #4 --> L5	#1,#6,#7	#1,#4,#6,#7
L5 : add $-1 #4 --> L4	#1,#4,#6,#7	#1,#4,#6,#7
L4 : goto --> L12	#1,#4,#6,#7	#1,#4,#6,#7
L12: mov #4 %rdi --> L11	#1,#4,#6,#7	#1,#6,#7,%rdi
L11: call fact(1) --> L10	#1,#6,#7,%rdi	#1,#6,#7,%rax
L10: mov %rax #2 --> L3	#1,#6,#7,%rax	#1,#2,#6,#7
L3 : mov #1 #3 --> L2	#1,#2,#6,#7	#2,#3,#6,#7
L2 : imul #3 #2 --> L1	#2,#3,#6,#7	#2,#6,#7

Figure 10.7: Liveness Analysis for the `fact` Function.

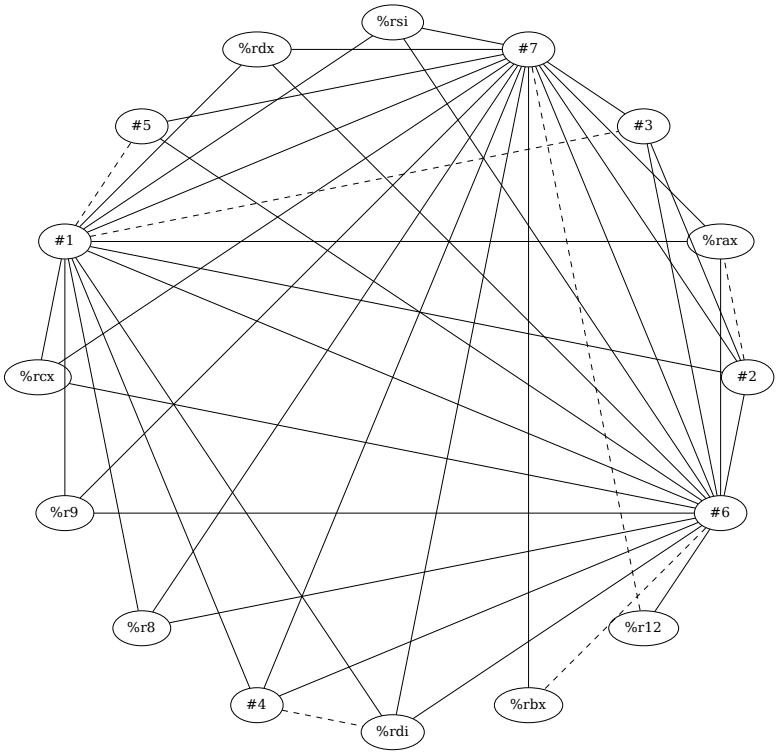


Figure 10.8: Interference graph for the `fact` function.

Figure 10.8 shows the graph obtained for the `fact` function, considering its 7 pseudo-registers and only 9 physical registers (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, `%rax`, `%rbx`, and `%r12`), as some registers have been omitted for simplicity[6].

Preference edges are indicated with dashed lines. For example, there is a preference edge between `#2` and `%rax`, which comes from the instruction `mov #2 %rax`. We observe that some vertices have a high degree. For `#6` and `#7`, this is explained by the fact that we saved the callee-saved registers there, and thus their liveness extends over the entire function. The vertex `#1` also has a high degree because it contains the argument of the `fact` function, which is used throughout the function and particularly after the recursive call.

10.4.3 Coloring the Interference Graph

Once the interference graph is constructed, we can view the register allocation problem as a *graph coloring* problem, where the colors are the physical registers. Two vertices connected by an interference edge cannot receive the same color. Two vertices connected by a preference edge should receive the same color as much as possible. There are vertices in the graph that are physical registers, *i.e.*, vertices that are already colored.

If we carefully observe the graph of the `fact` function, we notice that coloring is impossible. Indeed, we only have two colors to color `#1`, `#6`, and `#7`, namely `%rbx` and

[6]More precisely, we ignored three callee-saved registers for simplicity (`%r13`, `%r14`, and `%r15`); additionally, two caller-saved registers are missing (`%r10` and `%r11`) for a reason that will be explained later; finally, the registers `%rbp` and `%rsp` do not participate in register allocation.

`%r12`, and these three pseudo-registers interfere pairwise. Fortunately, we are not in a typical graph coloring situation: if a vertex cannot be colored, we can always allocate it on the stack. We then say that it is *spilled*..

Even if the graph could be colored, determining this would be too costly because it is an NP-complete problem. We will therefore color using *heuristics*, with the objectives of linear or quasi-linear complexity and good exploitation of preference edges. A widespread technology today is called *Iterated Register Coalescing*. It exploits the following ideas.

Let K be the number of colors, *i.e.*, the number of physical registers ($K = 9$ in our example). A first idea, due to Kempe, dates back to 1879: if a vertex has a degree strictly less than K, then we can remove it from the graph, color the rest, and we will then be assured of being able to give it a color. This step is called *simplification*. The removed vertices are then placed on a stack. Since removing a vertex decreases the degree of other vertices, this can produce new candidates for simplification.

When only vertices with a degree greater than or equal to K remain, we choose one as a candidate to be spilled to memory (a *potential spill*). It is then removed from the graph, placed on the stack, and the simplification process can resume. We preferably choose a vertex that is rarely used, because memory accesses are costly, and that has a high degree, to favor future simplifications.

When the graph is empty, we begin the actual coloring process, called *selection*. We pop the vertices one by one and for each one, we assign a color as follows. If it is a vertex with a low degree, we are assured of finding a color for it. If, on the other hand, it is a vertex with a high degree, *i.e.*, a candidate to be spilled to memory, then one of two things will happen: either it can still be colored because its neighbors use fewer than K colors in total (this is called *optimistic coloring*); or it cannot be colored and must indeed be spilled to memory (an *actual spill*).

Finally, we must make the best use of preference edges. To do this, we use a technique called *coalescing*, which consists of merging two vertices of the graph. Since this can increase the degree of the resulting vertex, we add a sufficient criterion to ensure that K-colorability is not compromised. Here is an example of such a criterion:

Definition 30 (George's criterion)**.** A pseudo-register vertex v_2 can be merged with a vertex v_1 if every neighbor of v_1 that is a physical register or has a degree $\geq K$ is also a neighbor of v_2. Similarly, a physical vertex v_2 can be merged with a vertex v_1 if every neighbor of v_1 that is a pseudo-register or has a degree $\geq K$ is also a neighbor of v_2. □

Pseudocode for register allocation is given in figure 10.9. It is naturally written in the form of five mutually recursive functions, which take as an argument the graph g to be colored and return the coloring. The function `merge(g, v1, v2)` constructs a new graph by merging the vertices `v1` and `v2` into a single vertex `v2`. For the notion of cost used in the `spill` function, we can use, for example:

$$cost(v) = \frac{number\ of\ uses\ of\ v}{degree\ of\ v}$$

If we apply this algorithm to the interference graph of the `fact` function (figure 10.8 page 173), we perform the following actions in sequence:

1. `simplify` calls `coalesce`, which merges `#3` with `#1`.

```
simplify(g) =
  if there exists a vertex v without a preference edge
     with minimal degree and < K
  then
    select(g, v)
  else
    coalesce(g)

coalesce(g) =
  if there exists a preference edge v1-v2
     satisfying George's criterion
  then
    g <- merge(g, v1, v2)
    c <- simplify(g)
    c[v1] <- c[v2]
    return c
  else
    freeze(g)

freeze(g) =
  if there exists a vertex v with minimal degree < K
  then
    g <- forget the preference edges of v
    simplify(g)
  else
    spill(g)

spill(g) =
  if g is empty
  then
    return the empty coloring
  else
    choose a vertex v with minimal cost
    select(g, v)

select(g, v) =
  c <- simplify(g without v)
  if there exists a possible color r for v
  then
    c[v] <- r
  else
    c[v] <- spill
  return c
```

Figure 10.9: Pseudocode for register allocation.

2. `simplify` calls `coalesce`, which merges `#5` with `#1`.
3. `simplify` calls `coalesce`, which merges `#2` with `%rax`.
4. `simplify` calls `coalesce`, which merges `#4` with `%rdi`.
5. `simplify` calls `coalesce`. This time there are no more preference edges satisfying George's criterion. So `coalesce` then calls `freeze`, which calls `spill`, which calls `select` with `#6`.
6. `simplify` calls `select` with `#1`.
7. `simplify` calls `coalesce`, which merges `#7` with `%r12`.

At this point, the graph no longer contains any pseudo-registers. As a result, `simplify` calls `coalesce`, which calls `freeze`, which calls `spill`, which returns an empty coloring. We now pop one by one the vertices removed from the graph, assigning a color to each one, either in `coalesce` or in `select`.

1. `coalesce` assigns the color `%r12` to `#7`.
2. `select` assigns the color `%rbx` to `#1`.
3. `select` assigns the color `spill` to `#6`.
4. `coalesce` assigns the color `%rdi` to `#4`.
5. `coalesce` assigns the color `%rax` to `#2`.
6. `coalesce` assigns the color of `#1` to `#5`, *i.e.*, `%rbx`.
7. `coalesce` assigns the color of `#1` to `#3`, *i.e.*, `%rbx`.

It remains to allocate on the stack the registers spilled to memory. We can try to minimize this space. Indeed, several pseudo-registers can occupy the same stack location if they do not interfere. Additionally, we always want to take preference edges into account to eliminate `mov` operations between registers spilled to memory, which would be very costly. This is again a graph coloring problem, but this time with an infinite number of possible colors, each color corresponding to a different stack location. In our example of the `fact` function, there is only one register spilled to memory, namely `#6`. We therefore reserve a single location on the stack, and `#6` is allocated to the location `-8(%rbp)`.

10.4.4 Translation to LTL

We can now translate our ERTL program into the LTL language. The LTL instructions are given in figure 10.10. In these instructions, r denotes a physical register, and o denotes an operand that can be a physical register or a stack location. The instructions `alloc_frame`, `delete_frame`, and `get_param` have disappeared in favor of explicit manipulation of `%rsp` and `%rbp`. The `call` instruction is now limited to the function name, as it is no longer necessary to remember the number of arguments passed in registers. The `push_param` instruction is now a more general `push` instruction, and a new `pop` instruction appears.

We translate each ERTL instruction into one or more LTL instructions, using the graph coloring and the structure of the activation record, which is now known. A variable `r` appearing in the ERTL code can already be a physical register, a pseudo-register realized by a physical register, or a pseudo-register realized by a stack location. In some cases, the translation is easy because the assembly instruction allows all combinations. For example, the ERTL instruction:

$$L_1 : \texttt{mov}\ n\ r \to L$$

becomes the LTL instruction:

$$L_1 : \texttt{mov}\ n\ color(r) \to L$$

whether $color(r)$ is a physical register, as in `mov $42, %rax`, or a stack location, as in `mov $42, -8(%rbp)`. In other cases, however, it is more complicated because not all operands are allowed by the x86-64 instruction set. For example, the case of accessing a global variable:

$$L_1 : \texttt{mov}\ x\ r \to L$$

poses a problem when `r` is allocated on the stack because we cannot write[7]:

```
movq x, n(%rbp)
```

We must therefore use an intermediate register. The problem is that we have just performed register allocation. A simple solution is to reserve two specific registers, which will be used as temporary registers for these memory transfers and will not be used otherwise. In practice, we do not necessarily have the luxury of wasting two registers in this way. We must then modify the interference graph and restart register allocation to determine a free register for the transfer. Fortunately, this converges very quickly in practice, in just two or three steps.

If we choose the first solution, for example by using `%r10` and `%r11`, we can then easily translate each ERTL instruction. Thus, to translate the ERTL instruction:

$$L_1 : \texttt{mov}\ x\ r \to L$$

we consider two cases. If $color(r)$ is a physical register hw, we have an LTL instruction:

$$L_1 : \texttt{mov}\ x\ hw \to L$$

If, on the other hand, $color(r)$ is a stack location $n(\texttt{\%rbp})$, then we have two LTL instructions:

$$\begin{aligned} L_1 &: \texttt{mov}\ x\ \texttt{\%r10} \to L_2 \\ L_2 &: \texttt{mov}\ \texttt{\%r10}\ n(\texttt{\%rbp}) \to L \end{aligned}$$

where L_2 is a fresh label. The same applies to reading the content of a variable. We sometimes need both temporaries, for example in the case of an ERTL instruction:

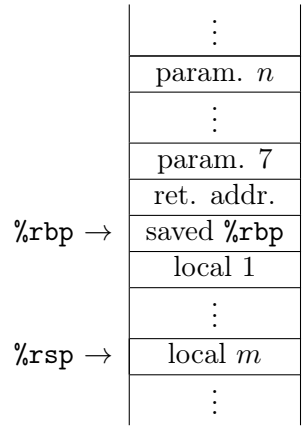
$$L_1 : \texttt{store}\ r_1\ n(r_2) \to L$$

where r_1 and r_2 are both allocated on the stack. During the translation to LTL, we apply special treatment in some cases. On the one hand, the instruction $\texttt{mov}\ r_1\ r_2 \to L$ is translated by $\texttt{goto} \to L$ when r_1 and r_2 have the same color. This is where we reap

[7] The assembler emits the error `too many memory references for 'movq'`.

the benefits of good register allocation. On the other hand, the x86-64 instruction `imul` requires that its second operand be a register. We must use a temporary if this is not the case. Finally, a binary operation cannot have both operands in memory. We must use a temporary if this is not the case.

We now know the size of the activation record. If n is the number of function arguments and m is the number of stack locations used by register allocation, the activation record has the structure shown on the right, where each cell occupies 8 bytes here. We translate `alloc_frame` by:




```
push %rbp
mov %rsp %rbp
add $ − 8m %rsp
```

and `delete_frame` by:

```
mov %rbp %rsp
pop %rbp
```

In the case where $m = 0$, we can simplify to a simple `push` and a simple `pop`, respectively. Finally, we can translate the ERTL instruction `get_param` k r in terms of access relative to `%rbp`, i.e.[8] `mov` k(`%rbp`) $color(r)$.

For the `fact` function, we finally obtain the LTL code given in figure 10.11. This code contains many `goto` instructions that will disappear during the next phase.

10.5 Generating x86-64 Assembly Code

We have one last step. Our code is still in the form of a *control flow graph*, and the goal is to produce *linear assembly code*. More precisely, the LTL conditional branching instructions contain two labels, one label for the positive test case and another for the negative test case, while the assembly conditional branching instructions contain a single label for the positive case and continue execution on the next instruction for the negative test case. This final transformation we need to perform is called *linearization*.

To perform this transformation, we traverse the control flow graph and produce the x86-64 code while noting in a table the labels already visited. When branching, we try as much as possible to produce natural assembly code if the part of the code corresponding to a negative test has not yet been visited. In the worst case, we use an unconditional branch (`jmp`). We also use a second table to note the labels that will need to appear in the final assembly code as jump destinations. Indeed, we do not want to produce assembly code where each instruction is preceded by a label. This would be correct but unreadable[9].

Linearization can be achieved using two mutually recursive functions. The first function, `lin`, produces the code from a given label if it has not already been produced, and a

[8] If $color(r)$ is a stack location, we must use a temporary.

[9] Rather than noting the destination labels of jumps in a table during linearization, we could also calculate this set of labels beforehand.

instr	::=	`mov` $n\ o \rightarrow L$	loading a constant
		`mov` $x\ r \rightarrow L$	reading a global variable
		`mov` $r\ x \rightarrow L$	writing a global variable
		`load` $n(r)\ r \rightarrow L$	reading from memory
		`store` $r\ n(r) \rightarrow L$	writing to memory
		unop $o \rightarrow L$	unary operation
		binop $o\ o \rightarrow L$	binary operation
		ubranch $r \rightarrow L, L$	unary branch
		bbranch $r\ r \rightarrow L, L$	binary branch
		`goto` $\rightarrow L$	unconditional branch
		`call` $x \rightarrow L$	function call
		`return`	explicit return instruction
		`push` $o \rightarrow L$	pushing a value
		`pop` $r \rightarrow L$	popping a value

Figure 10.10: LTL Language.

```
fact()
  entry : L16
  L16: push %rbp          -> L23
  L23: mov %rsp %rbp      -> L22
  L22: add $-8 %rsp       -> L15
  L15: mov %rbx -8(%rbp) -> L14
  L14: goto               -> L13
  L13: mov %rdi %rbx      -> L9
  L9 : goto               -> L8
  L8 : jle $1 %rbx     -> L7, L6
  L7 : mov $1 %rax        -> L1
  L1 : goto               -> L21
  L21: goto               -> L20
  L20: mov -8(%rbp) %rbx  -> L19
  L19: goto               -> L18
  L18: mov %rbp %rsp      -> L24
  L24: pop %rbp           -> L17
  L17: return
  L6 : mov %rbx %rdi      -> L5
  L5 : add $-1 %rdi       -> L4
  L4 : goto               -> L12
  L12: goto               -> L11
  L11: call fact          -> L10
  L10: goto               -> L3
  L3 : goto               -> L2
  L2 : imul %rbx %rax     -> L1
```

Figure 10.11: LTL Code for the `fact` function.

jump instruction to that label otherwise. The second function, `instr`, translates the instruction corresponding to a given label unconditionally and calls `lin` for the continuation of the linearization.

```
lin(L) =
  if L has not been visited
  then
    mark L as visited
    instr(L)
  else
    mark the label L as necessary
    produce the instruction "jmp L"
```

The `instr` function translates each LTL instruction into one or more x86-64 instructions. In some cases, it is a direct translation, such as for loading a constant or reading a global variable:

$$\begin{aligned}\texttt{instr}(L_1 : \texttt{mov}\ n\ d \to L) = &\quad \text{produce } L_1 : \texttt{mov}\ n\ d\\ &\quad \text{call } \texttt{lin}(L)\\ \texttt{instr}(L_1 : \texttt{mov}\ x\ r \to L) = &\quad \text{produce } L_1 : \texttt{mov}\ x\ r\\ &\quad \text{call } \texttt{lin}(L)\end{aligned}$$

The interesting case is that of a branch. We first consider the favorable case where the code corresponding to a negative test has not yet been produced. It can therefore be placed immediately after the conditional branch[10].

$$\begin{aligned}\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = &\quad \text{produce } L_1 : \texttt{j}cc\ L_2\\ &\quad \text{call } \texttt{lin}(L_3)\\ &\quad \text{call } \texttt{lin}(L_2)\end{aligned}$$

Otherwise, it is possible that the code corresponding to the positive test (L_2) has not yet been produced, and we can then advantageously *invert the branch condition.*

$$\begin{aligned}\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = &\quad \text{produce } L_1 : \texttt{j}\overline{cc}\ L_3\\ &\quad \text{call } \texttt{lin}(L_2)\\ &\quad \text{call } \texttt{lin}(L_3)\end{aligned}$$

where the condition $\overline{cc}$ is the inverse of the condition cc. Finally, in the case where the code corresponding to both branches has already been produced, we have no choice but to produce an unconditional branch.

$$\begin{aligned}\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = &\quad \text{produce } L_1 : \texttt{j}cc\ L_2\\ &\quad \text{produce } \texttt{jmp}\ L_3\end{aligned}$$

We can try to estimate which condition will be true most often so that the branch is taken less often. If it is a loop test, for example, we can consider that it is more often true, as we rarely write loops with at most one iteration.

As we have seen, the LTL code contains many `goto` instructions, from various origins: `while` loops in the RTL phase, code insertion in the ERTL phase, removal of `mov`

[10]It is not necessary to call $\texttt{lin}(L_2)$ afterward if this code has already been produced, but it is not incorrect. It will simply produce an unnecessary and unreachable `jmp` instruction.

instructions in the LTL phase. We strive to eliminate them when possible.

$$\begin{aligned} \texttt{instr}(L_1 : \texttt{goto} \to L_2) = &\ \text{produce } \texttt{jmp } L_2 && \text{if } L_2 \text{ has already been visited} \\ = &\ \text{produce the label } L_1 \\ &\ \text{call } \texttt{lin}(L_2) && \text{otherwise} \end{aligned}$$

In the end, we obtain for the `fact` function the x86-64 code given in figure 10.12. This code is not optimal, but it is nevertheless on the order of what `gcc -O1` produces. It is much better than that produced by `gcc -O0` or `clang -O0`. However, it is not as good as that produced by `gcc -O2` or `clang -O1`, where the `fact` function is transformed into a loop. Without discussing such a radical transformation, we can note several points where our assembly code is not perfect. First, the activation record is created and `%rbx` is saved there before testing if `x <= 1`. When this is the case, we could have spared ourselves this work. This is due to our translation to ERTL, which systematically saves the callee-saved registers at the function entry. Additionally, when `x <= 1`, we jump to `L7` to set 1 in `%rax`, then we perform an unconditional jump to `L1` to finish the function. We could have spared ourselves this additional jump, either by duplicating the end of the function or by setting 1 in `%rax` from the start, to then jump directly to `L1`. We also note that the instruction `movq %rbx, %rdi` is useless, as these two registers already contain the same value. Indeed, we copied `%rdi` into `%rbx` three instructions earlier. Here, a fairly simple data flow analysis could determine that the two registers contain the same value and remove this instruction. Finally, we could have used the instruction `decq %rdi` instead of `addq $-1, %rdi` through better instruction selection, just as we could have used `push` and `pop` to save and restore `%rbx`. But it is always easier to optimize *one* program by hand.

Exercise 48. Using *Compiler Explorer* (`godbolt.org`), observe and compare the instruction selections made by `gcc -Og` and `clang -Og` on

- arithmetic, for example

  ```
  int f(int x, int y) { return 5*x+8*y+13; }
  ```

- memory access, for example

  ```
  int f(int x, int *y) { return y[2*x+1]; }
  ```

Solution □

Exercise 49. Observe, explain, and compare the code produced by `gcc -Og` and `clang -Og` on

```
int fib(int n) {
  if (n <= 1) return n;
  return fib(n-2) + fib(n-1);
}
```

Solution □

Bibliographical Notes. The register allocation of the *Iterated Register Coalescing* type is due to George and Appel [9], based on earlier work by Chaitin.

```
fact:   pushq %rbp
        movq  %rsp, %rbp
        addq  $-8, %rsp
        movq  %rbx, -8(%rbp)
        movq  %rdi, %rbx
        cmpq  $1, %rbx
        jle   L7
        movq  %rbx, %rdi      ## useless, too bad
        addq  $-1, %rdi
        call  fact
        imulq %rbx, %rax
L1:     movq  -8(%rbp), %rbx
        movq %rbp, %rsp
        popq %rbp
        ret
L7:     movq  $1, %rax
        jmp   L1
```

Figure 10.12: x86-64 code for the `fact` function.

Appendices

A Solutions to the Exercises

Exercise 1, page 9

It sets the `%rax` register to zero, because for any bit value b, the exclusive OR of b and b is always 0. This is slightly more efficient than `movq $0, %rax`, as the instruction is represented in machine code by three bytes instead of seven.

Exercise 2, page 9

The integer -16 is written as $111\ldots1110000_2$ in two's complement (a sequence of 1s ending with four 0s). Therefore, the instruction `andq $-16, %rsp` clears the four least significant bits of `%rsp`. This can be interpreted arithmetically as rounding down `%rsp` to the nearest multiple of 16.

This is a common way to align the stack before a `call` instruction, especially when it is unknown whether the stack is already aligned. In such cases, additional instructions are needed to restore the stack to its original state. Section 1.3 explains how to use the `%rbp` register for this purpose.

Exercise 3, page 10

1. $y = 13x + 1$

```
leaq    (%rdi,%rdi,2), %rax
leaq    1(%rdi,%rax,4), %rax
```

2. $y = (x + 1)^2$

```
leaq    1(%rdi), %rax
imulq   %rax, %rax
```

3. $y = x \oplus (x - 1)$ (exclusive OR)

```
leaq    -1(%rdi), %rax
xorq    %rdi, %rax
```

Exercise 4, page 10

1. Computes `%rdi` $>$ 42 in `%eax` (and thus in `%rax`).

   Equivalent to the C code `long f(long x) { return x > 42; }`

2. Computes 17 $\times$ `%rdi` $-$ 42 in `%rax`.

   Equivalent to the C code `long f(long x) { return x * 17 - 42; }`

3. Computes `%rsi` or `%rsi` $-$ 3 in `%rax`, depending on whether `%rdi` is zero or not.

   Equivalent to the C code `long f(long x, long y) { if (x) y -= 3; return y; }`

4. Computes the Boolean value 0 $\leq$ `%rdi` in `%rax` (0 or 1).

   Equivalent to the C code `long f(long x) { return 0 <= x; }`

Exercise 5, page 11

This program computes X^N using the fast exponentiation algorithm. Here is the same code with explanations:

```
   movq  $1, %rax    # res <- 1
1: movq  %rax, %rcx  # tmp <- res
   imul  %rdi, %rcx  # tmp <- X * tmp
   imul  %rdi, %rdi  # X   <- X * X
   shr   $1, %rsi    # N   <- N / 2
   cmovb %rcx, %rax  # res <- tmp if CF, i.e., if N was odd
   jnz   1b          # while (the old) N is not 0
```

Note how the flags set by `shr` are used twice: once for the conditional `mov` and once for the conditional branch.

Exercise 6, page 14

The integer `a` represents the columns of the chessboard that remain to be filled. The integers `b` and `c` represent the columns that are under attack by queens already placed on previous rows. The integer `a & ~b & ~c` thus represents the columns that need to be considered for the current row. The loop `while` iterates over the bits set to 1 in this integer, each time extracting the least significant bit set to 1 using `e & -e`. For each column examined, a recursive call is made with `a`, `b`, and `c` updated accordingly. A solution is found when `a` $= 0$.

Note: A justification for the trick `e & -e` can be found in Henry Warren's excellent book *Hacker's Delight* [29]. This book also contains much useful information for someone writing a compiler, such as a systematic method for replacing division by a constant with less expensive operations than division.

Exercise 7, page 16

The value of `n` is in `%rdi` and remains unmodified. We choose to place `c` in `%rax` since it will be the return value. We choose to place `s` in `%rsi`, a caller-saved register. Here is a possible implementation:

```
isqrt:  xorq  %rax, %rax
        movq  $1, %rsi
        jmp   2f
1:      incq  %rax
        leaq  1(%rsi, %rax, 2), %rsi
2:      cmpq  %rdi, %rsi
        jle   1b
        ret
```

The loop test (label 2) is placed *after* the loop body (label 1). Initially, we jump to the test with `jmp 2f`. This approach ensures that only one branch operation is performed per loop iteration.

To compute `isqrt`(17), it is sufficient to write:

```
main:   movq  $17, %rdi
        call  isqrt
```

The result is then in `%rax`. If we want to display it, we can use the library function `printf`, as follows:

```
        movq  $Sprint, %rdi
        movq  %rax, %rsi
        xorq  %rax, %rax      # no floating-point arguments
        call  printf
Sprint:
        .string "isqrt(17) = %d\n"
```

(Since `printf` is a variadic function, we must indicate the number of floating-point arguments in `%rax`; here, there are none.)

Exercise 8, page 16

We start with the version implemented using a loop.

```
fact:   movq    $1, %rax        # r <- 1
1:      imulq   %rdi, %rax      # r <- x * r
        decq    %rdi            # x <- x-1
        jg      1b              # continue if x > 0
        ret
```

Here, we used the fact that the `decq` instruction sets the flags. Therefore, we can perform a conditional branch immediately afterward.

For the recursive version, it is more complex because we need to save the argument on the stack.

```
factrec:cmpq    $1, %rdi        # x <= 1?
        jle     1f
        pushq   %rdi            # save x on the stack
        decq    %rdi            # x <- x-1
        call    factrec         # call fact(x-1)
        popq    %rcx            # restore x
        imulq   %rcx, %rax      # x * fact(x-1)
        ret
1:      movq    $1, %rax
        ret
```

Exercise 9, page 18

This function `f` takes two integer parameters a and b, received in `%rdi` and `%rsi` respectively, and returns an integer result in `%rax`. It computes the greatest common divisor (GCD) of a and b, assumed to be positive or zero, using Euclid's algorithm, in its original form that only performs subtractions, *i.e.*,

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b; else b = b - a;
  return b;
}
```

This assembly code can be seen as a relatively optimized version of this C program.

Exercise 10, page 18

Let's take the following C program as an example, which calculates the sum $1+2+\cdots+n$:

```
  long f (long n) {
    long s = 0;
    while (n > 0) s += n--;
    return s;
  }
```

With `gcc -Og`, we obtain something straightforward:

```
   f:     movl    $0, %eax
          jmp     .L2
  .L3:    addq    %rdi, %rax
          leaq    -1(%rdi), %rdi
  .L2:    testq   %rdi, %rdi
          jg      .L3
          ret
```

Whereas with `gcc -O1` or `gcc -O2`, we get slightly more sophisticated code:

```
f:      xorl    %eax, %eax
        testq   %rdi, %rdi
        jle     .L4
.L3:    addq    %rdi, %rax
        subq    $1, %rdi
        jne     .L3
        ret
.L4:    ret
```

We note in particular:

- that the test is performed once outside the loop;
- then, in each iteration, it takes advantage of the flags set by `sub`.

Now, consider a `do while` loop:

```
long f (long n) {
  long s = 0;
  do s += n; while (--n);
  return s;
}
```

With `gcc -Og` or `gcc -O1`, we obtain something immediate, "identical" to the C code:

```
f:      movl    $0, %eax
.L2:    addq    %rdi, %rax
        subq    $1, %rdi
        jne     .L2
        ret
```

With `gcc -O2`, however, the code becomes more complex (explanations in the margin):

```
f:      xorl    %eax, %eax
        leaq    -1(%rdi), %rdx
        testb   $1, %dil          # tests the parity of n
        je      .L2
        movq    %rdi, %rax        # handles the odd case
        movq    %rdx, %rdi
        testq   %rdx, %rdx        # and tests if n=1
        je      .L11
.L2:
        leaq    -1(%rax,%rdi,2), %rax   # proceeds 2 by 2 by doing
        subq    $2, %rdi                # s += n + n-1; n -= 2
        jne     .L2
        ret
.L11:
        ret
```

Exercise 11, page 23

```
let succ e =
  Add (e, Cte 1)
```

Exercise 12, page 23

```
let rec eval env = function
  | Cte n         -> n
  | Var x         -> env x
  | Add (e1, e2) -> eval env e1 + eval env e2
  | Mul (e1, e2) -> eval env e1 * eval env e2
```

Exercise 13, page 24

The primitive + must be applied to a pair, whereas the function $\texttt{fun}\ x \to \texttt{fun}\ y \to +\ (x,\ y)$ is applied successively to two arguments. More precisely, its application to one argument returns a function, which can then be applied to a second argument. In particular, it can therefore be partially applied to a single argument, unlike +. A function that takes its arguments successively, returning a function each time, is said to be *curried*. The term comes from the name of the mathematician Haskell Curry.

Exercise 14, page 28

For simplicity, let F denote the term $\texttt{fun}\ \mathit{fact} \to \texttt{fun}\ n \to e$, where e is the term

$$\texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1)))$$

The derivation then has the following form:

$$\dfrac{\dfrac{\mathit{opfix} \twoheadrightarrow \mathit{opfix} \quad F \twoheadrightarrow (\texttt{fun}\ \mathit{fact} \to \ldots) \quad \ldots}{(\mathit{opfix}\ F) \twoheadrightarrow (\texttt{fun}\ n \to e')}(\textsc{fix}) \quad 2 \twoheadrightarrow 2 \quad \dfrac{\vdots}{e'[n \leftarrow 2] \twoheadrightarrow 2}}{(\mathit{opfix}\ F)\ 2}(\textsc{app})$$

where e' is the term $e[\mathit{fact} \leftarrow \mathit{opfix}\ F]$. We leave it to the (courageous) reader to complete this derivation.

Exercise 15, page 30

Let e be the expression (2.3), *i.e.*, $\Delta\ \Delta$. Suppose there is a value v for this expression. Then the derivation is necessarily of the form:

$$\dfrac{\dfrac{}{\Delta \twoheadrightarrow \Delta} \quad \dfrac{}{\Delta \twoheadrightarrow \Delta} \quad \dfrac{\vdots}{\Delta[x \leftarrow \Delta] \twoheadrightarrow v}}{\Delta\ \Delta \twoheadrightarrow v}(\textsc{app})$$

because the rule (APP) is the only one that applies. But $\Delta[x \leftarrow \Delta] = (x\ x)[x \leftarrow \Delta] = \Delta\ \Delta$. So the third premise is a derivation of the same conclusion. By initially considering a derivation of minimal height for $e \twoheadrightarrow v$, we obtain a contradiction.

Exercise 16, page 33

For simplicity, let f denote the term

$$\mathit{opfix}\ (\texttt{fun}\ \mathit{fact} \to \texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1)))).$$

We then have the following sequence of reductions:

$$\begin{array}{ll}
 & f\ 2 \\
\to & (\texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 2 \\
\to & \texttt{if}\ {=}(2,0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))) \\
\to & \texttt{if}\ \mathit{false}\ \texttt{then}\ 1\ \texttt{else}\ \times(2,\ f\ (+(2,\ -1))) \\
\to & \times(2,\ f\ (+(2,\ -1))) \\
\to & \times(2,\ (\texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ (+(2,\ -1))) \\
\to & \times(2,\ (\texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 1)) \\
\to & \times(2,\ \texttt{if}\ {=}(1,0)\ \texttt{then}\ 1\ \texttt{else}\ \times(1,\ f\ (+(1,\ -1)))) \\
\to & \times(2,\ \texttt{if}\ \mathit{false}\ \texttt{then}\ 1\ \texttt{else}\ \times(1,\ f\ (+(1,\ -1)))) \\
\to & \times(2,\ \times(1,\ f\ (+(1,\ -1)))) \\
\to & \times(2,\ \times(1,\ (\texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ (+(1,\ -1)))) \\
\to & \times(2,\ \times(1,\ (\texttt{fun}\ n \to \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 0)) \\
\to & \times(2,\ \times(1,\ \texttt{if}\ {=}(0,0)\ \texttt{then}\ 1\ \texttt{else}\ \times(0,\ f\ (+(0,\ -1))))) \\
\to & \times(2,\ \times(1,\ \texttt{if}\ \mathit{true}\ \texttt{then}\ 1\ \texttt{else}\ \times(0,\ f\ (+(0,\ -1))))) \\
\to & \times(2,\ \times(1,\ 1)) \\
\to & \times(2,\ 1) \\
\to & 2
\end{array}$$

Exercise 17, page 38

For *opif*, either add Boolean constants to our abstract syntax, or decide, for example, that *opif* tests equality to zero. In the latter case, we can write:

```
| Op "opif" ->
    let (Pair (Const n1, Pair (Fun (_,bt), Fun (_,be)))) = eval e2 in
    eval (if n1 = 0 then bt else be)
```

For *opfix*, we follow the semantics rule:

```
| Op "opfix" ->
    let Fun (f, e) as b = eval e2 in
    eval (subst e f (App (Op "opfix", b)))
```

Note the use of `as` to avoid reconstructing `Fun (f, e)`.

Exercise 18, page 40

First, it would be necessary to distinguish the names of all variables so that a program like:

```
let x = 1 in
let f = fun y -> x+y in
let x = 2 in
f 0
```

does not overwrite the value `x` = 1 saved in the closure of `f` with the value `x` = 2. It would be sufficient here to rename the second variable. But this is not enough. Indeed, in a program like:

```
let f = fun x -> fun y -> x+y in
let f1 = f 1 in
let f2 = f 2 in
f1 0 + f2 0
```

we have two closures representing the values of `f1` and `f2`, the first saving the value `x` = 1 and the second the value `x` = 2. If the environment stored in the closure were a mutable structure, the construction of the second closure would have a side effect on the first, leading to an incorrect value.

Of course, it would be sufficient to *copy* the environment at the time of the closure construction. But this would precisely be the symptom of a poor choice of structure for the environment. Using a purely applicative structure directly is simpler. Adding and accessing cost a bit more ($O(\log n)$ instead of $O(1)$), but copying is avoided.

Exercise 19, page 42

The rules for big-step semantics are as follows:

$$\frac{}{E, \texttt{skip} \twoheadrightarrow E} \qquad \frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, s_2 \twoheadrightarrow E_2}{E, s_1; s_2 \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \twoheadrightarrow E\{x \mapsto v\}}$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \twoheadrightarrow E_1} \qquad \frac{E, e \twoheadrightarrow \texttt{false} \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while } e \texttt{ do } s \twoheadrightarrow E_2}{E, \texttt{while } e \texttt{ do } s \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while } e \texttt{ do } s \twoheadrightarrow E}$$

Exercise 20, page 51

A simple solution consists of combining three regular expressions for the languages of words containing zero, one, and two characters b, respectively:

$$a\star \mid a\star b a\star \mid a\star b a\star b a\star$$

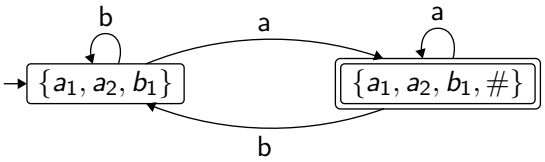
Exercise 21, page 53

The only difference with *first* lies in the concatenation.

$$\begin{aligned}
last(\emptyset) &= \emptyset \\
last(\epsilon) &= \emptyset \\
last(a) &= \{a\} \\
last(r_1\, r_2) &= last(r_1) \cup last(r_2) \quad \text{if } null(r_2) \\
&= last(r_2) \quad \text{otherwise} \\
last(r_1 \,|\, r_2) &= last(r_1) \cup last(r_2) \\
last(r\star) &= last(r)
\end{aligned}$$

Exercise 22, page 54

We distinguish the characters as follows: $(a_1|b_1) \star a_2$. We then obtain the automaton:

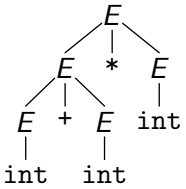


Unlike the automaton on page 51, it is deterministic, meaning that for each state and each character, there is only one possible transition.

Exercise 23, page 67

We show the double inclusion of the languages recognized by grammars (4.1) and (4.2). One direction is easy: every word recognized by grammar (4.2) is recognized by grammar (4.1). Indeed, given a derivation tree, it is sufficient to replace $E \to E{+}T$ with $E \to E{+}E$ and $T \to T{*}F$ with $E \to E{*}E$ and to remove the nodes $E \to T$ and $T \to F$. This can be done rigorously by induction on the size of the derivation tree.

The other direction is more delicate. Indeed, a derivation tree such as:



must be transformed to make the rule $E \to E{+}T$ appear at its root, which is a more subtle change. Let $L(E)$, $L(T)$, and $L(F)$ denote the languages recognized by the three non-terminals of grammar (4.2). We clearly have $L(F) \subseteq L(T) \subseteq L(E)$.

We start by showing that if $w_1, w_2 \in L(T)$, then $w_1{*}w_2 \in L(T)$, by induction on the size of the derivation of w_2. If the derivation starts with $T \to F$, *i.e.*, $w_2 \in L(F)$, it is obvious. Otherwise, the derivation starts with $T \to T{*}F$, *i.e.*, $w_2 \to^\star w_2'{*}f$, and we can apply the induction hypothesis $w_1{*}w_2'$ and then conclude with $T \to T{*}F$.

Similarly, we show that if $w_1 \in L(E)$ and $w_2 \in L(T)$, then w_1`*`$w_2 \in L(E)$, by induction on the size of the derivation of w_1. Then we show that if $w_1, w_2 \in L(E)$, then w_1`*`$w_2 \in L(E)$, by induction on the size of the derivation of w_2. Finally, we show that if $w_1, w_2 \in L(E)$, then w_1`+`$w_2 \in L(E)$, by induction on the size of the derivation of w_2.

We can now show by induction on the size of the derivation tree that a word recognized by grammar (4.1) is recognized by grammar (4.2). If the word is of the form `int`, it is obvious. If the word is of the form `(`E`)`, then we apply the induction hypothesis to the derivation of E and add $E \to T \to F \to$ `(`E`)`. If the derivation is of the form $E \to E$`*`E, then we apply the induction hypothesis to the two sub-derivations and then use the previous result. Similarly, if the derivation is of the form $E \to E$`+`E.

Exercise 24, page 67

This grammar is ambiguous because the word `lam var var` admits two derivation trees, corresponding respectively to (`lam var`) `var` and `lam` (`var var`).

We can recognize the same language with this unambiguous grammar:

$$\begin{array}{rcl} T & \to & A \\ & | & T\ A \\ A & \to & \texttt{nat} \\ & | & \texttt{lam}\ A \end{array}$$

We can prove this by double inclusion, as in the previous exercise.

Exercise 25, page 69

On the one hand, we show by induction on the number of steps in the fixed point computation that if we obtain $\text{NULL}(X) =$ `true`, then indeed $X \to^\star \epsilon$.

On the other hand, we show by induction on the number of steps in the derivation $X \to^\star \epsilon$ that we indeed obtain $\text{NULL}(X) =$ `true` by the fixed point computation.

Exercise 26, page 70

	S	T
$\text{NULL}(X)$	`false`	`false`

	S	T
$\text{FIRST}(X)$	{`(`, `nat`, `lam`}	{`(`, `nat`, `lam`}

	S	T
$\text{FOLLOW}(X)$	{#}	{`(`, `)`, `nat`, `lam`, #}

Exercise 27, page 70

	S	E	L
$\text{NULL}(X)$	`false`	`false`	`true`

	S	E	L
$\text{FIRST}(X)$	{`(`, `sym`}	{`(`, `sym`}	{`(`, `sym`}

	S	E	L
$\text{FOLLOW}(X)$	{#}	{`(`, `)`, `sym`, #}	{`)`}

Exercise 29, page 72

We calculated NULL, FIRST, and FOLLOW for this grammar in exercise 26. We deduce the following expansion table:

	`nat`	`lam`	`(`	`)`	#
S	$T\#$	$T\#$	$T\#$		
T	`nat`	`lam` T	`(` TT `)`		

This grammar is therefore LL(1).

Exercise 30, page 72

We calculated NULL, FIRST, and FOLLOW for this grammar in exercise 27. We deduce the following expansion table:

	`sym`	`(`	`)`	#
S	$E\#$	$E\#$		
E	`sym`	`(` L `)`		
L	EL	EL	ϵ	

The grammar of LISP is therefore LL(1).

Exercise 31, page 77

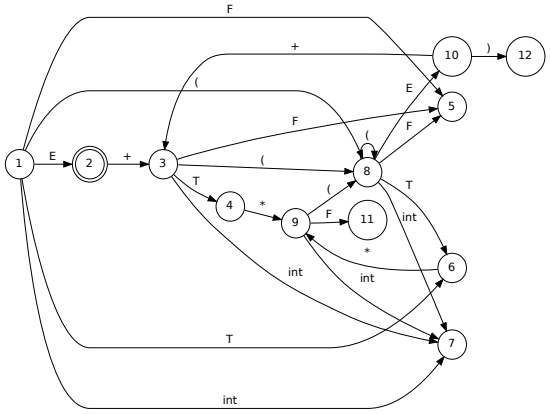
We obtain an automaton with 9 states and the following tables:

	action					*goto*
state	`nat`	`lam`	`(`	`)`	#	T
0	`shift` 2	`shift` 4	`shift` 3			1
1					success	
2	`reduce` $T \to$ `nat`					
3	`shift` 2	`shift` 4	`shift` 3			6
4	`shift` 2	`shift` 4	`shift` 3			5
5	`reduce` $T \to$ `lam` T					
6	`shift` 2	`shift` 4	`shift` 3			7
7				`shift` 8		
8	`reduce` $T \to$ `(` T T `)`					

This grammar is therefore LR(0).

Exercise 32, page 77

The automaton is as follows:



Exercise 33, page 77

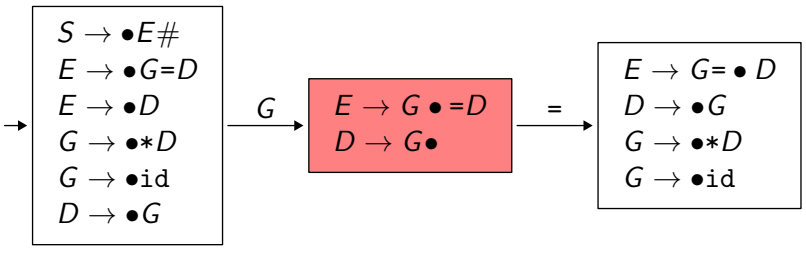
We obtain an automaton with 8 states and the following tables:

	action				*action*	
state	`sym`	`(`	`)`	#	E	L
0	`shift` 2	`shift` 3			1	
1				success		
2	`reduce` $E \to$ `sym`					
3	`shift` 2	`shift` 3	`reduce` $L \to \epsilon$		6	4
4			`shift` 5			
5	`reduce` $E \to$ `(`L`)`					
6	`shift` 2	`shift` 3	`reduce` $L \to \epsilon$		6	7
7			`reduce` $L \to EL$			

This grammar is therefore SLR(1).

Exercise 34, page 78

The automaton contains, in particular, the following three states:



In the second (in red), there is a shift/reduce conflict when reading the character =.

	=	
1	...	...
2	`shift` 3 `reduce` $D \to G$	...
3	⋮	⋱

The character = is indeed part of the followers of G, and therefore the grammar is not SLR(1).

Exercise 35, page 78

The reduction $D \to G$ is now only performed for the character # and the conflict disappears:

	#	=	
1	...	...	...
2	reduce $D \to G$	shift 3	...
3	⋮	⋮	⋱

Exercise 36, page 81

The conflict corresponds to an input of the form:

IF CONST THEN IF CONST THEN CONST ELSE CONST

at the time of reading the ELSE. Indeed, one can then reduce IF CONST THEN CONST, which would correspond to:

IF CONST THEN (IF CONST THEN CONST) ELSE CONST

or, on the contrary, read ELSE, which would correspond to:

IF CONST THEN (IF CONST THEN CONST ELSE CONST)

This conflict, present in the grammar of many programming languages, is known as the *dangling else*. It is generally resolved in favor of the shift, to associate the ELSE with the closest THEN. Since the priority of the rule:

$$E \to \texttt{IF}\ E\ \texttt{THEN}\ E$$

is that of THEN, it is sufficient to give ELSE a higher priority than that of THEN, i.e.:

```
%nonassoc THEN
%nonassoc ELSE
```

Exercise 37, page 85

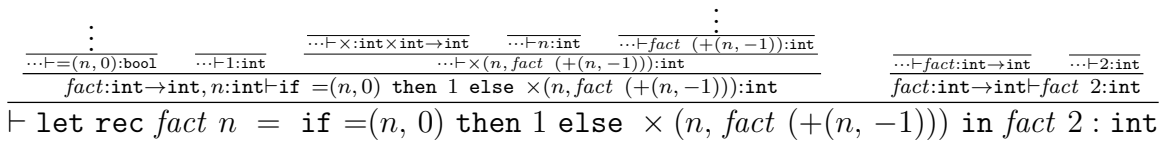
It is sufficient to give the first occurrence of fun $x \to x$ the type:

$$(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

and the second occurrence the type:

$$(\texttt{int} \to \texttt{int}).$$

Exercise 38, page 86



Exercise 39, page 86

We define the following types for the types and expressions of Mini-ML.

```
type typ =
  | Tint
  | Tarrow   of typ * typ
  | Tproduct of typ * typ
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

We can easily implement the environment Γ with the OCaml `Map` module:

```
module Smap = Map.Make(String)
type env = typ Smap.t
```

This is a persistent structure, which will be useful later. Moreover, it is a balanced tree, and we therefore have an insertion and a search in $O(\log n)$ in an environment containing n entries, which is quite acceptable. Let's now write the function that computes the type of an expression. Some cases are immediate:

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) -> Tproduct (type_expr env e1, type_expr env e2)
```

For the function, the type of the variable is given:

```
  | Fun (x, ty, e) ->
      Tarrow (ty, type_expr (Smap.add x ty env) e)
```

For the local variable, it is computed:

```
  | Let (x, e1, e2) ->
      type_expr (Smap.add x (type_expr env e1) env) e2
```

We note here the interest of the persistence of `env`: we add to `env`, obtaining a new structure, and it is never necessary to remove anything from `env`. Finally, the only checks are in the application:

```
  | App (e1, e2) -> begin match type_expr env e1 with
      | Tarrow (ty2, ty) ->
          if type_expr env e2 = ty2 then ty
          else failwith "error: argument of wrong type"
      | _ ->
          failwith "error: function expected"
    end
```

In practice, we make an effort to provide better error messages, more precise (for example by displaying the found and expected types) and located (when the abstract syntax trees are).

Exercise 40, page 90

We can give `fun` $x \to x\ x$ the type $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$, as follows:

$$\dfrac{\dfrac{\overline{x : \forall\alpha.\alpha \to \alpha \vdash x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)} \qquad \overline{x : \forall\alpha.\alpha \to \alpha \vdash x : \forall\alpha.\alpha \to \alpha}}{x : \forall\alpha.\alpha \to \alpha \vdash x\ x : \forall\alpha.\alpha \to \alpha}}{\emptyset \vdash \texttt{fun}\ x \to x\ x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)}$$

It is therefore possible to type Δ in System F. On the other hand, it is not possible to type the expression $\Delta\ \Delta$, nor the definition of *opfix* on page 31.

Exercise 41, page 97

The type of `map_id` is not generalized, because it is an application.

```
  val map_id : '_a list -> '_a list = <fun>
```

As a result, the first application of `map_id` resolves `'_a` as being `int` and the second application fails. A solution consists in writing `map_id` rather in the form:

```
  let map_id l = List.map (fun x -> x) l
```

(We speak of η-expansion when we write `fun x -> f x` rather than `f`.) As a result, the type of `map_id` is now generalized because it is a value, in this case an abstraction.

Exercise 42, page 123

For `f`, the argument is an integer (passed in `%rdi`):

```
f(int): leal    -1(%rdi), %eax
        ret
```

We observe that `%rdi` is not modified, because there is no need to do so: `x` is passed by value, so its modification is not observable to the caller. That said, the compiler *could* modify `%rdi`, for example by decrementing it before copying it to `%rax`, because it is a caller-saved register.

For `g`, the argument is now a pointer (passed in `%rdi`) to an integer:

```
g(int&): movl    (%rdi), %eax
         subl    $1, %eax
         movl    %eax, (%rdi)
         ret
```

Here, the effect on `x` is observable because it is the *memory* that is modified.

Now let's add a function `h` containing a call to `g`:

```
int h(int v) { int s = g(v); return s; }
```

We get the following code:

```
h(int): subq    $8, %rsp
        movl    %edi, 4(%rsp)
        leaq    4(%rsp), %rdi
        call    g(int&)
        addq    $8, %rsp
        ret
```

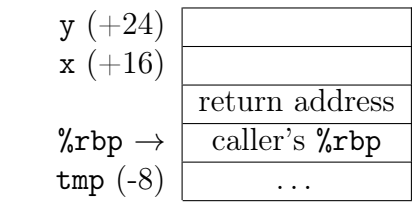
We observe that:

- the variable `v` is allocated on the stack: 8 bytes are reserved, and the top 4 receive the value of the parameter `v` (32-bit integer);
- the address `%rsp` + 4 is passed as an argument to the function `g`.

Exercise 43, page 130

Below is the code obtained with our small compiler in the middle, and a more straightforward version on the right, where we do not bother to position `%rbp`, among other things.

```
void swap(int &x, int &y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```




```
swap:
  pushq %rbp
  movq %rsp, %rbp
  subq $8, %rsp
  movq 16(%rbp), %rdi
  movq (%rdi), %rdi
  leaq -8(%rbp), %rsi
  movq %rdi, 0(%rsi)
  movq 24(%rbp), %rdi
  movq (%rdi), %rdi
  movq 16(%rbp), %rsi
  movq %rdi, 0(%rsi)
  movq -8(%rbp), %rdi
  movq 24(%rbp), %rsi
  movq %rdi, 0(%rsi)
  movq %rbp, %rsp
  popq %rbp
  ret
```

```
swap:
  movq 8(%rsp), %rdi
  movq 16(%rsp), %rsi
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

In a more realistic version, the parameters `x` and `y` would be passed in registers (respectively `%rdi` and `%rsi`), and the code is even simpler:

```
swap:   movq  (%rdi), %rax
        movq  (%rsi), %rdx
        movq  %rdx, (%rdi)
        movq  %rax, (%rsi)
        ret
```

(This is the code obtained with `g++ -O2`.)

Exercise 44, page 139

It is sufficient to manually construct closures. This can even be done with an ad hoc type:

```
enum kind { Kid, Kleft, Kright };

struct Kont {
  enum kind kind;
  union { struct Node *r; int hl; };
  struct Kont *kont;
};
```

And a function to apply it:

```
int apply(struct Kont *k, int v) { ... }
```

This is called *defunctionalization* (Reynolds, 1972).

Exercise 45, page 139

1. It is sufficient to sort the smaller of the two portions first:

```
void quicksort(int a[], int l, int r) {
  ...
  if (lo - l < r - hi) {
    quicksort(a, l, lo);
    quicksort(a, hi, r); // tail call
  } else {
    quicksort(a, hi, r);
    quicksort(a, l, lo); // tail call
  }
}
```

2. It is sufficient to replace the second call with a loop:

```
void quicksort(int a[], int l, int r) {
  while (r - l > 1) {
    ...
    if (lo - l < r - hi) {
      quicksort(a, l, lo);
      l = hi;
    } else {
      quicksort(a, hi, r);
      r = lo;
    }
  }
}
```

   In other words, we manually perform tail call optimization here.

Exercise 46, page 150

The method call `g1.move(10, 5)` is compiled as follows:

```
        movq %r12, %rdi      # g1.move(10, 5)
        movq $10, %rsi
        movq $5, %rdx
        movq (%rdi), %rcx
        call *8(%rcx)
```

Similarly for `g1.draw()`.

```
        movq %r12, %rdi      # g1.draw()
        movq (%rdi), %rcx
        call *16(%rcx)
```

We compile `new Circle` as we did for `new Rectangle`, assuming here that `g2` is allocated in `%r13`.

```
        movq $48, %rdi # %r13 = g2 = new Circle(10,10,2)
        call malloc
        movq %rax, %r13
        movq $descr_Circle, (%r13)
        movq %r13, %rdi
        movq $10, %rsi
        movq $10, %rdx
        movq $2, %rcx
        call new_Circle
```

The rest does not pose any additional difficulty.

```
        movq $16, %rdi      # %r14 = g3 = new Group()
        call malloc
        movq %rax, %r14
        movq $descr_Group, (%r14)
        movq %r14, %rdi
        call new_Group

        movq %r14, %rdi      # g3.add(g1)
        movq %r12, %rsi
        movq (%rdi), %rcx
        call *24(%rcx)
        ...
```

Exercise 47, page 151

Let's compile the construct e `instanceof` C assuming the value of e is in `%rdi` and the descriptor of C is in `%rsi`. We write a loop that traverses the class hierarchy. We know we have reached `Object` when the superclass descriptor is null.

```
instanceof:
        movq    (%rdi), %rdi
L:      cmpq    %rdi, %rsi      # same descriptor?
        je      Ltrue
        movq    (%rdi), %rdi    # move to the superclass
        testq   %rdi, %rdi
        jnz     L               # have we reached Object?
Lfalse: movq    $0, %rax
        ret
Ltrue:  movq    $1, %rax
        ret
```

We made sure to test for equality before testing if the superclass exists, so that e `instanceof Object` correctly returns `true`.

Exercise 48, page 181

For code such as

```
int f(int x, int y) { return 5*x+8*y+13; }
```

we note a small difference (`lea` vs `add`):

gcc

```
leal    (%rdi,%rdi,4), %eax
leal    13(%rax,%rsi,8), %eax
```

clang

```
leal    (%rdi,%rdi,4), %eax
leal    (%rax,%rsi,8), %eax
addl    $13, %eax
```

For memory access, with code such as

```
int f(int x, int *y) { return y[2*x+1]; }
```

we again note a difference, still related to the use or non-use of a constant at the `lea` level:

gcc

```
addl    %edi, %edi
movslq  %edi, %rdi
movl    4(%rsi,%rdi,4), %eax
```

clang

```
leal    1(,%rdi,2), %eax
cltq
movl    (%rsi,%rax,4), %eax
```

Exercise 49, page 181

With `gcc`, we obtain the following code:

```
fib:
        pushq   %rbp                        leal    -1(%rbx), %edi
        pushq   %rbx                        call    fib
        subq    $8, %rsp                    addl    %ebp, %eax
        movl    %edi, %ebx          .L3:    addq    $8, %rsp
        cmpl    $1, %edi                    popq    %rbx
        jle     .L5                         popq    %rbp
        leal    -2(%rdi), %edi              ret
        call    fib                 .L5:    movl    %edi, %eax
        movl    %eax, %ebp                  jmp     .L3
```

where we see the use of `%rbp` and `%rbx` to save `n` and the result of `fib(n-2)`, respectively. This is very close to what we obtained with our compiler.

With `clang`, we obtain a very different code:

```
fib:
        pushq   %rbp                        decl    %ebx
        pushq   %rbx                        cmpl    $2, %ebx
        pushq   %rax                        jge     .LBB1_2
        movl    %edi, %ebx          .LBB1_3:addl    %ebp, %ebx
        xorl    %ebp, %ebp                  movl    %ebx, %eax
        cmpl    $2, %ebx                    addq    $8, %rsp
        jl      .LBB1_3                     popq    %rbx
.LBB1_2:leal    -2(%rbx), %edi              popq    %rbp
        call    fib                         ret
        addl    %eax, %ebp
```

There is still the idea of using two callee-saved registers, but the second call to `fib` is compiled *as a loop*, with the sum being done in `%rbp` (which is initially set to zero).

This is a form of tail call optimization (see section 8.2), but it is starting to get quite subtle here (since `fib(n-1)` is not technically a tail call, given that there is still an addition to be done afterward).

B

Concise French-English Glossary of Compilation Terms

French	English	see pages
affectation	assignment	
appel	call	114
par valeur	by value	
par référence	by reference	
par nom	by name	
par nécessité	by need	
appel terminal	tail call	137
appelant	caller	11
appelé	callee	11
assembleur (langage)	assembly language	5
assembleur (outil)	assembler	5
compilateur	compiler	
décalage	shift	
drapeau	flag	9
filtrage	pattern-matching	140
grammaire	grammar	
interprète	interpreter	37
langage source	source language	
langage cible	target language	
mémoire	memory	
octet	byte	3
pile d'appels	call stack	11
redéfinition	overriding	102
registre	register	
règles d'inférence	inference rules	26
sémantique	semantics	23
dénotationnelle	denotational semantics	
opérationnelle	operational semantics	
à petits pas	small steps semantics	

French	English	see pages
à grands pas	big steps semantics	
sucre syntaxique	syntactic sugar	23
syntaxe abstraite	abstract syntax	22
tableau d'activation	stack frame	12
tas	heap	
valeur gauche	left value	113
variable libre	free variable	25

Bibliography

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils*. Pearson, 2007.

[2] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011.

[3] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117 – 126, 1986.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2015. http://csapp.cs.cmu.edu/.

[5] Olivier Carton. *Langages formels, Calculabilité et Complexité*. Éditions Vuibert, 2014.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[7] Jean-Christophe Filliâtre. Mesurer la hauteur d'un arbre. In Zaynah Dargaye and Yann Régis-Gianas, editors, *Trente-et-unièmes Journées Francophones des Langages Applicatifs*, Gruissan, France, January 2020. https://usr.lmf.cnrs.fr/~jcf/hauteur/.

[8] Jacques Garrigue. *Relaxing the Value Restriction*, pages 196–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[12] G. Kahn, D. Clement, J. Despeyroux, T. Despeyroux, and L. Hascoet. Natural semantics on the computer. Technical Report R.G. 4-85, Greco de Programmation, Université de Bordeaux, June 1985.

[13] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[14] D.-M. Kernighan and B.-W. Ritchie. *Le langage C ANSI (2ème édition)*. Masson, 1990.

[15] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[16] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308, 1964.

[17] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[18] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] Xavier Leroy. *Control structures in programming languages: from goto to algebraic effects*. Cambridge University Press, 2026. https://xavierleroy.org/control-structures/.

[20] John R. Levine. *Linkers and loaders*. Morgan Kaufmann, 2000.

[21] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17, May 2007.

[22] Luc Maranget. Compiling pattern matching to good decision trees. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, New York, NY, USA, 2008. ACM.

[23] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[24] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[25] François Pottier. Menhir, a LR(1) parser generator for OCaml. http://gallium.inria.fr/~fpottier/menhir/.

[26] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[27] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[28] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[29] Henry S. Warren. *Hacker's Delight.* Addison-Wesley, 2003.

[30] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111 – 156, 1999.

[31] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

Index