Friendly F# - fun with logic.

Dr. Giuseppe Maggiore
Dr. Giulia Costantini
Francesco Di Giacomo
Gerard van Kruiningen

July 3, 2015

Contents

Chapter 1

Introduction

The goal of this document is to make you acquainted with the concept of inference systems. You will learn various aspects of logic and inference system, which we will use to give rise to definitions of concepts such as numbers, boolean expressions, etc. As an advanced example you will also learn how to model programming languages constructs into inference systems. This leads to an important end result: you will learn the *what* of programming languages, in contrast to the *how* that you have studied so far. As a short conclusion we will discuss the strong relationship between logic and programming, and draw some conclusions about computability.

1.0.1 Why bother?

Try drawing a circle or a similar shape. The circle starts out as a single "piece". Choose two points along the border and draw a connecting line between the two: the circle is now split into two pieces. Chose now a third point and connect it to the other two points: the circle is now split into four pieces. A clear pattern is emerging: every point we add, we get twice as many pieces. This suggests that by `k` points we get 2^{k-1} pieces.

This seems all quite logical, but there is a catch. Try now doing this with six points along the border of the circle: instead of the expected 32 pieces, we will end up with 31. This means that the original conclusion was definitely not correct.

What is the moral of this story? If you want (or need) to be absolutely sure about some assertion (often called *proposition* or *hypothesis*), then you cannot trust a few examples and an intuitively appealing story to show that some pattern will always hold true. In the practice, and certainly in that of informatics or medical technology, there are many situations where a few tests and an intuition of why the system works are sufficient. Would you dare fly in an airplane where the programmer concluded after a dozen of tests that "it all works, send it to production"? Would you dare step into an MRI machine (which throws radioactive isotopes at your body) knowing that the software was quickly written during a weekend with a bit of unit testing? Of course not[1].

The use of logic as a foundation for programming does not automatically solve all our problems. It is very well possible (we will get there) that there is no single mathematical formalism or piece of software that will entirely remove the need for human intellect when writing software. **In any case, the use of logic and formality when decomposing problems can greatly help in writing software that is better thought out and over which it is far easier to reason and prove properties.**

1.0.2 Structure of the document

In the rest of this document we will: *i*) introduce informal logical reasoning in Section 1.1; *ii*) introduce a more formal logical formalism in Section 1.2; *iii*) give a first example of logic in action to define basic boolean operators in Section 1.3; *iv*) use logic to define *apparently atomic* and unrelated concepts such as numbers in Chapter 2; *v*) construct complex data structures such as lists, binary search trees, and even balanced binary search trees in Chapter 3; *vi*) build a small programming language in Chapter 4; *vii*) we conclude with a short and woefully incomplete presentation of fragments of computability in Chapter 5.

[1] as a side note, if you are comfortable with these scenarios, then perhaps the motivation for the rest of the document will feel quite weak, so you might better go back to your vacation in Chernobyl or flying in planes held together by duct tape.

1.1 An informal introduction to logic

The basic rules of logic are very simple. Logica is entirely defined as a way to manipulate symbols. These symbols are usually common signs such as numbers, letter, Greek letters, etc., but it is not forbidden to use any symbol we find useful or interesting. In this section, precisely with the goal of illustrating the *independence of logic from the symbols chosen*, we shall manipulate smileys and other icons. **The first step is thus choosing the symbols.** Logic is really the description of most (complex) dynamic processes that search for an answer in a complex space. This search may be a purely mathematical search, or it could be a program running. The dynamic process is defined in terms of a series of rules which are activated whenever we find a matching input. Thus, **the second step is choosing the rules that define our dynamic processes**.

After defining symbols and rules, things become interesting and we can try to use our system to find answers and process information. We usually start with a given proposition, which is the input of the whole process. A *proposition* is just a series of symbols, such as for example: `3 + 2 = 5 of` 🙂 🙂 🕯 🕯. We apply the rules to the proposition until we reach the desired answer, and we have also answered all of the intermediate questions that came to existance during the dynamic process.

A concrete example Let us consider a full example. Consider the symbols of our language to be:

- A smiley 🙂
- A candle 🕯
- A tree 🌳
- A coffee cup ☕

We consider a proposition to be true if and only if we can process it until we reach a ☕. [2] Our rules are:

[2]Given the extreme importance of coffee in the diet of mathematicians and computer scientists, equating coffee with truth does not really seem that illogical

- **(G0)** A ☕ means we are done
- **(R1)** Two 🙂 followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer
- **(R2)** Two 🕯 followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer

Consider now the input proposition of
🙂 🙂 🌳 🕯 🕯 🌳 🙂 🙂 🌳 ☕

We begin by using **R2**, therefore obtaining:
🕯 🕯 🌳 🙂 🙂 🌳 ☕

We then use rule **R1**, therefore obtaining:
🙂 🙂 🌳 ☕

We then use rule **R2**, therefore obtaining:
☕

Now according to **G0** we are done. The proof is successfull, therefore we can conclude that 🙂 🙂 🌳 🕯 🕯 🌳 🙂 🙂 🌳 ☕ was **true within our system of rules**.

Consider now the new input proposition of
🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕

We begin by using **R2**, therefore obtaining:
🙂 🙂 🌳 🕯 🌳 ☕

We then use rule **R1**, therefore obtaining:
🕯 🌳 ☕

Unfortunately now we cannot apply rule **R1** again, because we have no 🙂 at the head of our proposition; we cannot apply rule **R2** because we have no 🕯 at the head of our proposition; and we can certainly not apply rule **G0** because there is no lonely ☕. If we cannot

a step

apply any of our rules, the process is *stuck*. This means that we cannot prove 🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕ with our rules, thus 🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕ was **not true within our system of rules**.

1.2 Inference systems

In this section we present a slightly more formal treatment of inference systems. Moreover, we give a clearer syntax for expressing operators and rules. The formalism which we will use, which is also a programming language, is *Meta-Casanova*. Meta-Casanova falls under the broader category of *logic programming languages*, but given its structure and some of its applications it can also be considered a *meta-compiler*.

1.2.1 Syntax for symbol definitions

Defining symbols in Meta-Casanova requires drawing a distinction: some symbols are considered *data*, that is they are just a way to structure information, while other symbols are considered *functions*, that is they are a way to define transformations from propositions to propositions.

Data symbols Data symbols are defined with the following syntax:

```
Data [α1] "α2" [α3] Priority α4 Type α5
```

The α_i's are the arguments of the symbols:

α_1 are the arguments that come left of the symbol

α_2 is the name of the symbol

α_3 are the arguments that come right of the symbol

α_4 is the priority of the symbol

α_5 is the type of the symbol

Examples of such a definition could be:

```
Data [] "TRUE" [] Priority 10000 Type Expr
Data [Expr] "|" [Expr] Priority 10 Type Expr
```

the above come from the boolean expressions sample, which will be discussed in depth in a further chapter. The first line defines the `TRUE` symbols, which has no arguments left or right (it is just a value), has a very high priority (which does not matter given that this symbol has no parameters), and which is of type `Value`. The second line defines an *or*-style operator that takes an `Expr` to the left, an `Expr` to the right, has a priority of 10, and is an expression itself.

With the definitions above we can write a proposition such as `TRUE | TRUE`, which would be perfectly valid. Associativity is by default to the right, thus `TRUE | TRUE | TRUE` is equivalent to `TRUE | (TRUE | TRUE)`. We can modify the default associativity by adding a last parameter of:

```
Data [Expr] "|" [Expr] Priority 10 Type Expr Associativity Left
```

We might even go as fast as wanting to re-define `TRUE` to a new type, because indeed we could say that `TRUE` is more a value than an expression:

```
Data [] "TRUE" [] Priority 10000 Type Value
```

With the above definition it becomes impossible to write expressions such as `TRUE | TRUE`: the compiler will refuse this as invalid code, because `|` expects arguments of type `Expr` but it is instead given arguments of type `Value`. We can say:

```
Value is Expr
```

to inform the compiler that whenever an `Expr` is expected, then we can also give a `Value` without issues. This can be considered a form of *inheritance* such as that found and used in object-oriented languages.

Function symbols Function symbols are defined with the following syntax:

```
Func [α1] "α2" [α3] Priority α4 Type α5 => α6
```

These symbols are defined almost identically to data symbols, with the difference that they also specify what is the type (α_6) of the data they will return as a result of transformation.

An example of such a definition could be a function that, given a boolean expression, return the corresponding value:

```
Func [] "eval" [Expr] Priority 1 Type Expr => Value
```

It will now be possible to write `eval (TRUE | TRUE)`. Of course so far we have only defined the aacceptable *shape* of terms, but we have not yet said anything about how computations happen.

1.2.2 Syntax and semantics of rules

Rules define how propositions that begin with a *functions* process the *data* that they have as parameters. The rules of an inference system are syntactically quite simple. The simplicity comes from the fact that a rule syntax is only made up of two operators and two additional concepts. The two operators are:

- the **horizontal bar** `------------`
- the **arrow** `=>`

Both horizontal bar and implication arrow can be read out loud as "therefore", as they both capture a form of implication. The horizontal bar separates the main implication from its premises, and thus operates at a higher level of precedence and abstraction with respect to the arrow. A single inference rule will consist of a series of **premises** which are separated from the **main proposition** by the horizontal bar:

```
PREMISE 1
PREMISE 2
...
PREMISE N
-----------------
MAIN PROPOSITION
```

Both individual premises and the main proposition are defined as the **input** sequence of symbols, separated from the **output sequence of symbols** by the implication arrow:

```
INPUT SYMBOLS => OUTPUT SYMBOLS
```

A pattern consists of a series of keywords, operators, and variables, recursively **applied to each other**. Applied in this case is meant in the sense of function application, as in `f(3)` where `f` is applied to three, or `3+x`, where `+` is applied to symbol `3` and variable `x`.

For example, with reference to the example above, we might want to define a rule that has no premises as:

```
------------------------
eval (TRUE | x) => TRUE
```

The rule above tells us that whenever we encounter `eval (TRUE | x)` for some `x` which is left unspecified, then we can directly return `TRUE`.

A rule with premises, which therefore shall require further nested computation, could be:

```
eval x => TRUE
------------------------
eval (FALSE | x) => TRUE
```

The rule above tells us that whenever we encounter `eval (FALSE | x)` for some `x` which is left unspecified, then we need to evaluate premise `eval x => TRUE`; this premise begins by evaluating `eval x`, and if it returns `TRUE` then we proceed to return `TRUE` as the final result. If the premise returns something else than `TRUE`, then execution of the rule is interrupted (it is indeed a failure) and no result is returned. To ensure termination we could add another complementary rule:

```
eval x => FALSE
------------------------
eval (FALSE | x) => FALSE
```

Of course we could merge the two rules above into a single rule so that the result of the premise is directly propagated:

```
eval x => y
-----------------------
eval (FALSE | x) => y
```

Main input The overall input sequence of symbols of the inference system, which can be considered as the *main question* to answer or the *main program* to execute, is just a sequence of symbols without variables such as `eval TRUE` or `eval (TRUE | TRUE)`. Meta-Casanova

will return a list of the sequences of symbols that are obtained as a result of applying the rules of the system to the input; the list will either be:

- empty if no rule could be applied at some point; this means that the input is *not supported* by our rules
- one or more results if the input can be processed without issues
- undefined if the program keeps looping forever

Note that the possiblity to return multiple results is very important to support algorithms such as *backtracking* or similar searches, where multiple possible branches of derivation are followed simultaneously to see how many of them will reach an answer. This specific aspect means that the language behaviour can be considered as a sort of path-finder

1.3 A first example: boolean expressions

The first full-blown logic program that we see is, quite fittingly[3], a program for the definition and evaluation of boolean expressions.

Symbols The basic symbols that we need represents the boolean values of `TRUE` and `FALSE`, and they both have type `Value`:

```
Data [] "TRUE" [] Priority 10000 Type Value
Data [] "FALSE" [] Priority 10000 Type Value
```

Boolean expressions, all of type `Expr`, are: *i*) traditional negation (the unary operator `!` following the tradition of C-like languages); *ii*) disjunction of boolean expressions (commonly known as "or"); *iii*) conjunction of boolean expressions (commonly known as "and").

```
Data [] "!" [Expr] Priority 30 Type Expr
Data [Expr] "|" [Expr] Priority 10 Type Expr
Data [Expr] "&" [Expr] Priority 20 Type Expr
```

[3]Boolean logic is an interpretation of logic itself, which fits snugly within our own implementation of an interpretation of logic. Recursion can be quite fun, as we will see in one of the last chapters.

Of course we want to be able to use values as expressions, otherwise writing `TRUE & TRUE` would not be allowed. For this reason we specify that anything that has type `Value` can also be used where a type `Expr` is expected:

```
Value is Expr
```

In order to compute the `Value` of an `Expr`, we define the `eval` function.

```
Func [] "eval" [Expr] Priority 1 Type Expr => Value
```

Rules The only rules that we can define involve the `eval` function, which is the only function that we have defined. The first two rules trivially specify that when we reach the evaluation of `TRUE` or `FALSE`, then we do not need to further proceed:

```
------------------- (G0)
eval TRUE => TRUE

--------------------- (G1)
eval FALSE => FALSE
```

If we reach the evaluation of the negation of some expression `a`, then we will evaluate `a`. If the evaluation of `a` returns `TRUE`, then the evaluation of the negation of `a` returns `FALSE`:

```
eval a => TRUE
------------------ (NEG0)
eval !a => FALSE
```

Similarly, if we reach the evaluation of the negation of some expression `a`, and evaluation of `a` returns `FALSE`, then the evaluation of the negation of `a` returns `TRUE`:

```
eval a => FALSE
----------------- (NEG1)
eval !a => TRUE
```

When evaluating the disjunction of two expressions `a` and `b`, we try to evaluate `a`: *i*) if `a` evaluates to `TRUE`, then there is no need to further evaluate `b` and we can directly return `TRUE`; *ii*) if `a` evaluates to `FALSE`, then we evaluate `b` and return whatever result of its evaluation.

```
eval a => TRUE
------------------- (OR0)
```

```
eval (a|b) => TRUE

eval a => FALSE
eval b => y
----------------- (OR1)
eval (a|b) => y
```

When evaluating the conjunction of two expressions `a` and `b`, we try to evaluate `a`: *i*) if `a` evaluates to `FALSE`, then there is no need to further evaluate `b` and we can directly return `FALSE`; *ii*) if `a` evaluates to `TRUE`, then we evaluate `b` and return whatever result of its evaluation.

```
eval a => FALSE
-------------------- (AND0)
eval (a&b) => FALSE

eval a => TRUE
eval b => y
---------------- (AND1)
eval (a&b) => y
```

Example run Consider now the evaluation of an expression such as `eval (FALSE | !(TRUE & FALSE))`. We begin with

```
-----------------------------------
eval (FALSE | !(TRUE & FALSE)) => ?
```

This expression is an instance of `eval (a|b)`, therefore we investigate the first premise according to rule `OR0` (notice the question marks at the end of some lines, which mean that we are verifying that this is indeed the case):

```
eval FALSE => TRUE?
---------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE?
```

Since `eval FALSE` does never return `TRUE`, this branch of execution is interrupted. We try the alternate rule `OR1` for `eval (a|b)`, therefore we investigate two premises:

```
eval FALSE => FALSE?
eval !(TRUE & FALSE) => y
-----------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
```

The first premise, `eval FALSE => FALSE`, is trivially verified by the rule `G1`. We may thus delete it from the tree of derivation:

```
eval !(TRUE & FALSE) => y
-----------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
```

The second premise is itself complex, so we need to find its result and then assign it to `y`. We begin with rule `NEG1`[4], which expects the input to evaluate to `FALSE`. In this case, since we know from `NEG1` that `eval !(TRUE & FALSE)` would return `TRUE`, we substitute `y` with `TRUE` speculatively:

```
eval (TRUE & FALSE) => FALSE?
-----------------------------
eval !(TRUE & FALSE) => TRUE
-------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

We now need to evaluate premise `eval (TRUE & FALSE)`, and we do so with application of rule `AND1`[5]. Rule `AND1` evaluates the second term and returns its result:

```
eval TRUE => TRUE?
eval FALSE => y
------------------------------
eval (TRUE & FALSE) => y
y == FALSE?
-----------------------------
eval !(TRUE & FALSE) => TRUE
-------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

Fortunately, `eval TRUE => TRUE` is trivially verified by rule `G0`; we can safely remove it from our tree of derivation:

```
eval FALSE => y
------------------------------
eval (TRUE & FALSE) => y
y == FALSE?
-----------------------------
eval !(TRUE & FALSE) => TRUE
-------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

`eval FALSE` returns `FALSE` as an immediate consequence of rule `G1`. Therefore, we can replace `y` with `FALSE`:

[4]Note that Meta-Casanova would actually first try `NEG0`, which fails, but for reasons of space we skip that lengthy and fruitless derivation.

[5]Again, `AND1` is chosen ad-hoc to go directly to the result.

```
eval FALSE => FALSE
------------------------------
eval (TRUE & FALSE) => FALSE
FALSE == FALSE?
------------------------------
eval !(TRUE & FALSE) => TRUE
--------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

The two upper premises are now completely evaluated, so we can safely remove them from the derivation tree:

```
FALSE == FALSE?
------------------------------
eval !(TRUE & FALSE) => TRUE
--------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

We now need to verify that the result of the last evaluation of `AND1` is compatible with the evaluation of `NEG1`, that is `FALSE == FALSE`. This is trivially the case, and therefore we can discharg the last two premises:

```
eval (FALSE | !(TRUE & FALSE)) => TRUE
```

Since we have no more premises, we can safely conclude that indeed the result of the evaluation of the original proposition `eval (FALSE | !(TRUE & FALSE))` yields `TRUE`, which is also what we would expect from intuition.

Chapter 2

Numbers

In order to investigate the power of a logical framework such as the one we are working with, it may be interesting to see what we can do about the lack of direct support for numbers in the language. At a first glance, there is no relationship between our ability to define symbols and rules, and for example natural numbers, and as such we may be tempted to just "add primitive support" to manipulate such quantities directly.

Interestingly enough, natural numbers are not strictly needed as a primitive concept. Even though for practical reasons[1] a modern programming language should always offer the possibility of directly interfacing with machine integers, they can be easily re-built outside of hardware support.

In this chapter we will begin with formalizing and implementing the concept of *counting* in Section 2.1. We will then implement a higher performance version of numbers that uses the positional system in Section 2.2.

After this chapter we will have seen that logic is powerful enough to define apparently primitive concepts such as natural numbers. Of course it would be possible to go much further, into the definition of rational numbers, and then many other concepts of traditional arithmetics. In any case for practical reasons we will not use this in later chapters as it would have a dramatic impact on performance.

[1] They are fast, and time is money. Thus, machine integers are money. QED. You never realized that your computer was indeed full of gold!

2.1 Peano numbers

An instance of an inference system begins by defining the set of recognized keywords and operators. Keywords and operators all have an ariety, a class to which they belong, and a priority level to define how lack of parentheses should be interpreted. As an example we will consider the unary notation for defining natural numbers.

2.1.1 Inductive definition of natural numbers

Natural numbers (`Num`) in the *unary notation*, also known as *Peano numbers*, are defined in terms of one zero value (`z`), which does not take anything as input and one successor value (`s`), which takes as input another number[2]

```
Data [] "z" []    Priority 2  Type Num
Data [] "s" [Num] Priority 3  Type Num
```

With the keywords above it becomes possible to express some natural numbers. A few examples are:

0 `z`

1 `s(z)`

2 `s(s(z))`

3 `s(s(s(z)))`

All the examples above are *expressions*, because they only use keywords and composition of keywords. We could also define *patterns*, which can be informally considered as "expressions with holes", where the holes are symbols that may be replaced with any valid expression. Consider pattern `s(s(a))`, which is the pattern that describes the "successor of the successor of `a`", whatever `a` will be. Possible expressions that may generated from this pattern are:

[2]The successor of a natural number is that number plus one, thus `s(a)` can be seen as intuitively equivalent to `a+1`. A word of warning: we are implying a distinction here between the operation `+1`, which is an elementary operation that we always know how to perform in a single step, and arbitrary addition of potentially large numbers: although the two operations both use the same symbol `+`, **they are not the same!**

- `s(s(z))`, for `a = z`
- `s(s(s(z)))`, for `a = s(z)`

We call expressions that may generated from a pattern *instances*. We can consider a pattern as a function that takes one or more expressions as parameters and returns a resulting expression.

2.1.2 Expressions of natural numbers

Natural numbers may be composed together to form expressions. An expression is defined recursively as the sum of two expressions, `+`, or the product of two expressions `*`:

```
Func [Expr] "+" [Expr] Priority 0  Type Expr => Num
Func [Expr] "*" [Expr] Priority 1  Type Expr => Num
```

Notice that numbers were defined in the above as belonging to class `Num`, whereas addition and multiplication belong to the class `Expr`, and both expect an `Expr` to the right and an `Expr` to the left. So far thus it would be impossible to use numbers as built with `s` and `z` as left and right parameters of `+` and `*`. In order to connect these two sets of keywords we can specify a relationship between `Num` and `Expr`, namely that a `Num` can be used where an `Expr` is expected. We can do so with the following code:

```
Num is Expr
```

The `is` operator plays a role that is akin to that of the inheritance operators found in object-oriented languages.[3]

Multiplication takes syntactic precedence over addition, thus we can be slightly less verbose with parentheses just like with the usual definitions of addition and multiplication.

Valid expressions written in this notation could be:

- `z + z`
- `s(z) + s(z)`
- `z + s(z)`

[3]To be precise, this is an example of *subtype polymorphism*.

- `s(s(z)) * s(s(z))`

We can also define patterns such as:

- `z + a`, the addition of `z` and an arbitrary expression `a`
- `s(z) + s(a)`, the addition of `s(z)` and the successor of an arbitrary expression `a`

2.1.3 Addition as repeated succession

There is no intrinsic difference between keywords such as `s` and `z`, and keywords such as `+` and `*`, even though one might be tempted to think that only because we used symbols commonly known in arithmetics then they will have their usual meaning here as well. It is very important to realize that this is not the case. We could have just as easily used other symbols such as `++` for multiplication and `?` for addition, or anything else we might have fancied. The inference system assumes no prior meaning of symbols, and indeed we are not forced in any way to follow the usual rules of engagement known for numbers. Nevertheless, to reduce the confusion, we will indeed follow such conventions.

We will now assign meaning to the various symbols we just defined. This will allow us to perform transformations on our expressions, in order to perform computations on numbers. We do so by specifiying a *set of rules*. Let us begin with addition.

We can very easily state that adding zero (`z`) to an arbitrary number `a` will result in `a` itself, without any further steps. We express this as a rule without premises as follows:

```
-----------
z + a => a
```

The lack of premises above the horizontal bar means that as soon as the inference system recognizes a pattern of the form `z+a`, then it will immediately yield `a` as the output result.

Suppose now that we add a number which is not zero to another arbitrary number. Since we are dealing with natural numbers, this means that a non-zero number will always be at least one, thus it will be at least one application of `s` to some arbitrary number. The

input pattern that describes this set of circumstances would be `s(a) + b`. Given this pattern it is not immediately possible to derive a result: rather, we must decompose the determination of the result into a series of intermediate steps. As a first step, we add `a` and `b` together. Then we can return the result by appliying the successor operation once to the result. We can express this as a single premise:

```
a + b => c
------------------
s(a) + b => s(c)
```

Notice that the above is exactly equivalent to the following very simple equation:

$$(a + 1) + b = (a + b) + 1$$

where $a + b$ is an intermediate value called `c`, and $a + 1$ is the successor of `a`, thus `s(a)`.

Termination proof (informal sketch)

One might be tempted to wonder how such a blatantly cyclical definition would ever be able to reach any useful conclusion. After all we are defining addition in terms of another addition: does this not equate to "looping forever"?

We can show, inductively, that we have no infinite looping. Let us define the `height` of an addition `a + b` as the number of applications of `s` within `a`. For example:

- `height(z + z)` is 0
- `height(z + s(z))` is 0
- `height(s(z) + z)` is 1

The **base case** of our proof is that of height equal to 0. This means that our addition takes `z` as a first parameter, and thus rule

```
-----------
z + a => a
```

can be applied. Addition terminates in this case, yielding the second term as a result.

The **inductive case** of our proof is that of height equal to $n > 1$. This means that our addition takes `s(a)` as a first parameter. We cannot apply the previous rule, but we can apply the second:

```
a + b => c
-----------------
s(a) + b => s(c)
```

This rule will terminate only if `a + b => c` terminates. Given that `height(a + b)` is $n - 1$, because of the induction hypothesis we can assume its evaluation will terminate as well. Evaluation of `s(c)` is just a trivial step which does not pose any risk of recursive behaviour, and thus we can conclude that indeed the process will terminate.

Example of addition execution

Consider the following addition: `s(s(z)) + s(z)`. Let us see how it is evaluated. First we see which rules we can apply. Clearly the first term is not `z`, thus we must apply

```
a + b => c
-----------------
s(a) + b => s(c)
```

where `a = s(z)` and `b = s(z)`. Let us perform this replacement:

```
s(z) + s(z) => c
-----------------
s(s(z)) + s(z) => s(c)
```

We cannot directly determine `c`, but we can use `s(z) + s(z)` as the current expression to evaluate. Since the first operand of the intermediate addition is not `z`, then we have to apply the same rule again, this time with `a = z` and `b = s(z)`:

```
z + s(z) => c'
--------------------
s(z) + s(z) => s(c')
---------------------- c = s(c')
s(s(z)) + s(z) => s(c)
```

Notice that the result of the second application of the rule will yield its own result, which we called `c'` to disambiguate it with the result of the first application. The successor of the result of the second

application will be the result of the intermediate application of the first rule, thus we could also write:

```
z + s(z) => c'
--------------------
s(z) + s(z) => s(c')
---------------------------
s(s(z)) + s(z) => s(s(c'))
```

Now we can apply rule

```
-----------
z + a => a
```

to the current expression to evaluate, which is `z + s(z)`, for `a = s(z)`. This leads us to finding out that `c' = a = s(z)`, thus yielding:

```
z + s(z) => s(z)
--------------------
s(z) + s(z) => s(s(z))
---------------------------
s(s(z)) + s(z) => s(s(s(z)))
```

Let us take a quick step back and realize that `s(s(z))` is the successor of the succcessor of zero, that is 2, and `s(z)` is the successor of zero, that is 1. Their sum, as computed by our system, is `s(s(s(z)))`, that is 3, precisely as we would have expected given common arithmetic sense.

2.1.4 Multiplication as repeated addition

Multiplication is defined in a way that is very similar to addition. We start with a base case of multiplication that we know how to solve immediately. From basic arithmetic we know that multiplication of zero and an arbitrary number always yields zero, thus:

```
-----------
z * a => z
```

For the intermediate step, we will use the same decomposition technique that we used for addition. The input pattern will thus be multiplication of a number greater than zero, `s(a)`, and an arbitrary number `b`: `s(a) * b`. To solve this multiplication, first we will multiply `a` and `b` together (remember the termination proof that we saw for addition: `height(a) < height(s(a))`, therefore we expect this smaller multiplication to terminate), and then we add `b` to the result:

```
a * b => c
c + b => d
--------------
s(a) * b => d
```

Notice that the above is exactly equivalent to the following very simple equation:

$$(a + 1) \times b = (a \times b) + b$$

where $a \times b$ is an intermediate value called `c`.

2.1.5 Resolving nested operations

So far we have only defined operators between numbers. As a further level of sophistication, we could define a stronger version of our operations that do not only work with numbers, but rather with arbitrarily nested operations. This means that we wish to be able to support expressions of the form `a+b+c`, which we currently have not specified how to solve.

We begin with the definition of a new operator which will be responsible for the recursive exploration of an expression in order to resolve its inner operations:

```
Func = "!" LeftArguments = [] RightArguments = [Expr] Priority = 1
    Type = Expr => Num
```

As a base case we can say that if we encounter a number, then `!` simply returns it as it was:

```
--------
!z => z

---------------
!(s a) => s a
```

The inductive step, on the other hand, works with operations and ensures that both operands have been simplified to numbers before performing the actual operation:

```
!a => a'
!b => b'
a' + b' => c
--------------
!(a + b) => c
```

```
!a => a'
!b => b'
a' * b' => c
--------------
!(a * b) => c
```

The two rules above are relatively straightforward, and require no change in the definition of addition and multiplication. Thanks to these rules, we can now solve a complex input expression such as `!(((s(s(z))) * (s(s(z)))) * (s(s(z)) + s(z)))`, equivalent to 2∗2∗(2+1)), and obtaining as a result the expected `(s(s(s(s(s(s(s(s(s(s(sz)))`
which is 12.

2.1.6 Closing considerations

The natural numbers example that we have just seen is very important. Its importance is not of a practical nature, since natural numbers are well known and moreover unary encoding is one of the least efficient ways to manipulate numbers. The importance of the example above is rather conceptual: armed with only the most basic constructs of logical derivation, a highly abstract concept, we have built something that is very concrete and apparently unrelated.

The process of logical derivation is indeed very powerful. In the following we will use it to define increasingly complex systems. We will begin by reformulating integer numbers with the well-known *binary encoding*. Then we will show how to manipulate sequences of things, in order to dispose over more complex data structures. Finally, we will build a small imperative language interpreter and its type system. Finally, we will explore the consequences of having implemented a full-blown programming language within our logical system.

2.2 Binary numbers

In this section we will study the limitations of our previous encoding of natural numbers. We will then use a positional system (the simplest of which is binary numbers) to show a much more efficient conversion process.

2.2.1 Performance considerations of unary encoding

Let us begin with an analysis of how long it takes to perform addition of two Peano numbers. Let us assume that the cost of building or decomposing terms is $O(1)$, since it can be implemented with a constant number of low-level operations such as memory allocation, runtime type checks, and similar, but it does not require recursion or iteration. Consider addition between two numbers $n + m$ with the rules:

```
----------- r0
z + a => a

a + b => c
----------------- r1
s(a) + b => s(c)
```

We have two alternatives that clearly only depend from n (m does not have influx on the number of operations, since it is just returned directly as the last step): either $n = 0$, or $n = n' + 1$. In case of $n = 0$, then rule `r0` is used, producing a result directly. This means that the number of steps T performed by the system is:

$$T(n) = 1 \text{ when } n = 0$$

In the case of $n = n' + 1$, then rule `r1` is used. Evaluating rule `r1` requires first evaluating the expression $n' + m$[4], and then performing a single operation which is the addition of the `s` keyword to intermediate result `c`:

$$T(n) = T(n') + 1 = T(n - 1) + 1 \text{ when } n = n' + 1$$

This means that the system will perform exactly $n+1$ computation steps, therefore having a complexity of $O(n)$ where n is the value of the first number. This is highly inefficient.

2.2.2 Binary encoding

Let us consider positional encoding. Specifically, let us consider binary encoding (base 2 encoding). We will define a binary number as a series

[4]Note that this again an application of *induction*, albeit less formal

of binary digits, which are either zero or one:

```
Data [] "d0" [] Priority 0 Type Digit
Data [] "d1" [] Priority 0 Type Digit
```

The binary number itself is defined as a digit, followed by the rest of the number. We will use the comma operator to specify this separation:

```
Data [Num] "," [Digit] Priority 10 Type Num
```

Of course numbers need to end at some point[5], so we define a special symbol that represents the "end of list" number:

```
Data [] "nil" [] Priority 0 Type Num
```

A number such as $3 = 011$ would then be expressed as `(((nil,d0),d1),d1)`. Notice that we associate the digits to the left, so that the first digit we encounter is the one of the lowest order. This is not strictly necessary, but it makes it easier to extract the digits in order of significance, and this in turns simplifies the traditional definition of addition.

2.2.3 Addition

To add two numbers, we will encode the well-known process of adding numbers with a *carry*. This means that we will add, repeatedly, three digits at a time: the first two digits of the numbers that we are adding, plus the carry that comes from the previous addition of digits. We now define this sum of digits:

```
Func [] "addDigits" [Digit Digit Digit] Priority 5 Type Expr => Num
```

We have eight instances of `addDigits`, one for each possible combination of digits (each digit has two possible values, `d0` and `d1`, thus we have 2^3 total combinations). For each such combination, we directly return the sum of the three digits in the format of a two-digit binary number. We need no more than two digits for the result, since the maximum number we will need to return is the sum of `d1` three times:

[5] not in real life, where we could simply assume an infinite sequence of zero's, but in a computer with limited memory this simplification is ubiquitous

```
---------------------------------
addDigits d0 d0 d0 => (nil,d0),d0

---------------------------------
addDigits d0 d0 d1 => (nil,d0),d1

...

---------------------------------
addDigits d1 d1 d1 => (nil,d1),d1
```

Addition with carry performs the digit addition of the current digits of the numbers being added, plus the current carry. The two resulting digits are the lower-order unit of the result, together with the carry that will be used to add together the remaining bits of the input numbers. The function that recursively adds the digits is `addCarry`:

```
Func [] "addCarry" [Num Num Digit] Priority 5 Type Expr => Num
```

```
addDigits da db dr => (nil,dr',d)
addCarry a b dr' => res
---------------------------------
addCarry a,da b,db dr => res,d
```

Of course addition ends when it encounters the end of the numbers, which needs to be accompanied with a null carry:

```
--------------------------
addCarry nil nil d0 => nil
```

We might even add an `overflow` keyword and graciously handle the case of non-null last carry as follows:

```
--------------------------------
addCarry nil nil d1 => overflow
```

The first step of addition simply instances `addCarry` with a null initial carry:

```
addCarry a b d0 => c
--------------------
a + b => c
```

Multiplication follows a similar scheme, and as such we will not give it here. Also, notice that we only support numbers of the same length. Supporting addition of numbers of different lengths would be relatively trivial, and as such is also not shown here.

2.2.4 Performance considerations of binary encoding

Binary encoding allows us to perform operations much faster, that is with less rules used when compared with Peano addition. Remember that Peano addition took a significant $O(n)$ amount of operations to perform, with n being the value of the first operand.

Consider only the rule which performs a single step of the binary addition:

```
addDigits da db dr => (nil,dr',d)
addCarry a b dr' => res
---------------------------------
addCarry a,da b,db dr => res,d
```

`addDigits` is just one single step, in that it directly resolves the parameters and returns the appropriate result. The only thing that matters for the purpose of the complexity of `addCarry` is the recursive call. Let us observe that the recursive call `addCarry a b dr'` uses `a` and `b` as input numbers, whereas the original addition `addCarry a,da b,db dr` used `a,da` and `b,db`, which are precisely equal to `a` and `b`, but with the additional digits `da` and `db` respectively. What is the relationship between `a` and `a,da`? Simply enough, we know that `a` is `a,da` divided by two. In general, removing the least-significant digit in a number encoded in base b is equivalent to dividing the number by b itself. Consider the following examples of removal of a least-significant digit:

$$\begin{aligned} 110/2 &= 11 \\ 111/2 &= 11 \\ 1000/2 &= 100 \end{aligned}$$

This happens because a number n in an arbitrary base b is decomposed into:

$$d_m \times b^m \cdots + d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0$$

Removing one digit decreases all powers of b, resulting in:

$$d_m \times b^{m-1} \cdots + d_2 \times b^{2-1} + d_1 \times b^{1-1}$$

which is precisely equal to the original number divided by b.

When there are no more digits, it means that the number has been divided by the base (2 for our binary numbers) enough times to render the highest coefficient null. If the original number was n, we will have stopped only after k steps such that $\frac{n}{2^k} = 0$. The smallest solution to this is actually well known, and is $k = \log_2 n$. This leads us to the conclusion that the complexity of this method of addition is $O(\log n)$, which is significantly faster than $O(n)$. As a way of comparison, assume that we are computing the sum $1000000 + 1000000$: the Peano addition will take about 1000000 steps, whereas binary addition will take about 6.

Chapter 3

Data structures

It is one thing to implement numbers, but of course in order to tackle computationally interesting problems we do need to define data structures. This should, again at a first glance, give an impression of an impossible task. After all, how could we define something as complex as a self-balancing binary tree with something as fine-grained such as logical inference?

Thanks to data structures like lists, trees, etc. defined in this chapter, we will be able to implement most of the well-known algorithms that are found in collections such as [**?**]. Moreover, these data structures will become fundamental helpers for the meta-programming that we will see in Chapter 4.

3.1 List manipulation

Our final goal is to define complex computational structures through the use of our inference system. The most complex computational structures known to us are programming languages, and as such we will set the mark there: we intend to define the syntax and semantics of a (small) programming language through a series of inference rules. Before we can dive into that, we need the ability to represent some basic algorithms which will prove of fundamental usefulness to defining a programming language. After all, our programming language will have memory of some sort, and as such we need to represent memory

somehow.

The first, and perhaps most basic data structure that we can define is the *list*. A list is used to represent a sequence of values which may contain any number of elements: from no elements (the special *empty list*) to hundreds of thousands of elements.

3.1.1 Lists of integers

For our example, we will limit ourselves to lists of integers. It will be possible to extend this definition thanks to the use of inheritance (a list of `Expr`) or with generic datatypes (which will be explored in a later section), but for now we shall keep things simple.

A list is defined very much like Peano numbers or binary digits. We have a "termination symbol" `nil`, which defines an empty list of integers:

```
Data [] "nil" [] Priority 0 Type ListInt
```

Given a list of integers (commonly known as *tail*), we can add an integer to it (commonly known as *head*) and obtain a new list of integers. Head and tail are separated by a symbol such as:

```
Data [<<int>>] ";" [ListInt] Priority 1000 Type ListInt
```

Notice that our language assumes the existance of *predefined data types*, which can be accessed with the special brackets `<<` and `>>`.

Thanks to this definition, we can now build a few example lists:

- `nil`
- `1;nil`
- `0;(1;(2;(3;nil)))`
- ...

3.1.2 List querying

We will now define a series of list operations that allow us to process a list and extract various properties from it.

Length of a list Among the most fundamental operations that we can perform on lists, we have the determination of the length of a list. We define the `length` symbol that takes a parameter of type list and returns an integer[1] as result (the length of the list):

```
Func "length" [ListInt] Priority 100 Type Expr => <<int>>
```

Determining the length of the empty list is simple, as we can immediately answer that it has a length of zero:

```
----------------
length nil => 0
```

If the input list is not empty, then we compute the length of its tail (`xs`) and then return this length plus one as the total length of the whole list:

```
length xs => y
--------------------------
length x;xs => <<1 + y>>
```

Consider now execution of program `length 1;2;3;nil`. We start with the initial proposition, which is clearly no instance of `length nil`:

```
----------------------
length 1;2;3;nil => ?
```

The proposition above instances a recursive premise on the tail of the list:

```
length 2;3;nil => y1
------------------------------
length 1;2;3;nil => <<1 + y1>>
```

Proposition `length 2;3;nil` is, once again, clearly no instance of `length nil`, and we proceed for a few steps[2] until we get to the following derivation tree:

```
length nil => y3
y2 := <<1 + y3>>
------------------------
length 3;nil => y2
```

[1]Remember that `<<` and `>>` identify built-in types, so `<<int>>` literally means a machine-integer.

[2]Which, with a smile and some handwaving, allows us to skip a couple of boring paragraphs of the book, which you smart reader feel no need for.

```
y1 := <<1 + y2>>
------------------------
length 2;3;nil => y1
-------------------------------
length 1;2;3;nil => <<1 + y1>>
```

The last premise, `length nil`, is quite obviously an instance of `length nil`, therefore we can directly jump to the result that `y3 := 0`:

```
y3 := 0
-----------------
length nil => y3
y2 := <<1 + y3>>
------------------------
length 3;nil => y2
y1 := <<1 + y2>>
------------------------
length 2;3;nil => y1
-------------------------------
length 1;2;3;nil => <<1 + y1>>
```

We can now fold back the results, just like a stack unwinding in a series of recursive calls. The first unwinding step yields:

```
length nil => 0
y2 := <<1 + 0>>
------------------------
length 3;nil => y2
y1 := <<1 + y2>>
------------------------
length 2;3;nil => y1
-------------------------------
length 1;2;3;nil => <<1 + y1>>
```

We then proceed with the second unwinding step:

```
------------------------
length 3;nil => 1
y1 := <<1 + 1>>
------------------------
length 2;3;nil => y1
-------------------------------
length 1;2;3;nil => <<1 + y1>>
```

And so on, until we reach the final expected answer which was `length 1;2;3;nil => 3`.

Searching A slightly more complex operation that we can perform is searching. Search determines whether or not a list `contains` an

element. We define the `contains` function that takes a list and an integer as parameters:

```
Func "contains" [ListInt <<int>>] Priority 100 Type Expr => <<int>>
```

The simplest test we perform is on an empty list. An empty list `nil` never contains a value `k`, independently of its value. The `contains` function in this case will thus return `no`[3] directly:

```
---------------------
nil contains k => no
```

If the input list is not empty, then it means that it is not `nil`, and therefore it is made up of an element `x` followed by the rest of the list `xs`: `x;xs`. In this case, we check to see whether or not the head of the list is equal to the search element (the single premise `x == k`). If the premise is positively discharged (it is true) then we can return `yes` as the final result:

```
x == k
-----------------------
x;xs contains k => yes
```

If the input list is not empty, but the first element is not equal to the searched key, then we need to keep searching. The result of the overall search will therefore be the result of searching in the rest of the list `xs` with the premise `xs contains k => res`:

```
x != k
xs contains k => res
-----------------------
x;xs contains k => res
```

Transforming Consider now the problem of transforming a list by incrementing all of its elements by `k`. This example is interesting because it shows how a new list can be rebuilt and returned, instead of a single value like we did in the previous examples.

We begin by deifning the `plus` function which takes as input a list and an integer:

```
Func [] "plus" [ListInt <<int>>] Priority 100 Type Expr => ListInt
```

[3]Assume we have defined symbols `yes` and `no`, but of course as a sane alternative boolean values can be used.

The transformation of an empty list remains an empty list, as there are no elements to transform:

```
------------------
plus nil k => nil
```

The transform of the non-empty list proceeds first with the transformation of the rest of the list into `xs'`, then it computes `x+k` (the transformed head) into `x'`, and finally it returns the list `x';xs'` as the whole transformed list:

```
plus xs k => xs'
<<x+k>> => x'
----------------------
plus x;xs k => x';xs'
```

Many useful functions on lists take the general form above: an immediate (often trivial) answer for the case of an empty list, and a recursive step which somehow combines premises on the first element with premises on the remaining elements. In a sense, lists are the embodiement of the simplest principles of recursion.

Sorting Let us now consider a slightly more complex algorithm, which allows us to test the algorithmic expressive power of the language. In particular, let us consider a powerful and fast sorting algorithm, the well-known *merge-sort*. Merge sort works by, recursively, dividing the list of elements into two sublists, sorting them separately, and then merging the two sorted lists into a single, final sorted list.

We need two auxiliary symbol definitions: one is the comma, which will be used to represent a pair of lists, and the other is the `mergeSort` function itself:

```
Data [] [ListInt] "," [ListInt] Priority 900 Type ListIntPair
Func [] "mergeSort" [ListInt] Priority 100 Type Expr => ListInt
```

The definition of `mergeSort` then becomes quite simple. If the list is empty, or contains only one element, then no sorting needs to take place and we can immediately return the sorted list as an answer:

```
---------------------
mergeSort nil => nil

-------------------------
mergeSort x;nil => x;nil
```

If the list contains at least two elements, then: *i*) we split it into two sub-lists `l` and `r`; *ii*) we sort `l` into `l'` with a recursive call; *iii*) we sort `r` into `r'` with a recursive call; *iv*) we merge `l'` and `r'`, therefore obtaining the final sorted list.

```
split x;y;xs => l,r
mergeSort l => l'
mergeSort r => r'
merge l' r' => res
------------------------
mergeSort x;y;xs => res
```

Splitting a list is done with function `split`:

```
Func [] "split" [ListInt] Priority 100 Type Expr => ListIntPair
```

If the list to split has less than two elements, then splitting is trivial:

```
---------------------
split nil => nil,nil

-------------------------
split x;nil => x;nil,nil
```

If the list to split has at least two elements `x` and `y`, then: *i*) we split the remaining list recursively, obtaining partial splits `l` and `r`; *ii*) list `x;l` is the first item of the result; *iii*) list `y;r` is the second item of the result.

```
split xs => l,r
---------------------------
split x;y;xs => (x;l),(y;r)
```

Let us consider a short example of how split works on a list such as:

```
------------------------
split 0;1;2;3;4;nil => ?
```

Since the list contains at least two elements, we apply the recursive rule for `x=0`, `y=1`, and `xs=2;3;4;nil`:

```
split 2;3;4;nil => l,r
---------------------------------
split 0;1;2;3;4;nil => (0;l),(1;r)
```

We apply the same rule again because `2;3;4;nil` has at least two elements, thus:

```
split 4;nil => l',r'
l := 2;l'
r := 3;r'
------------------------
split 2;3;4;nil => l,r
----------------------------------
split 0;1;2;3;4;nil => (0;l),(1;r)
```

We end up with list `4;nil`, which has only one element. This requires us to apply one of the trivial rules above:

```
l' := 4;nil
r' := nil
--------------------
split 4;nil => l',r'
l := 2;l'
r := 3;r'
------------------------
split 2;3;4;nil => l,r
----------------------------------
split 0;1;2;3;4;nil => (0;l),(1;r)
```

We can now unwind the stack of intermediate results. The first unwind step yields:

```
l := 2;4;nil
r := 3;nil
------------------------
split 2;3;4;nil => l,r
----------------------------------
split 0;1;2;3;4;nil => (0;l),(1;r)
```

The second and last unwind step returns the final result:

```
-------------------------------------------
split 0;1;2;3;4;nil => (0;2;4;nil),(1;3;nil)
```

The role of `split` is thus to split the list in two, but instead of taking the first half and then the second half of the list, we *interleave* the elements: elements in odd positions go in the first sublist, elements in even positions go in the second sublist.

Merging two lists requires a bit of care, because we take as input two sorted lists and we must give as output a single list which contains all elements of the two lists to merge *in sorted order*. Merging is done with function `merge`:

```
Func [] "merge" [ListInt ListInt] Priority 100 Type Expr => ListInt
```

If any of the input lists to merge is empty, then the answer is trivial to find and return:

```
---------------------
merge nil nil => nil

-----------------------
merge x;xs nil => x;xs

-----------------------
merge nil y;ys => y;ys
```

If the first list begins with an element that is smaller than the first element of the second list, then the resulting list will need to start with that element (and vice-versa):

```
x <= y
merge xs y;ys => res
-------------------------
merge x;xs y;ys => x;res

x > y
merge x;xs ys => res
-----------------------
merge x;xs y;ys => y;res
```

Consider now the merging of two partial sorted lists `0;2;nil` and `1;3;nil` (the lists do not need to be of the same length, nor do the elements need to interleave so strictly):

```
---------------------------
merge 0;2;nil 1;3;nil => ?
```

The first element of the first list is smaller than the first element of the second list, thus it will become the first element of the result:

```
merge 2;nil 1;3;nil => res
-------------------------------
merge 0;2;nil 1;3;nil => 0;res
```

The first element of the second list is smaller than the first element of the first list, thus it will be selected as the first element of the (intermediate) result:

```
-------------------------
merge 2;nil 3;nil => res'
res := 1;res'
---------------------------
merge 2;nil 1;3;nil => res
-------------------------------
merge 0;2;nil 1;3;nil => 0;res
```

We now select `2` as the new head of the intermediate result:

```
merge nil 3;nil => res''
res' := 2;res''
-------------------------
merge 2;nil 3;nil => res'
res := 1;res'
-------------------------
merge 2;nil 1;3;nil => res
-----------------------------
merge 0;2;nil 1;3;nil => 0;res
```

Finally, since the first list is empty, we directly return the second:

```
res'' := 3;nil
-------------------------
merge nil 3;nil => res''
res' := 2;res''
-------------------------
merge 2;nil 3;nil => res'
res := 1;res'
-------------------------
merge 2;nil 1;3;nil => res
-----------------------------
merge 0;2;nil 1;3;nil => 0;res
```

Since we have no more intermediate results to process, we can now begin the unwinding. First of all `res''` is replaced with `3;nil`:

```
res' := 2;3;nil
-------------------------
merge 2;nil 3;nil => res'
res := 1;res'
-------------------------
merge 2;nil 1;3;nil => res
-----------------------------
merge 0;2;nil 1;3;nil => 0;res
```

Then we replace `res'` with its value of `2;3;nil`:

```
res := 1;2;3;nil
-------------------------
merge 2;nil 1;3;nil => res
-----------------------------
merge 0;2;nil 1;3;nil => 0;res
```

Finally we replace `res`, therefore obtaining the final answer:

```
merge 0;2;nil 1;3;nil => 0;1;2;3;nil
```

At this point we can better discuss what `mergeSort` does, without doing the full derivation which would take too long. Suppose we wished to sort sequence `5;4;3;2;1;nil`:

```
split 5;4;3;2;1;nil => l,r
mergeSort l => l'
mergeSort r => r'
merge l' r' => res
------------------------
mergeSort 5;4;3;2;1;nil => res
```

Without going (again) into the details of `split`, we can just assume it will work as expected and obtain:

```
split 5;4;3;2;1;nil => (5;3;1;nil),(4;2;nil)
mergeSort 5;3;1;nil => l'
mergeSort 4;2;nil => r'
merge l' r' => res
------------------------
mergeSort 5;4;3;2;1;nil => res
```

At this point we make a strong assumption, which we can safely make because of the principle of induction, that is we assume that `mergeSort 5;3;1;nil` and `mergeSort 4;2;nil` will both return a sorted list. The next step is therefore:

```
mergeSort 5;3;1;nil => 1;3;5;nil
mergeSort 4;2;nil => 2;4;nil
merge 1;3;5;nil 2;4;nil => res
------------------------------
mergeSort 5;4;3;2;1;nil => res
```

Without going (again) into the details of `merge`, we can just assume it will work as expected and obtain:

```
merge 1;3;5;nil 2;4;nil => 1;2;3;4;5;nil
----------------------------------------
mergeSort 5;4;3;2;1;nil => 1;2;3;4;5;nil
```

which is precisely the expected result.

Conclusions What we have seen so far extends (with more or less translation work depending on the circumstances) to all known algorithms on lists. It is possible to write algorithms such as insertion sort, various kinds of search, and so on with minimal adjustments. In the coming sections we will dig yet deeper into recursive data structures such as binary trees.

3.2 Binary search trees

Lists are intrinsically monodimensional, without any additional information besides sequentialisation. The implication of this is that any operation on lists will require to potentially go through the whole list, and we will be sure that we can stop only after having processed all elements. For this reason lists are expensive as containers: as the number of elements of the list grows, so grows the number of steps needed for processing operations.

A solution to this issue is to build data structures where the position of elements within the data structures has some correlation with the value of the elements themselves, and stronger yet with the relationship between the value of each element and the values of other elements. One of the most used such data structures is the binary search tree. A binary search tree is a tree where each node contains an element and two subtrees, which are usually called `left` and `right`. The *fundamental search property* of binary trees states that, for any node within the tree, each element is bigger than all elements within the left tree and smaller than all elements within the right tree. Thanks to this fundamental property, when searching a tree for an element we will always be able to determine with a single comparison whether all elements of the left or right subtree may be immediately discarded.

Of course a tree may also be empty. We encode such a tree as follows, with the `nil` keyword:

```
Data [] "nil" [] Priority 0 Type BinTreeInt
```

A node of a tree, encoded with the `node` keyword, takes three parameters as input: the left sub-tree, the element of the node, and the right sub-tree:

```
Data [] "node" [BinTreeInt <<int>> BinTreeInt] Priority 1010 Type
    BinTreeInt
```

According to the definitions above, a tree such as:

```
    5
   / \
  2   7
 / \
1   3
```

would be encoded as: `node (node (node nil 1 nil) 2 (node nil 3 nil)) 5 (node nil 7 nil)`.

3.2.1 Insertion

Adding an element to a binary tree is not trivial as it is for lists. Whereas in a list we simply use `;` to push yet an element on top of the list, for binary trees we must make sure to return a tree which still satisfies the fundamental search property. The `add` keyword, which takes a left parameter which is the initial tree, and a right parameter which is the integer value to add, will return a new tree which contains the elements of the initial tree, plus the new element to add, all the while respecting the fundamental search property:

```
Func [BinTreeInt] "add" [<<int>>] Priority 100 Class Expr =>
    BinTreeInt
```

Let us begin with the simplest case of adding an element `k` to an empty tree. There is not much to do in this case, since we can simply create a new node with element `k` and empty left and right subtrees:

```
----------------------------
nil add k => node nil k nil
```

If the current node of the binary tree contains an element which is identical to the one we are inserting, then there is not much to do. In this case we simply return the original tree, as it already satisfied our requirement that the returned tree contains the desired element:

```
x == k
---------------------------------
(node l x r) add k => node l k r
```

If the current node of the binary tree contains an element which is bigger than the element we are adding, then we add the element to the left subtree and we use the resulting left subtree (which contains the elements of `l` plus `k`) as the left subtree of the final result:

```
k < x
l add k => l'
-----------------------------------
(node l x r) add k => node l' x r
```

If the current node of the binary tree contains an element which is smaller than the element we are adding, then we add the element to

the right subtree and we use the resulting right subtree (which contains the elements of `r` plus `k`) as the right subtree of the final result:

```
k > x
r add k => r'
-----------------------------------
(node l x r) add k => (node l x r')
```

For example, consider adding element `3` to tree `node nil 5 (node nil 7 nil)`:

```
5
 \
  7
```

We start with the following program:

```
----------------------------------------
(node nil 5 (node nil 7 nil)) add 3 => ?
```

The first rule which is applied is the rule for recursion in the left subtree, thus:

```
3 < 5
nil add 3 => l'
-------------------------------------------------------------------
(node nil 5 (node nil 7 nil)) add 3 => node l' 5 (node nil 7 nil)
```

Addition within an empty sub-tree is trivially resolved, thus we obtain:

```
l' := node nil 3 nil
--------------------
nil add 3 => l'
-------------------------------------------------------------------
(node nil 5 (node nil 7 nil)) add 3 => node l' 5 (node nil 7 nil)
```

We can now unwind the stack, which after just one unwinding step yields the final result:

```
-------------------------------------------------------------------------------

(node nil 5 (node nil 7 nil)) add 3 => node (node nil 3 nil) 5 (node
    nil 7 nil)
```

which (as expected) is the tree:

```
  5
 / \
3   7
```

3.2.2 Search

Just like we did for lists, we can perform a search within a binary search tree. Search determines whether or not a given element is contained within the tree, and we encode it with the operator `contains` which has a tree as its left parameter and an integer element as right parameter:

```
Func [BinTreeInt] "contains" [<<int>>] Priority 100 Type Expr =>
    YesNo
```

An empty tree never contains an element:

```
---------------------
nil contains k => no
```

On the other hand, a tree that begins with the element we are looking for trivially contains the element:

```
x == k
-------------------------------
(node l k r) contains x => yes
```

If the tree begins with an element `x` that is bigger than the element `k` we are looking for, than we know that only the left subtree is of interest: all elements of the right subtree are bigger than `x`, and thus also of `k`:

```
k < x
l contains k  => res
-------------------------------
(node l x r) contains k => res
```

Symmetrically for a tree that begins with an element `x` that is bigger than the element `k` we are looking for, than we know that only the right subtree is of interest: all elements of the left subtree are smaller than `x`, and thus also of `k`:

```
k > x
r contains k  => res
-------------------------------
(node l x r) contains k => res
```

Consider now searching element `10` within tree:

```
  5
 / \
3   7
```

This means resolving:

```
-------------------------------------------
(node nil 5 (node nil 7 nil)) contains 10 => ?
```

Since `10` is bigger than `5`, we proceed recursively into the right sub-tree:

```
5 < 10
(node nil 7 nil) contains 10 => res
-----------------------------------------------
(node nil 5 (node nil 7 nil)) contains 10 => res
```

Again, since `10` is bigger than `7`, we proceed recursively into the right-subtree:

```
10 < 7
nil contains 10 => res'
res := res'
-----------------------------------
(node nil 7 nil) contains 10 => res
-----------------------------------------------
(node nil 5 (node nil 7 nil)) contains 10 => res
```

Searching within the empty tree always fails, therefore:

```
res' := no
------------------------
nil contains 10 => res'
res := res'
-----------------------------------
(node nil 7 nil) contains 10 => res
-----------------------------------------------
(node nil 5 (node nil 7 nil)) contains 10 => res
```

At this point we can begin unwinding. The first unwinding step yields:

```
nil contains 10 => no
res := no
-----------------------------------
(node nil 7 nil) contains 10 => res
-----------------------------------------------
(node nil 5 (node nil 7 nil)) contains 10 => res
```

after just another unwinding step we end up with the final result:

```
(node nil 5 (node nil 7 nil)) contains 10 => no
```

3.2.3 Performance

The more the tree is balanced, the more effective search and insertion into a binary search tree will be. Let us intuitively analyse the number of steps we will need to perform during a binary search. Assume that we have a perfectly balanced tree with `n+1` elements; since the tree is balanced, then both the left and the right sub-trees contain exactly `n/2` elements each.

In the beginning we are searching among the whole `n+1` elements, that is our tree is:

```
  x    = 1 element
 / \
l   r  = n/2 + n/2 elements
```

After comparing the searched value and the current element of the tree, we choose one between the two sub-trees and fully discard the other. This means that after one step we focus on a smaller sub-tree:

```
  x'    = 1 element
 / \
l'  r'  ≈ n/4 + n/4 elements
```

After yet another step we get:

```
  x''    = 1 element
 / \
l'' r''  ≈ n/8 + n/8 elements
```

It should be clear that every step roughly divides the number of active elements (the set within which we are searching) by two, therefore after `k` steps we have only

$$\frac{n}{2^k}$$

elements left to consider. We are sure that at worst we will stop when this set of elements left has at most one element, thus we stop when:

$$\frac{n}{2^k} = 1$$

By multiplying both sides by 2^k, we get:

$$n = 2^k$$

And by taking the logarithm of both sides, we obtain the final result that:

$$\log_2 n = k$$

that is we will stop searching after (at most) `k` steps, which is the logarithm in base 2 of `n`.[4]

Logarithms grow quite slowly. This means that we will take roughly:

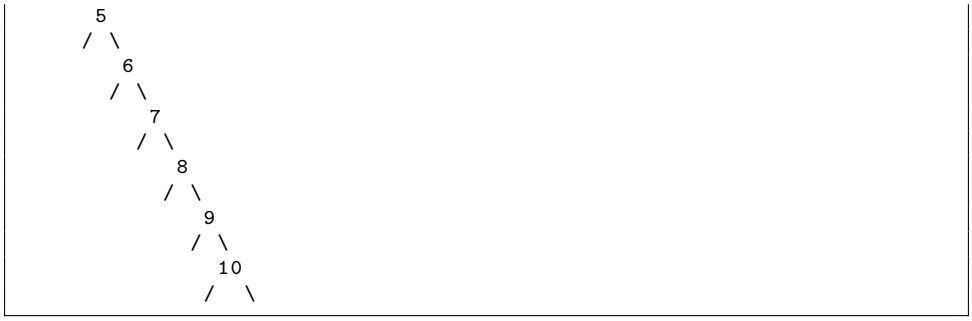
- 10 search steps if the tree has 10^3 elements
- 20 search steps if the tree has 10^6 elements
- 40 search steps if the tree has 10^{12} elements

This makes binary search trees one of the most efficient data structures for handling large data sets, but one word of warning is necessary. The original assumption that binary search trees need to be balanced for such good performance properties to be verified is quite a strong hypothesis, which needs further exploration.

3.3 Balanced binary trees

Binary search trees suffer from a problem which is known as "degeneration". A binary tree is said to be *degenerate* if for each node there is only one non-empty child. This phenomenon happens when the elements inserted do not have a substantially random distribution, that is they are partially ordered. The limit case is the insertion of a sorted sequence such as the numbers from 1 to 10, which results in the following tree:

[4]One might go back to the definition of logarithm for a moment to clarify this last step: *the logarithm is the exponent to which a fixed value, the base, must be raised to produce the desired number.* This means literally that if $\log_b(x) = y$ then $b^y = x$.



Search and removal operations in this sort of tree will have $O(N)$ complexity, which is significantly worse than the $O(\log N)$ that we expect from a balanced tree.

In the rest of this section, we will present a new sort of binary trees, 2-3-4 trees, which always remain balanced by only adding elements to the root node and therefore increasing the height of the tree uniformly by one.

There are multiple possible implementations of balanced binary trees. We present here 2-3-4 trees as they are very widely used in many common implementation, and are also slightly simpler to describe and relatively simple to implement in our language. A widely used alternative to 2-3-4 trees is *red-black* trees, which are *isomorph* to 2-3-4. In other words, any 2-3-4 tree can be translated to an equivalent red-black tree without loss of structure or information, and vice-versa.

Structure of 2-3-4 trees A 2-3-4 tree does not only have nodes with one value[5] and two children. In a 2-3-4 tree we distinguish three kinds of nodes: *i*) 2-nodes have one value k and two children l and r such that $\forall k' \in l.k' < k$ and $\forall k' \in r.k' > k$; *ii*) 3-nodes have two value k_1 and k_2 and three children l, m, and r such that $\forall k' \in l.k' < k_1$, $\forall k' \in r.k' > k_2$, and $\forall k' \in m.k_1 < k' < k_2$; *iii*) 4-nodes have three values k_1, k_2, and k_3 and four children l, m_1, m_2, and r such that $\forall k' \in l.k' < k_1$, $\forall k' \in r.k' > k_3$, $\forall k' \in m_1.k_1 < k' < k_2$, and $\forall k' \in m_2.k_2 < k' < k_3$.

An example of such a tree with a depth of only two could be:

[5] *values* are often also called *keys*.

```
 /   |    \
5   17   22 24 29
```

Here the root `10 20` is a 3-node with two elements, `10` and `20`, and three children: *i*) 2-node `5`; *ii*) 2-node `17`; *iii*) 4-node `22 24 29`.

Our goal is to insert elements in such a manner that the tree maintains its balance, independent of the order of insertion of the elements.

Our implementation of 2-3-4 trees is based on lists of elements. Lists are defined as usual, either a list is empty (`nil`) or it is the append of an element to a list (`;`):

```
Data [] "nil" [] Priority 0 Type List
Data [ListElem] ";" [List] Priority 100 Type List
```

The elements of the list can either be values (in our case just integers) or children, thus:

```
<<int>> is ListElem
BTree is ListElem
```

The tree is defined as either an `empty` node without values and children, or a proper `node` that contains a list of values and children:

```
Data [] "empty" [] Priority 0 Type BTree
Data [] "node" [List] Priority 0 Type BTree
```

The above definition means that a 2-node with value `k` and children `l` and `r` will be represented as:

```
node l;k;r;nil
```

a 3-node with values `k1` and `k2` and children `l`, `m`, and `r` will be represented as:

```
node l;k1;m;k2;r;nil
```

and similarly for 4-nodes.

Insertion The main idea behind insertion into a `2-3-4` tree is that for each node, we: *i*) find the appropriate child where we wish to insert by checking the values inside the node; *ii*) insert into that child; *iii*) if the child becomes too big (a 5-node, which is not allowed) then we split the node into two smaller nodes and insert the excess value into the current node.

Insertion is thus a function that given a tree and an element to add returns the transformed tree:

```
Func [BTree] "insert" [<<int>>] Priority 0 Class Expr => BTree
```

Insertion into an empty tree is quite simple, in that we add the element between two empty nodes, therefore creating a trivial 2-node. This will only happen during the first insertion, that is when the tree is empty:

```
-----------------------------------------------
empty insert kv => node (empty;(kv;(empty;nil)))
```

For non-empty trees we will always stop right before the empty children, and we will always insert into leaf nodes. If the node is not a leaf, then we find the appropriate position and insert the new value there with function `insertInto`. This yields a new tree which needs to be `split` if its root is too big (a 5-node):

```
isLeaf l => no
kv insertInto l => l'
split l' => res
---------------------------------------
(node l) insert kv => res
```

If the node is indeed a leaf, then we find the appropriate position within its list of content and add the element plus a new `empty` child with the `insertSorted` function. The node, just like in the previous case, might become too big (a 5-node) as a result of insertion, and may thus need to be `split`:

```
isLeaf l => yes
l insertSorted kv empty => l1
split l1 => res
-------------------------------
(node l) insert kv => res
```

The function that checks whether or not a tree is a leaf takes as input its list of values and children and returns a boolean value:

```
Func [] "isLeaf" [List] Priority 0 Type Expr => YesNo
```

`isLeaf` returns `yes` when the children of the node are all `empty`. Since we know that this is a balanced tree, then we know that if one child (for example the first) is `empty`, then all other children will be as well:

```
---------------------------------
isLeaf (empty;rest) => yes
```

If the first child of the tre is not `empty`, then we can be sure that all other children are not empty as well and thus we are not in presence of a leaf:

```
l != empty
---------------------------------
isLeaf (l;rest) => no
```

In case of leaves, we simply[6] need to find the proper position within the list where to insert our element. The `insertSorted` will therefore take a list of values and trees plus a value to insert as input, and return another such list:

```
Func [List] "insertSorted" [<<int>>] Priority 0 Type Expr => List
```

If we reach the end of the list, then we simply add the value followed by the `empty` tree. The function `insertSorted` is only called on leaves, thus all children are already `empty`:

```
-------------------------------------
l;nil insertSorted k => l;k;empty;nil
```

If we find the element already in the tree, then we do nothing and return the original tree untouched:

```
x == k
--------------------------------
l;x;xs insertSorted k => l;x;xs
```

If the element to insert is bigger than the current value in the list, then we have found the point of insertion. We put the new element right before the bigger element and between them we put an `empty` child:

```
x > k
----------------------------------------
l;x;xs insertSorted k => l;k;empty;x;xs
```

If the element to insert is smaller than the current value in the list, then we will have to go on looking in the rest of the list. We perform the insertion in the rest of the list, which then becomes the updated tail[7]:

[6] For a large enough value of "simply".

[7] *Tail* is a common name given to "the rest of a recursively defined list".

```
x < k
xs insertSorted k => xs'
--------------------------------
l;x;xs insertSorted k => l;x;xs'
```

Insertion into non-leaf elements is very similar to `insertSorted`. The first aspect of `insertInto` is that it looks for the right position in the list of values and children where the insertion needs to happen. Whereas `insertSorted` at that point simply adds two elements to the list, `insertInto` recursively calls `insert` into the appropriate child. We know that the child where we perform the recursive insertion is not `empty` because otherwise `insertSorted` would have been called instead of `insertInto`.

`insertInto` takes as input the value to insert and the list where the insertion needs to take place, and returns the list with the value added:

```
Func [<<int>>] "insertInto" [List] Priority 0 Type Expr => List
```

The base case of `insertInto` happens when we reach the last child of the input list. In this case we cannot proceed forward, and therefore we will have to perform the insertion into the last child itself. After the recursive insertion step is performed, the child is updated and we need to `merge` it back into the original list:

```
l insert k => l'
merge l' nil => l''
---------------------------
k insertInto l;nil => l''
```

If the value we are trying to insert is smaller than the current value, then we have found the position of insertion and perform the `insert` and the `merge` steps precisely like we did for the previous case:

```
x > k
l insert k => l'
merge l' x;xs => l''
---------------------------
k insertInto l;x;xs => l''
```

If we find the value we are trying to insert, then we simply return the original list unchanged:

```
x == k
-----------------------------
k insertInto l;x;xs => l;k;xs
```

If the value we are trying to insert is bigger than the current value, then we try to insert it at a later position with a recursive call to `insertInto` the tail of the list `xs`:

```
x < k
k insertInto xs => xs'
------------------------------
k insertInto l;x;xs => l;x;xs'
```

The role of `split` is extremely important. After insertion, we may get a 5-node, which is not allowed. Split therefore takes as input a list of values and nodes, and returns a 2-3-4 tree where the eventual 5-nodes have been split:

```
Func [] "split" [List] Priority 0 Class Expr => BTree
```

If `split` encounters a 2-node (list of three elements), 3-node (list of five elements), or 4-node (list of seven elements), then it simply "wraps" the list into a single node and returns the node as the resulting tree:

```
-----------------------------------------
split l;(k;(r;nil)) => node l;(k;(r;nil))

-------------------------------------------------------------
split l;(k1;(m;(k2;(r;nil)))) => node l;(k1;(m;(k2;(r;nil))))

-----------------------------------------------------------------------

split l;(k1;(l_m;(k2;(r_m;(k3;(r;nil)))))) => node l;(k1;(l_m;(k2;(
    r_m;(k3;(r;nil))))))
```

If the list is a 5-node, then it contains nine elements. We take the middle value as a *pivot*, and use the first three elements of the list as a 2-node and the remaining five elements of the list as a 3-node. We then return a 2-node where the newly created elements are the left and right child and the *pivot* is the value:

```
l' := node l;(k1;(l_m;nil))
r' := node m;(k3;(r_m;(k4;(r;nil))))
-----------------------------------------------------------------------------

split l;(k1;(l_m;(k2;(m;(k3;(r_m;(k4;(r;nil)))))))) => node l';(k2;(r
    ';nil))
```

After insertion we always perform a `split`. The fact that we just performed an insertion means that the sub-tree is now bigger by one element. The fact that we just `split` means that if we find a 2-node

then this node was just split and therefore its single element should be added to the current list of elements, thereby promoting it by one level and therefore potentially propagating the excess size to the parent node. This way, if we need to create a new node, we keep sending new elements to parent nodes until we reach the root. When we split the root, then *the whole tree* has grown by one level. This will always keep the tree perfectly balanced.

`merge` takes as input a tree (where the insertion just took place) and the list of elements where the node needs to be merged, and returns a new list of elements where the input tree has been added:

```
Func [] "merge" [BTree List] Priority 0 Type Expr => List
```

If we find a 2-node, then we know that it has just been split. This means that we effectively promote its value to the current level:

```
--------------------------------------
merge (node l;(k;(r;nil))) xs => l;(k;(r;xs))
```

If we find a 3-node or a 4-node (in general a node where the input list is longer than four elements) then we simply add this node as a single child of the list to return:

```
------------------------------------------------
merge (node l;(k1;(m;(k2;rs)))) xs => (node l;(k1;(m;(k2;rs))));xs
```

Let us now try to visualize[8] this process. We will consider the most interesting path of insertion, that is insertion into a tree which needs to grow in height by one.

The initial tree, where we shall insert the value 9, is:

```
   1  3  5
 /  |   |  \
0   2   4  6 7 8
```

The initial call to `insert` processes the root of the tree. Since the root is not a leaf (all its children are non-empty), then it will perform an `insertInto` followed by a `split`.

`insertInto` finds that 9 is bigger than all elements of the root, thus insertion needs to happen in the right-most child: `6 7 8`. This leads us to another call to `insert` of value 9 into tree, which will then be followed by a call to `merge`.

[8] "Try" is the keyword here.

```
  6 7 8
/  |  \
```

Since the node is a leaf, then `insert` calls `insertSorted`, followed by a `split`. `insertSorted` simply places the new value into the right position in the tree:

```
  6 7 8 9
/ | | | \
```

The first `split` creates a new 2-node:

```
     7
   /   \
  6     8 9
/ \   / | \
```

This tree is sent back to as the result of `insertInto`, which merges it with the list of values which was the original root of the tree (that is `|2|4|5|`). `merge` adds the value of the 2-node as a sibling of the elements of the root, since it recognizes the result of a succesful split. The resulting tree (not yet valid) is thus:

```
   1  3  5  7
 /  |   |  |  \
0   2   4  6  8 9
```

This is returned as the result of `insertInto`, which is then processed by the last call to `split`. Since `split` finds a 5-node, it proceeds with its splitting and returns the result as the final tree:

```
          3
       /     \
      /       \
   1          5  7
 /   \      /  |  \
0     2    4   6  8 9
```

This elegant algorithm has thus managed to push the excess elements back up until the root was reached, therefore performing the final split at the root instead of one of the leaves and thus all nodes increase by one level of height instead of just a few, which would progressively unbalance the tree. Notice that the nodes of the resulting tree are all 2-nodes or 3-nodes, therefore it will take a few more insertions (at least three) before we can increase the height of the tree yet again.

A coward's approach to search We now quickly conclude the section by showing a "lazy" (in the sense that we use very simple predicates and some code duplication) approach to lookup. We define function `contains` which determines whether or not a value is present within the tree:

```
Func [] "contains" [BTree <<int>>] Priority 0 Type Expr => Bool
```

The trivial case is that the search has ended up in an empty tree. In this case we simply return a negative result:

```
-----------------------
contains empty k => no
```

If we have found a two-node, then we check whether the value of the node is the one we are looking for. If this is the case, then we can return a positive result:

```
x == k
-----------------------------------
contains (node l;x;r;nil) k => yes
```

If the value of the node is bigger (smaller) than the searched value, then we proceed with a recursive search in the left (right) child:

```
k < x
contains l key => res
---------------------------------
contains node(l;x;r;nil) k => res

k > x
contains r key => res
---------------------------------
contains node(l;x;r;nil) k => res
```

Similarly, for 3-nodes we need to find the proper interval where search must recursively continue:

```
key > x1
key < x2
contains m k => res
----------------------------------------
contains node(l;x1;m;x2;r;nil) k => res
```

The rest would need to be written by hand.

A courageous approach to search The above is perhaps useful for illustration purposes. As an alternative, we could write a recursive search functions similar to `insertSorted` or `insertInto`.

Containment fails immediately on an `empty` sub-tree:

```
----------------------
contains empty k => no
```

On a non-empty `node` we invoke auxiliary function `containsIterate`, which simply takes a list of elements instead of a tree as a parameter:

```
Func [] "containsIterate" [List <<int>>] Priority 0 Type Expr => Bool
```

```
containsIterate l k => res
---------------------------
contains node l k => res
```

`containsIterate` defaults to searching in the right-most child if no other suitable children have been found so far:

```
contains r k => res
-------------------------
containsIterate r;nil k => res
```

If the current value of the list is equal to the value we are searching, then search has been concluded positively:

```
k = x
--------------------------------
containsIterate l;x;xs k => yes
```

If the current value of the list is bigger than the value we are searching, then we have found the child where search needs to continue recursively:

```
k < x
contains l k => res
--------------------------------
containsIterate l;x;xs k => res
```

If the current value of the list is smaller than the value we are searching, then we have not yet found the child where search needs to recurse, and we must continue iterating the elements of the list:

```
k > x
containsIterate xs k => res
--------------------------------
containsIterate l;x;xs k => res
```

This second formulation of search is quite shorter than the previous, and illustrates nicely how a well-though out recursive procedure

is often much shorter to write and read [9] than lots of repeated, boilerplate code.

[9]but perhaps not invent

Chapter 4

Syntax and semantics of a programming language

As our "crowning achievement", we will push our logic language yet further. We will define the semantics of a small, imperative programming language within our logic language. Thanks to this we will see that even the most complex scenario, that of building a whole programming language, is not too much for our programming language.

4.1 Evaluating expressions

After data structures, which are a complex part of programming knowledge, we can move on to an even more ambitious goal. We will now try and build a full-blown programming language with our logic language. We will begin with simple arithmetic expressions, which we will augment with memory and flow-control statements.

4.1.1 Evaluating simple arithmetic expressions

Simple arithmetic expressions are defined as trees[1]. Each node of the tree is either a `+` or a `*`, and the two children of the nodes are the sub expressions that must be, respectively, added or multiplied together. The recursion ends when we find a leaf (`$`), which contains a single

[1]These are, surprisingly, commonly referred to as *expression trees*.

integer value. Examples of expressions that we can represent are `3 + 5`, `5 + 2 * 3`, etc.

The reason why such expressions can be represented with a tree can be visualized graphically quite intuitively. Consider expression `5 + 2 * 3`; we can render it into a tree as:

```
  +
 / \
5  *
  / \
 2   3
```

The above definition of expressions can be summed up into three new data declarations:

```
Data [Expr] "+" [Expr] Priority 10 Type Expr
Data [Expr] "*" [Expr] Priority 20 Type Expr
Data [] "$" [<<int>>] Priority 10000 Type Value
```

Of course a `Value` sometimes needs to be used where an `Expr` is expected, therefore we add[2]:

```
Value is Expr
```

Evaluating an expression is done with the `eval` function, which takes an `Expr` as input and then returns an `<<int>>`:

```
Func "eval" [Expr] Priority 1 Type Expr => <<int>>
```

When the expression to evaluate is an integer constant (the `$` operator is just a wrapper to make an `<<int>>` appear as an `Expr`), then the result of the evaluation is just the integer constant itself:

```
---------------
eval ($i) => i
```

When the expression to evaluate is a compound expression of sub-expressions `a` and `b`, then: *i*) we evaluate the sub-expressions into `x` and `y` respectively; *ii*) we combine `x` and `y` into the final result `res` by invoking native operations such as machine integer addition between `<<` and `>>` brackets.

```
eval a => x
eval b => y
<<x+y>> => res
------------------
```

[2]At this point we might even say "the usual".

```
eval (a+b) => res

eval a => x
eval b => y
<<x*y>> => res
------------------
eval (a*b) => res
```

It is trivial to extend the above to yet more arithmetic operations such as subtraction, division, etc.

Consider now the evaluation of the expression seen before: `$5 + ($2 * $3)`:

```
-----------------------
eval ($5+($2*$3)) => ?
```

The first rule that is instanced is the rule for addition:

```
eval $5 => x
eval ($2*$3) => y
<<x+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

The premise above (`$5`) is very easily solved by applying the rule for constants:

```
x := 5
--------------
eval $5 => x
eval ($2*$3) => y
<<x+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

At this point we cannot proceed before unwinding the stack by one step, thus we obtain:

```
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

We now have to resolve the second premise, thus we apply the rule for multiplication and we obtain:

```
eval $2 => x'
eval $3 => y'
<<x'*y'>> => res'
y := res'
------------------
eval ($2*$3) => y
```

```
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

We can now use a rule to solve the evaluation of the first constant:

```
x' := 2
--------------
eval $2 => x'
eval $3 => y'
<<x'+y'>> => res'
y := res'
------------------
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

We unwind the stack by one step:

```
eval $3 => y'
<<2*y'>> => res'
y := res'
------------------
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

Again we apply the rule for constants:

```
y' := 3
--------------
eval $3 => y'
<<2+y'>> => res'
y := res'
------------------
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

We unwind the stack yet again:

```
<<2*3>> => res'
y := res'
------------------
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

At this point we can just unwind the stack multiple times, from:

```
y := 6
------------------
eval ($2*$3) => y
<<5+y>> => res
-------------------------
eval ($5+($2*$3)) => res
```

to:

```
<<5+6>> => res
-------------------------
eval ($5+($2*$3)) => res
```

And finally we obtain the desired result of:

```
-----------------------
eval ($5+($2*$3)) => 11
```

4.1.2 Evaluating expressions with memory

Suppose that we now want to be able to define expressions with symbols such as ‘‘x’’ or ‘‘y’’ in the middle, for example: `5 + 2 * x`.

The immediate result is that to represent such an expression we need a new data term for variables[3]

```
Data [] "!" [<<string>>] Priority 10000 Type Variable
```

Of course since we want variables to be useable where expressions are expected, we add:

```
Variable is Expr
```

Evaluation of an expression cannot be done directly, because we encounter variables we do not know what value to assign them. For this reason we shall extend the `eval` function so that it also takes as input parameter a `Memory`:

```
Func [] "eval" [Expr Memory] Priority 1 Type Expr => <<int>>
```

In the following we assume that `Memory` is any container of data. The concrete implementation of `Memory` is actually not that important for the purpose of our scneario: it could be a list of `<<string>>`,

[3]This term, much like `$` for integer constants, is essentially just a wrapper around `<<string>>` that allows us to treat a string as an `Expr`.

`<<int>>` pairs, a binary search tree, a balanced binary search tree, or some other data structure such as those defined in Chapter 3. The only requirement that we impose is that we have an operator to `lookup` the value of a variable given its name:

```
Func [Memory] "lookup" [<<string>>] Priority 10 Type Expr => <<int>>
```

The rules that we have seen so far are substantially unchanged, that is they do not use the memory but rather just pass it around:

```
-----------------
eval ($i) m => i

eval a m => x
eval b m => y
<<x+y>> => res
------------------
eval (a+b) m => res

...
```

When we encounter a variable, which is prefixed by `!`, we simply look up its value from memory and return it:

```
m lookup v => res
------------------
eval !v m => res
```

Consider now a very simple evaluation: expression `!v + $3`, where memory is `[v -> 10]`. We start at:

```
------------------------------
eval (!v + $3) [v -> 10] => ?
```

We begin by applying the rule for addition, which yields:

```
eval !v [v -> 10] => x
eval $3 [v -> 10] => y
<<x + y>> => res
--------------------------------
eval (!v + $3) [v -> 10] => res
```

We now use the rule for variable lookup, which results in:

```
[v -> 10] lookup v => res'
x := res'
--------------------------
eval !v [v -> 10] => x
eval $3 [v -> 10] => y
<<x + y>> => res
--------------------------------
eval (!v + $3) [v -> 10] => res
```

The result of looking up is that of obtaining the appropriate value from memory, therefore leading to:

```
x := 10
--------------------------
eval !v [v -> 10] => x
eval $3 [v -> 10] => y
<<x + y>> => res
--------------------------------
eval (!v + $3) [v -> 10] => res
```

We must now unwind the stack once for `x`:

```
eval $3 [v -> 10] => y
<<10 + y>> => res
--------------------------------
eval (!v + $3) [v -> 10] => res
```

We have to evaluate constant `$3`, which immediately yields `3` itself as the result:

```
<<10 + 3>> => res
--------------------------------
eval (!v + $3) [v -> 10] => res
```

The final unwinding step gives us the expected result of:

```
------------------------------
eval (!v + $3) [v -> 10] => 13
```

4.1.3 Evaluating statements

Let us now consider yet another extension: we want to be able to assign variables. This means that we have a new possible expression data term, which we will use for assigning values to variables:

```
Data [Variable] ":=" [Expr] Priority 1 Type Expr
```

Evaluation of expressions such as assignment can be considered to return nothing, and just affect memory. We therefore define the `nothing` value:

```
Data [] "nothing" [] Priority 1 Type Value
```

A natural extension of expression that comes from assignment is the fact that expressions can also be sequenced together, much like lists:

```
Data [] "nil" [] Priority 1 Type Expr
Data [Expr] ";" [Expr] Priority 1 Type Expr
```

After evaluating an expression, we do not just get the resulting value; this happens because if one expression contains an assignment then evaluation will return the resulting value, but also the changed memory after the assignment. We therefore define a new data type that contains both a value and memory, and extend the return type of `eval` so that it now returns such a pair:

```
Data [Value] "," [Memory] Priority 1 Type ValueMemory
Func [] "eval" [Expr Memory] Priority 1 Type Expr => ValueMemory
```

Evaluating a null expression is quite simple: we just return nothing and pass the memory through:

```
------------------------
eval nil m => nothing,m
```

When evaluating a sequence of expressions, we: *i*) evaluate the first sub-expression with the input memory `m`, therefore obtaining the new memory `m1`; *ii*) evaluate the second sub-expression with the new memory `m1`, therefore obtaining a result `res` and new memory `m2`; *iii*) return `res` and `m2` as the final result of the whole expression:

```
eval a m0 => nothing,m1
eval b m1 => res,m2
------------------------
eval (a;b) m0 => res,m2
```

It is important to realize that the `;` operator role is simply that of letting the changes done to memory "fall through" from one expression to the next.

Assume now that we have some implementation of a function that adds (or updates) a key-value parir into memory:

```
Func [Memory] "add" [<<string>> <<int>>] Priority 1 Type Expr =>
    Memory
```

We can use this function to define how the assignment operator works: *i*) we evaluate the right-hand side of the assignment, obtaining a result `res` and a potentially changed memory `m1`; *ii*) we bind `res` to `v`, therefore obtaining a new memory `m2`; *iii*) we return `nothing` and the final memory `m2` as the final result.

```
eval e m => res,m1
m1 add v res => m2
-------------------------------
eval (!v := e) m => nothing,m2
```

This actually gives us a complete memory model which can be extended to operators of all kinds. As strange as it may sound to think of memory as just another value that we freely pass around, consider the possibilities that this manner of reasoning offers us: *i*) we might decide to build a system where memory can be split so that one thread gets some variables and another thread gets some other variables, so that no risky interaction is possible; *ii*) we might define an operator that saves memory in order roll-back the program state to an earlier safe state in case of error; *iii*) we might define a transactional operator that creates a new copy of memory which is only committed after some kind of confirmation; *iv*) etc.

The possibilities of such a model are endless, and it is important not to be fooled by the fact that we are limiting ourselves *on purpose* to implement only well-known models of computation. [4]

4.1.4 Evaluating control flow statements

Most of the work has been done so far, but we can now add some finishing touches to turn our tiny language into something more complete, albeit still quite primitive. The final extensions to our language will therefore be a series of control flow operators such as `if`, `while`, etc., plus a few "decorative" symbols such as `then`, `else`, and `do` which, although not strictly necessary, make code much more readable:

```
Data [Expr] "gt" [Expr] Priority 10 Type Expr
Data [] "if" [Expr Then Expr Else Expr] Priority 5 Type Expr
Data [] "then" [] Priority 10000 Type Then
Data [] "else" [] Priority 10000 Type Else
Data [] "while" [Expr Do Expr] Priority 5 Type Expr
Data [] "do" [] Priority 10000 Type Do
```

Since we will now also need to manipulate boolean expressions, we will create a new data type to wrap boolean values:

[4] Indeed, new models of computation could just as easily be built, even though they would be quite poor examples for illustration!

```
Keyword [] "?" [<<bool>>] Priority 10000 Type Value
```

Since evaluation may now return values as well, we update the definition of `eval` so that its return type is now `Value` instead of just `<<int>>`. Evaluating a `gt` (greater than) comparison now looks very much like the evaluation of a sum or a product:

```
eval a m0 => $x,m1
eval b m1 => $y,m2
x > y
-----------------------------
eval (a gt b) m0 => ?true,m2

eval a m0 => $x,m1
eval b m1 => $y,m2
x <= y
------------------------------
eval (a gt b) m0 => ?false,m2
```

When evaluating a conditional statement, then we first evaluate the condition. If the condition evaluates to a value of `true`, then we evaluate the `then` branch and return the result of its evaluation:

```
eval c m0 => ?true, m1
eval a m1 => res,m2
--------------------------------------
eval (if c then a else b) m0 => res,m2
```

If the condition evaluates to a value of `false`, then we evaluate the `else` branch and return the result of its evaluation:

```
eval c m0 => ?false, m1
eval b m1 => res,m2
--------------------------------------
eval (if c then a else b) m0 => res,m2
```

The evaluation of a `while` loop is just as simple. We evaluate the condition of the `while` loop, and if we obtain a result of `false` then we are done and we can return a result of `nothing`:

```
eval c m0 => ?false,m1
-----------------------------------
eval (while c do b) m0 => nothing,m1
```

If the condition of the loop evaluates to `true`, then we evaluate the body of the loop once, obtaining a new memory `m2`. This new memory will be used to re-evaluate the `while` loop:

```
eval c m0 => ?true, m1
eval b m1 => nothing,m2
eval (while c do b) m2 => nothing,m3
-------------------------------------
eval (while c do b) m0 => nothing,m3
```

In this case we will skip a practical example as it would be quite long, and it does not really add much to the examples seen previously in the chapter.

Conclusions It is quite possible to further extend our tiny language to all sorts of additional niceness. For example, we could define functions, we could distinguish between *heap* and *stack* memory, we could define data structures, etc. Of course this would quite complicate our definitions, but surprisingly less than one would expect. For example, a much richer implementation that also contains declaration and even scoping of variables (the fundamental construct that makes function definition possible) requires as little as 360 lines of code. The implementation of a real-time oriented programming language for game development (Casanova) took less than 600 lines of code. Compared with the equivalent definition of a compiler written by hand, we are talking about a significant difference; the Casanova compiler written in F# is almost two orders of magnitude larger than the new implementation, at a whopping 10,000 lines of code.

Moreover, primitive or not, our tiny language now supports integer variables and some looping constructs. This means that the sample language we have defined in this chapter enjoys a very important property known as *Turing completeness*, which guarantees that it is powerful enough to compute anything that is computable. This important property and its consequences will be further elaborated in Chapter 5.

Chapter 5

Closing remarks

As we have seen so far, we have shown that we can build a multitude of different things with our logic language: *i*) basic concepts from mathematics such as numbers in various formats; *ii*) various concepts from programming such as data structures and even interpreters for programming languages.

It is quite evident that any mathematical procedure can be defined as a logic program, well beyond the simplicity of implementing numbers. Moreover we could build a modern assembler and use our logical language as a "universal" interpreter, therefore running any existing software in it (albeit at a snail's pace). If we can implement the whole of mathematical procedures and the whole of programming, then what is left that we cannot compute? In this chapter we draw some closing remarks on the field of computability and what consequences this has for programming languages and logical thinking in general.

5.1 Fragments of computability

As a closing remark, let us briefly consider the question of *what can and cannot be computed* with our logic language. On the surface our logic language looks very different from other programming languages such as, say, Java or F#. Does this difference translate into higher or lower expressive power? In this chapter, we briefly and informally explore questions about the expressive power of different programming

languages and programming paradigms.

We will do so to justify the importance of programming languages and appropriate formalisms, but not for the purpose of expressing constructs that cannot be expressed into the other formalisms: rather, different languages and formalisms allow *human users* to better reason, and as such do offer a different perspective on the same object (the set of all programs) which lets some different properties emerge better.

We will begin by discussing intrinsically hard problems that cannot have an algorithmic answer. These problems make up the boundaries of computability, and as such if we could[1] define a formalism that solves one of this problems algorithmically then we would have found something that is intrinsically more expressive than the formalisms and languages that we use now.

5.1.1 Halting problem

We will now explore the simple question: *are there limits to what can be computed?* In order to effectively manipulate programs we build a way to translate an arbitrary formula within some formalism into a natural number, that is we show a way to encode arbitrarily complex data structures into simpler objects that are easier to manipulate.

This means that any statement within our formalism is now a natural number, and thus we have an isomorphism between "properties of statements" $\Leftrightarrow$ "properties of numbers", but "properties of numbers" are easier to determine algorithmically because we do not need to define data structures.

Gödel numbering There are many ways to define such an encoding. We present Gödel's original encoding from many decades ago, mostly for historical reasons[2]

We assign a unique natural number to each basic symbol in our language, for example:

[1] Spoiler alert: we can nooooooooooooooooooot!

[2] A nice side-effect is that this encoding still works, which in a world of short-lived, superficial technological innovation where stuff stops working after a couple of years might seem strange. "Should we not use something new and shiny?", I hear you say: "no."

- `if` $= 0$
- `var` $= 1$
- `,` $= 2$
- ...

Given any formula $x_1, x_2, \ldots, x_n$, x_i where each x_i is a symbol with associated a number from the list above, we define the encoding function:

$$enc(x_1, \ldots, x_n) = 2^{x_1} \cdot 3^{x_2} \cdot \cdots \cdot p_n^{x_n}$$

where p_n is the n-th prime number. This means that we encode the i-th symbol of the sequence as the exponent of the i-th prime number. According to the fundamental theorem of arithmetic, we can always extract back the original sequence with a finite number of steps.

For example, in a specific Gödel numbering, the Gödel number for 0 could be 6, and the Gödel number for = could be 5. Thus, in such a system, the Gödel number of formula $0 = 0$ would be $2^6 \times 3^5 \times 5^6 = 243000000$.

Halting Let us now consider the halting problem itself. Consider the problem of determining whether a program `i` terminates on input `x`. We can formulate this problem as finding function $h(i, x)$ that returns:

- 1 if program i terminates on input x
- 0 otherwise

Since we wish to automate this process, we want to find a way to translate function $h(i, x)$ into some program. For this to be possible, then function h needs to be *total* and *computable*. A function is total if, for every input, it gives back a result. Trivially, function h is defined so as to always return either 0 or 1, therefore it is total by definition. A function is computable if it can be encoded into a Turing machine or any equivalent formalism such as the λ-calculus. We shall further

discuss these formalisms later, but for the moment let us treat them as mathematically flavoured assembly languages.

We will now show[3] a shocking result: no arbitrary total computable function f can be equal to h, and we will do so by constructing a counter example. The outline of the proof is as follows:

- we consider an *arbitrary* total computable function f
- we construct a partial but computable function g that is based on f
- we feed g into h (the halting function) and show that $f \neq h$
- since f was an *arbitrary* total computable function, then there exists no total computable function $f = h$

Let us now consider an arbitrary total computable function f in two arguments. We then define partial computable function $g(i)$ so that it returns:

- 0 if $f(i, i) = 0$
- $\uparrow$ otherwise

Note that in the above, $\uparrow$ denotes no or undefined return value. This is allowed, as g was meant to be a partial function. Not returning a value in practice amounts to infinite looping or (tail-)recursion, which effectively prevents a function from returning anything.

Since we chose f to be totally computable, then as a consequence g is will also be computable. We could formulate the above as follows in the logic programming language defined so far:

```
f i i => 0
-----------
g i => 0
```

As long as `f i i` is also defined within the same program and always returns a result, then the above is a valid formulation within our language.

[3] *Show* in this case is meant in the informal sense.

Since g is computable, albeit partial, we can give a program computes g and encode it with Gödel's numbering into e_g.

Let us now consider what happens if we give e_g *twice* to function h. $h(e_g, e_g)$ will return:

- $f(e_g, e_g) = 0 \rightarrow g(e_g) = 0 \ \rightarrow h(e_g, e_g) = 1$
- $f(e_g, e_g) = 1 \rightarrow g(e_g) = \uparrow \rightarrow h(e_g, e_g) = 0$

The above means that whatever f returns, then h returns something else. This means that $h \neq f$, but since f was an *arbitrary* total computable function, then h is different from it and thus h is different from all total computable functions.

The consequence of this is that h is not computable, that is we cannot encode it within a Turing machine, λ-calculus, etc.

Not all hope is lost, because even if we cannot give a function h that always answers `yes` or `no`, nothing stops us from building an *approximated* version of it. For example, we might restrict ourselves to solutions such as: *i*) a partial function, that may sometimes loop forever; *ii*) a function that can also return no answer, such as `unknown`; *iii*) a function that gives probabilistic answers; *iv*) ...

We will focus on approximation techniques (even though not applied to the halting problem itself but one of its cousins) in later chapters.

5.1.2 Some desperate grasping

A reader with an active imagination might, at this point, be trying to find reasons why this does not really apply to his or her experience with programming. After all, this might just be an isolated incident, and most interesting things hopefully remain easily doable, right?[4]

Unfortunately for us, the mere fact that we cannot solve the halting problem really means that we cannot solve many hardcore problems that would make the life of programmers much easier.[5]

[4]Yeah, you wish: this chapter goes from bad to worse, so giving up hope now will save you from depression further down the road. Sorry.

[5]This is actually the only silver lining: insightful programmers cannot be automated away, and will never be out of a job. Up yours, robotics!

Some examples of problems that cannot be solved automatically as a consequence of the halting problem are:

- **EQUIVALENCE PROBLEM** Given two programs, test to see if they are equivalent.
- **SIZE OPTIMIZATION PROBLEM** - Given a program, find the shortest equivalent program.
- **GRAMMAR PROBLEM** Given two grammars, find out whether they define the same language.
- ...

All of the problems above cannot be solved, because solving them would make it possible to also solve the halting problem.[6]

For example, consider the equivalence problem. Suppose (*ad absurdum*[7]) we have a function `equivalent` that is capable of reliably testing whether two programs are equivalent on some input. We will now show how we could use this equivalence test to determine whether or not a program `P` halts on input `i`. To do so, we construct a new program `Q` that works precisely like `P`, but which returns `1` whenever `P` would return something. This would be equivalent to simply ignoring the result of `P`, for example:

```
P => res
---------
Q => 1
```

We now construct a new, trivial program `T` that always returns `1`:

```
-------
T => 1
```

We now use our function `equivalent` on `Q` and `T`. If `Q` and `T` are equivalent, then this means that `Q` halts, therefore `P` halts as well. Since we know that we cannot solve the halting problem, then our original

[6]This proof strategy is known as *reduction*, because it reduces one problem to another one, showing that solving the first implies solving the second or vice-versa.

[7]A proof based on *reductio ad absurdum* assumes something that we suspect to be false and shows that from this assumption stems a clear contradiction against some previously established fact.

hypothesis that we have a total, computable `equivalent` function was absurd.

Similar proofs can be set up for all problems that involve the automated, reliable determination of complex properties of programs. This generalization is known as *Rice's Theorem*, which states that, for any non-trivial property of partial functions, there is no total computable function that decides whether an algorithm computes with that property. Here, non-trivial means that it is a property that holds for some but not all partial computable funcctions.

Rice's Theorem is quite a dramatic result, because it ultimately guarantees that no fully reliable algorithm will ever be able to perfectly optimize a program for speed or size, no compiler will be able to verify the correctness of programs, etc.[8]

Computers are really finite The last, desperate attempt at solving the halting problem could be done by a self-appointed smart person that observes how computers are not really correct implementations of Turing machines. In particular, it could be noticed how memory in a modern computer is actually finite, and thus a modern computer is really a large state machine. A non-terminating program will eventually run out of new memory states to explore, and revert back to an older memory state. Thus by tracking all memory states seen so far by the program we can determine if the program does not halt after *enough steps* to explore all the possible memory states. But *how large of a state machine are we talking about*, and thus how many is *enough steps*? Consider a computer with just one gigabyte, which for practical purpose we will assume to be 10^9 bits. This means that we would need to perform at most 2^{10^9} steps, in order to check all possible bit configurations. Now 2^{10^9} is indeed a finite number, if one is into this kind of distinction, but as far as finite numbers go it is sufficiently big to make any concrete implementation impossibly slow. It is safe to assume that the civilization of the creator of the program will likely be dead well before the program finishes running. Whether we choose to use a computer with less memory will still be pointless, since even

[8]This is the permanent guarantee of employment for at least some programmers that was mentioned in a previous footnote. Yay!

for a computer with one kilobyte we would need to perform at most 2^{10^5} steps, which is still a prohibitively large number. Whoever tries to run such a program will not likely see the result within his or her lifetime.

Of course modern computers have far more memory, and programs are often connected to the Internet. This means that the actual program running is a large distributed program which state is far larger and distributed, further making any "finite memory" argument completely unusable in practice.

5.1.3 Some equivalences and translations

Someone might wonder if it is possible to define some formalism that solves the halting problem by simply shifting the perspective. Perhaps there exists some new way to formulate new programming languages that do not suffer from this issue?

The basic formalisms of computing are two well-known theoretical programming languages, which are *Turing machines* on one hand and the *λ-calculus* on the other. Already during the previous century[9] it was discovered that these languages are **isomorph**, that is they can both express exactly the same programs, and neither can do something that the other cannot. This shocking result was brought about by the presence of translation routines that allow to translate *any* Turing-machine program in a semantically equivalent[10] λ-calculus program, and vice-versa.

Without diving into the details of these formalisms, it is possible to observe that we have not yet found a way to escape the circle. For example, we could notice that in our case, there exists a chain of translations that goes as follows:

```
λ-calculus => F# => our logic language
```

Now if we can show that we can implement the λ-calculus into our logic language, then we have "closed the circle". The grammar of the language is made up of three simple terms: *i*) identifiers (intuitively similar to variables), which are simply strings wrapped by `$`; *ii*) lambda

[9]For the perspective of future readers, we are talking about the 1900's.

[10]That is which performs exactly the same computation.

terms (intuitively similar to function declarations), which take the form `\$x -> TERM`; *iii*) applications (intuitively similar to function calls), which take the form `TERM | TERM`.

This syntax is defined in our language as:

```
Data [] [] "$" [<<string>>] Priority 10 Type Id
Data [] [] "\" [Id Arrow Term] Priority 9 Type Term
Data [] [Term] "|" [Term] Priority 8 Type Term
```

Of course an identifier is also a term, thus:

```
Id is Term
```

The arrow is not strictly needed, but it is aesthetically pleasing in that it helps distinguish a function arguments from its body:

```
Data [] [] "->" [] Priority 0 Type Arrow
```

Execution[11] of a program within the calculus is quite simple: whenever we apply a lambda term (the function being called) to another term (the argument), then the result *is the body of the function called where the function parameter has been replaced by the value of the argument.*

Execution of any term which is not the application of a lambda term to something else simply does nothing:

```
---------
$x => $x

----------------
\$x -> t => \$x -> t

------------------
$x | u => $x | u
```

Nested applications are solved from the right to the left:

```
w => w'
u | v => uv
uv | w' => res
--------------------
(u | v) | w => res
```

Finally, applications of lambda terms to other, arbitrary terms instance *substitutions*. Substitutions are of the form `TERM with $x`

[11] This is just one of the many possible evaluation strategies for the λ-calculus, which is perhaps the most intuitive for a programmer with a traditional imperative background.

`as TERM`, indicating that we are replacing variable `$x` with some other term:

```
Data [] [Term] "as" [Term] Priority 6 Type Where
Data [] [Term] "with" [Where] Priority 5 Type With
```

To evaluate such an application, which is of the form `\$x -> t | u`, we first evaluate `u` into `u'`, and then replace `$x` with `u'` within `t`:

```
u => u'
t with ($x as u') => t'
-----------------------------------
\$x -> t | u => t'
```

If we find the variable we needed to replace with term `u`, then instead of the variable we return `u` itself:

```
  x == y
  ----------------------
  $y with $x as u => u
```

If we find a different variable than the one we needed to replace, then we return the variable unchanged because no replacement is possible:

```
  x != y
  ----------------------
  $y with $x as u => $y
```

If we find a redefinition of the variable we were replacing, then we stop the replacement procedure as the variable we were replacement has been shadowed and a new one becomes active in the new scope created:

```
  x == y
  ---------------------------------
  \$x -> t with $y as u => \$x -> t
```

If we find a variable definition for a new variable which is not the one we were replacing, then we proceed with replacement within the body of the function definition:

```
  x != y
  t with $y as u => t'
  ----------------------------
  \$x -> t with $y as u => \$x -> t'
```

If we find an application, then we perform the replacement within both terms of the application and return the resulting application:

```
  t with $x as v => t'
  u with $x as v => u'
  -------------------------------
  (t | u) with $x as v => t' | u'
```

Consider now the evaluation of a simple term such as:

```
-----------------------------
(\$"x" -> $"x") | $"y" => ?
```

The first step of evaluation tries to evaluate term `$"y"`, and then replace `$"x"` with the result of this evaluation:

```
$"y" => u'
$"x" with $"x" as u' => t'
-----------------------------------
(\$"x" -> $"x") | $"y" => t'
```

The evaluation of `$"y"` simply returns `$"y"` itself, because a variable always evaluates to itself:

```
$"y" => $"y"
$"x" with $"x" as $"y" => t'
-----------------------------
(\$"x" -> $"x") | $"y" => t'
```

Evaluation then proceeds to the next premise, which is a replacement of the right variable

```
$"x" == $"x"
t' := $"y"
---------------------------------------
$"x" with $"x" as $"y" => t'
-----------------------------------
(\$"x" -> $"x") | $"y" => t'
```

We can now begin the unwinding procedure, which after just one step returns:

```
(\$"x" -> $"x") | $"y" => $"y"
```

Which is precisely the answer we expected.

By implementing the λ-calculus then we have shown how on one hand our language *is not more powerful* than any of the well-known formalisms of programming, but on the other hand we have shown how our language *is not less powerful* than any of the well-known formalisms of programming.

This being exactly as powerful as the λ-calculus or Turing machines is known as the **Church-Turing equivalence**, and is the best

known way to define the expressive power of programming. The existance and apparent inescapability of the Church-Turing equivalence suggests that there is no preferred encoding which "unlocks" impossible problems such as the halting problem, therefore making them computable.

5.2 Conclusion

What we have seen so far leaves us with a new perspective on computation. No matter what language we choose, there seems to be a hard limit on what is possible to compute. This means that:

i) programming languages all represent, at their core, the same set of objects (the set of all programs); *ii*) some problems cannot be solved with an automated solution, independently of the programming language;

Why is the search for new programming languages interesting thus, if they allow us no more expressive power? In reality, programming languages have more or less expressive power, but not because of what they can compute, but because they allow a clearer interaction with the *humans* that use them. As A. Whitehead[12] said in "The Importance of Good Notation", in his book *An Introduction to Mathematics*:

> by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain.
>
> [...]
>
> One very important property for symbolism to possess is that it should be concise, so as to be visible at one glance of the eye and to be rapidly written.
>
> [...]
>
> It is interesting to note how important for the development of science a modest-looking symbol may be. It may stand for the emphatic presentation of an idea, often a very subtle idea, and by its existence make it easy to exhibit the

[12] A famous mathematician for as far as fame goes for mathematicians.

> relation of this idea to all the complex trains of ideas in which it occurs. For example, take the most modest of all symbols, namely, 0, which stands for the number zero. The Roman notation for numbers had no symbol for zero, and probably most mathematicians of the ancient world would have been horribly puzzled by the idea of the number zero. For, after all, it is a very subtle idea, not at all obvious. A great deal of discussion on the meaning of the zero of quantity will be found in philosophic works. Zero is not, in real truth, more difficult or subtle in idea than the other cardinal numbers. (...)

Thus programming languages make it easier to express some aspects of thoughts rather than some others, and by emphasizing concepts such as correctness or reliability we can dramatically change the impact of the language on the thought process of the programmer.

About the unsolvable problem, also here is some silver present. Even though we cannot provide a perfect solution to these problems, nothing forbids us from building partial solutions that allow for uncertainty. We can thus always build a program that can answer questions about termination, correctness, or equivalence between programs with answers taken from the set `yes`, `no`, and `unknown`. The ability to return `unknown` suddenly makes the program possible to build, therefore exchanging some precision with implementability.

The challenge that arises from this is thus that of reducing the number of programs for which our analyser returns `unknown` so that it is as small as possible. Not only do we wish to reduce the number of uncertain programs, we might also find it acceptable to have our analysis "give up" for programs where the flow of control is complex or hard to follow, in the assumptions that these programs, even when working, are not acceptable for being too confusing[13]. Theoretical frameworks such as dependent types, model checking, and abstract interpretation (just to name a few) are written with the goal in mind of standardizing the concepts of approximated analysis of computer

[13]If a program is confusing for a carefully built analyser, imagine what it does to the brain of Bob, a 20-something junior programmer who just came out of a coding bootcamp and sits two cubicles over.

programs, precisely with the goal of providing an incomplete, but still useful, solution to problems such as halting.

Chapter 6

Conclusions

The goal of this document was to make you acquainted with the concept of inference systems. You have learned various aspects of logic and inference system, which we used to give rise to definitions of concepts such as numbers, boolean expressions, etc. Thanks to this we have seen how logical inference is actually self-sufficient to define all sorts of well-known mathematical concepts, without having to either:

i) introduce these complex as primitive concepts, therefore making our formalism "heavier"; *ii*) give them up and try to not use them.

Defining mathematical concepts could have gone further into most existing known aspects of mathematics, but instead after a while we have taken another route: defining whole programming languages as instances of logical inference. As a stepping stone we have defined a series of useful helpers, which are data structures such as lists or binary search trees. Then we have gone further to the definition of a small, yet whole, programming language.

This long trip has shown us not only the power of logic as a foundational tool for both mathematics and programming, but it has also shown the great potential of our logic-based language in the definition of programming languages.

Appendices

Appendix A

A not so small imperative language

A compiler is a software which translates a program written in a high-level language into machine code. Given the complexity of this task, a compiler is usually split into modules which accomplish different steps during the translation process. During the translation process the compiler checks also that the targeted code is correct as for the syntax and semantics of the language. Usually the compiler is made of the following components:

- The **Lexer** converts a sequence of characters into a sequence of *tokens*, which are meaningful character strings. It analyses that a sequence of symbols form a valid "word" of our language. For example in Java language the sequence `newObject` would be a valid token for an identifier, while `1234xyz` would be rejected by the lexer.

- The **Parser** checks the syntax of the program, i.e. that it checks that the grammar rules of our language are followed. It checks that the "sentences" of our language are well formed. For example in Java language the statement `int x = 35;` is a valid statement while `x + 3` is not a valid statement according to the language grammar.

- The **Type Checker** checks the semantics of the program, i.e.

that what we write has a meaning. This check can be performed either at compile time (*statically typed* languages such as C, C++, Java, C#, F#, ...) or at runtime (*dynamically typed* languages such as Javascript, Python, Lua, ...). Dynamically typed languages include in the output code the type verification code which is run every time a statement is executed.

- The **Intermediate code generator** defines the operational semantics, i.e. how the language statements and operators actually behave, and usually translates the high-level code to an *intermediate code* which is close to assembly.

- The **Target code generator** translates the intermediate code into the targeted architecture code (x86,x64,ARM,...).

Interpreted languages such as Java do not include the last component because they generate bytecode which is interpreted by a Virtual Machine. The lexer, parser, and type checker are defined as compiler *front-end*, while the intermediate and target code generators are defined as compiler *back-end*. All the components except the target code generator are independent on the CPU architecture. Interpreted languages trade off performance for independence from the CPU architecture.

A.0.1 Operational semantics of C--

In this section we will define the operational semantics of a language derived from C called C--. We will assume for brevity that the input program has already been correctly type-checked and here we just define the behaviour of the language structures and operators.

Informally a C-- program is a set of statements, which can act on the memory and alter it. It is not possible to make function calls nor define new functions. The language has 4 native types (`int`, `bool`, `double`, and `string`). The language has 3 control structures: *if-else*, *while*, and *for*. The language supports variable scoping, which means that you can re-define a variable with an existing name within a control structure body. The new variable will replace the existing one while inside the body. When the program exists the body the previous

variable is restored. Besides variables define inside bodies are visible only within the body itself or other nested bodies starting from the current one. The evaluation of the program returns the state of the memory, which is a table containing a variable name paired with its current value.

Example 1. Consider the following code snippets:

```
int z ;
int y;
z = 4;
y = 1;
if (z > 0)
  int y;
  y = 2;
  z = z + y;
  y = y + 1;
```

```
int z ;
int y;
z = 4;
y = 1;
if (z > 0)
  z = z + y;
  y = y + 1;
```

In the first snippet the variable `y` definition is replaced inside the `if` body according to the scoping rule. The state of the memory at the end of the program in the first snippet is:

VARIABLE	VALUE
z	6
y	1

while at the end of the second snippet is:

VARIABLE	VALUE
z	5
y	2

Implementing scoping requires that we use a list of *symbol tables*. A *symbol table* is a dictionary where the key is the variable name (which is indeed unique) and the value contains the value of the variable.

Each time we enter a new body we insert in the list an empty symbol table. When we exit the body we delete the symbol table. At the end of the program we will have a single symbol table containing only the global variables.

A.0.2 Evaluating a C-- program

A C-- program will take as input an empty memory state and a list of expressions (or statements). We define then the following keywords:

```
Keyword [] "$m" Dictionary Priority 300 Class SymbolTable
Keyword [SymbolTable] "nextTable" [TableList] Priority 10 Class
    TableList
Keyword [] "nilTable" [] Priority 500 Class TableList
Keyword [SymbolTable] "add" [Id Value] Priority 100 Class
    DictionaryOp
Keyword [SymbolTable] "lookup" [Id] Priority 100 Class DictionaryOp
Keyword [SymbolTable] "contains" [Id] Priority 100 Class DictionaryOp

Keyword [Expr] ";" [ExprList] Priority 5 Class ExprList
Keyword [] "nop" [] Priority 500 Class ExprList

Keyword [] "program" [SymbolTable ExprList] Priority 0 Class Program
```

Evaluating a C-- program means evaluating a series of statements separated by ";" and update the list of symbol tables at each evaluation. We can build an evaluation rule which takes the current statement, evaluates it, and then recursively call itself on the rest of the program. The evaluation rule returns the list of the updated symbol tables and the value returned by the evaluation.

```
Keyword [] "eval" [TableList Expr] Priority 0 Class RuntimeOp
Keyword [] "evalResult" [TableList Value] Priority 0 Class
    EvaluationResult
```

The list of values available in C-- are the following:

```
Keyword [] "$i" [<<int>>] Priority 300 Class Value
Keyword [] "$d" [<<double>>] Priority 300 Class Value
Keyword [] "$s" [<<string>>] Priority 300 Class Value
Keyword [] "$b" [<<bool>>] Priority 300 Class Value
Keyword [] "$void" [] Priority 300 Class Value
```

Starting from the `program`, we call the evaluation function on the program code. The program evaluation returns the updated global symbol table. The value returned by the program evaluation is `void`

because a program is made by a sequence of statements that alter the memory, but they do not return any result.

```
eval (memory nextTable nilTable) code => evalResult (memory'
    nextTable nilTable) $void
-------------------------------------------------------------
program memory code => memory'
```

The evaluation rule for a list of statements is the following:

```
eval tables a => evalResult tables' $void
eval tables' b => res
---------------------------------------
eval tables (a;b) => res

-----------------------------------------
eval tables nop => evalResult tables $void
```

We evaluate the head of the list, which possibly returns a modified list of symbol tables and returns `void` (we are evaluating a statement). We stop when we find the end of the sequence of statements.

In the following sections we will implement the evaluation rules for all the possible statements in the language.

A.0.3 Variable definition

C-- requires that the variable is defined before assigning a value to it. We define an operator called `variable` which takes a type followed by an identifier. The type are keywords themselves:

```
Keyword [] "t_int" [] Priority 500 Class Type
Keyword [] "t_double" [] Priority 500 Class Type
Keyword [] "t_string" [] Priority 500 Class Type
Keyword [] "t_bool" [] Priority 500 Class Type
```

An identifier is a string marked with the unary operator `$`:

```
Keyword [] "$" [<<string>>] Priority 300 Class Id
```

The rule for variable definition calls the rule for adding an element into the symbol table. The correct symbol table which must store the variable, according to the scoping rules defined above, is the symbol table of the current scope, which is the first element of the symbol table list. All the variables are initialize with the default value `$void`.

```
Keyword [SymbolTable] "defineVariable" [Id] Priority 300 Class
    MemoryOp
```

```
symbols defineVariable id => symbols'
-------------------------------------------------
eval (symbols nextTable tables) (variable t id) => evalResult (
    symbols' nextTable tables) $void


  symbols add id $void => symbols'
  ------------------------------------
  symbols defineVariable id => symbols'
```

A.0.4 Variable assignment

Variable assignment has the form `$id = expression`.

```
Keyword [Id] "=" [Expr] Priority 10 Class Expr
```

The left side of the assignment is called *lvalue*, and in our language can be only a variable. The right side of the assignment is called *rvalue* and can be an arithmetic expression, a boolean expression, or another variable. The evaluation rule must evaluate the expression, look up the variable **in the whole symbol table list** taking the first occurrence, and write the value in the table. Before that we have to write the evaluation rules for rvalues.

Atomic values

The simplest case of rvalues is an atomic value. Atomic values in C-- can be integer, double, string, or boolean values. `$void` value cannot be assigned to variables, nor be an expression. The evaluation rules for atomic values simply return the values themselves, without altering the symbol tables:

```
-------------------------------------------------
eval tables ($i v) => evalResult tables ($i v)

-------------------------------------------------
eval tables ($d v) => evalResult tables ($d v)

-------------------------------------------------
eval tables ($s v) => evalResult tables ($s v)

-------------------------------------------------
eval tables ($b v) => evalResult tables ($b v)
```

Arithmetic expression evaluation

An arithmetic expression is a composition of the operators +,-,*,/ whose left and right arguments can be recursively arithmetic expressions, variables or integer and double values. For example $1+3*5-4/6$ is made of + applied to 1 and $3*5-4/6$ whose arguments are $3*5$ and $4/6$. The evaluation rule simply evaluates recursively the arguments of an arithmetic operators and then combine the results. We will just show the evaluation rules for integer expressions, the rules for double expressions are the same. Also it is shown the + operator for string, which outputs the concatenation of two strings.

```
eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
arithmeticResult := <<val1 + val2>>
-----------------------------------------------------------------

eval tables expr1 + expr2 => evalResult tables'' ($i arithmeticResult
    )

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
arithmeticResult := <<val1 - val2>>
-----------------------------------------------------------------

eval tables expr1 - expr2 => evalResult tables'' ($i arithmeticResult
    )

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
arithmeticResult := <<val1 * val2>>
-----------------------------------------------------------------

eval tables expr1 * expr2 => evalResult tables'' ($i arithmeticResult
    )

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
arithmeticResult := <<val1 / val2>>
-----------------------------------------------------------------

eval tables expr1 / expr2 => evalResult tables'' ($i arithmeticResult
    )

eval tables expr1 => evalResult tables' ($s val1)
eval tables' expr2 => evalResult tables'' ($s val2)
arithmeticResult := <<val1 + val2>>
-----------------------------------------------------------------

eval tables expr1 + expr2 => evalResult tables'' ($s arithmeticResult
    )
```

Boolean expression evaluation

Boolean expressions can be obtained by either combining boolean operators or by using numeric comparison operators. In what follows we will give the evaluation rules only for integer comparison operators, the rules for double values are analogous. Equality and inequality operators can be applied to any value.

```
eval tables expr1 => evalResult tables' ($b val1)
eval tables' expr2 => evalResult tables'' ($b val2)
boolResult := <<val1 && val2>>
-----------------------------------------------------------------------

eval tables expr1 && expr2 => evalResult tables'' ($b boolResult)

eval tables expr1 => evalResult tables' ($b val1)
eval tables' expr2 => evalResult tables'' ($b val2)
boolResult := <<val1 || val2>>
-----------------------------------------------------------------------

eval tables expr1 || expr2 => evalResult tables'' ($b boolResult)


eval tables expr => evalResult tables' ($b val)
boolResult := << !val >>
-------------------------------------------------------
eval tables (!expr) => evalResult tables' ($b boolResult)

eval tables expr1 => evalResult tables' val1
eval tables' expr2 => evalResult tables'' val2
val1 == val2
-------------------------------------------------------
eval tables (expr1 equals expr2) => evalResult tables' ($b true)

eval tables expr1 => evalResult tables' val1
eval tables' expr2 => evalResult tables'' val2
val1 != val2
-------------------------------------------------------
eval tables (expr1 equals expr2) => evalResult tables' ($b false)


eval tables (expr1 equals expr2) => evalResult tables' ($b res)
boolResult := << !res >>
-----------------------------------------------------------------
eval tables (expr1 neq expr2) => evalResult tables' ($b boolResult)

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
boolResult := << val1 < val2 >>
-------------------------------------------------------
eval tables (expr1 ls expr2) => evalResult tables' ($b boolResult)

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
```

```
boolResult := << val1 <= val2 >>
-------------------------------------------------------
eval tables (expr1 leq expr2) => evalResult tables' ($b boolResult)

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
boolResult := << val1 > val2 >>
-------------------------------------------------------
eval tables (expr1 grt expr2) => evalResult tables' ($b boolResult)

eval tables expr1 => evalResult tables' ($i val1)
eval tables' expr2 => evalResult tables'' ($i val2)
boolResult := << val1 >= val2 >>
-------------------------------------------------------
eval tables (expr1 geq expr2) => evalResult tables' ($b boolResult)
```

Variable lookup

The last case of rvalue (and also of arithmetic and boolean expression) is the variable assignment. The rule must iterate the symbol table list until it finds a variable matching the variable name. Then it returns the value contained in that variable.

```
symbols contains ($name) => Yes
symbols lookup ($name) => val
-------------------------------------------------------------
eval (symbols nextTable tables) ($name) => evalResult (symbols
    nextTable tables) val


symbols contains ($name) => No
eval tables ($name) => evalResult tables' val
-------------------------------------------------------------
eval (symbols nextTable tables) ($name) => evalResult (symbols
    nextTable tables') val
```

Evaluation of the assignment

One could be prone to write the following rules for the assignment:

```
table contains id => Yes
eval table expr => evalResult table' val
table add id val => table''
-------------------------------------------------------
eval (table nextTable tables) (id = expr) => evalResult table''

table contains id => No
eval tables (id = expr) => res
-------------------------------------------------------
eval (table nextTable tables) (id = expr) => res
```

If the lvalue is contained in the current symbol table we update it with the result of the expression evaluation, otherwise we iterate the symbol table list. This implementation is however **wrong**. Indeed consider the assignment `z = z + y` in the example using variable scoping in Section A.0.1. The variable `z` is not in the current scope symbol table, so we recursively call `eval` dropping the head of the list. At this point we have a symbol table which contains `z`. Unfortunately, when we evaluate the rvalue, we cannot find `y` which was in the symbol table we dropped before the recursive call.

From this example we see that we need to keep the original list of symbol tables to be able to perform the correct lookup for rvalues. We rewrite the evaluation rule using a different rule `updateTables` which correctly perform the assignment by carrying both the current scope list and the original list with all the symbol tables.

```
symbols contains id => Yes
eval globals expr => evalResult globals' val
symbols add id val => symbols'
-------------------------------------------------------
updateTables globals (symbols nextTable tables) id expr => evalResult
     (symbols' nextTable tables) $void

symbols contains id => No
updateTables globals tables id expr => evalResult tables' $void
-------------------------------------------------------
updateTables globals (symbols nextTable tables) id expr => evalResult
     (symbols nextTable tables') $void


updateTables tables tables id expr => res
---------------------------------------
eval tables (id = expr) => res
```

The new rule evaluates the rvalue by using the global table instead of the partial symbol table carried by the recursive calls, so that it can correctly access all the variables defined so far.

A.0.5 Evaluation of the If-Else statement

This statement has the form `if (condition) then expr1 else expr2`.

```
Keyword [] "then" [] Priority 10 Class Then
Keyword [] "else" [] Priority 10 Class Else
Keyword [] "if" [Expr Then Expr Else Expr] Priority 10 Class Expr
```

The operational semantics is that if the boolean expression in the condition evaluates `true` we evaluate `expr1` otherwise we evaluate `expr2`. When we execute either of the two blocks, we have to add a symbol table for the scope of the new body we are going to evaluate. When the body has been evaluated, we discard the symbol table we added (because we are exiting the `if-else` scope) and keep the other possibly modified symbol tables. Thus we have two evaluation rules:

```
eval tables condition => evalResult tables' ($b true)
eval (empty nextTable tables) expr1 => evalResult (table' nextTable
    tables'') val
-----------------------------------------------------
eval tables (if condition then expr1 else expr2) => evalResult tables
    '' val

eval tables condition => evalResult tables' ($b false)
eval (empty nextTable tables) expr2 => evalResult (table' nextTable
    tables'') val
-----------------------------------------------------
eval tables (if condition then expr1 else expr2) => evalResult tables
    '' val
```

A.0.6 Evaluation of the While statement

This statement has the form `while (condition) expr`

```
Keyword [] "while" [Expr Expr] Priority 10 Class Expr
```

The statement evaluates the condition, then executes `expr` if the condition is true. At the end of the body evaluation we re-evaluate the condition and possibly execute the body again. If the condition is false we skip to the next statement. The considerations about the symbol tables done for the `if-else` statement are the same:

```
eval tables condition => evalResult tables' ($b true)
eval (empty nextTable tables) expr => evalResult (table' nextTable
    tables'') val
eval tables'' (while condition expr) => res
-------------------------------------------------
eval tables (while condition expr) => res

eval tables condition => evalResult tables' ($b false)
-----------------------------------------------------------
eval tables (while condition expr) => evalResult tables' $void
```

A.0.7 Evaluation of the For statement

This statement has the form `for (init condition step) expr`

```
Keyword [] "for" [Expr Expr Expr Expr] Priority 10 Class Expr
```

The statement evaluates once the `init` statement. After that it evaluates the condition, and if it is true `expr` is evaluated. At the end of the evaluation the `step` statement is evaluated and then the condition is evaluated again.

Since the `init` statement must be executed only once, we cannot recursively call the evaluation rule for the `for` statement directly. We then define a keyword `loopFor` which do the evaluations after the `init` statement has been evaluated.

```
Keyword [] "loopFor" [TableList Expr Expr Expr] Priority 0 Class
    RuntimeOp
```

The rule associated to this keyword evaluates the condition and the body, updating the symbol tables accordingly.

```
eval tables init => evalResult tables' $void
loopFor tables' condition step expr => res
-----------------------------------------------------
eval tables (for init condition step expr) => res

  eval tables condition => evalResult tables' ($b true)
  eval (empty nextTable tables) expr => evalResult (table' nextTable
      tables'') val
  eval tables'' step => evalResult tables3 val'
  loopFor tables3 condition step expr => res
  ------------------------------------------------------------------
  loopFor tables condition step expr => res

  eval tables condition => evalResult tables' ($b false)
  ------------------------------------------------------------------
  loopFor tables condition step expr => evalResult tables' $void
```

A.0.8 Conclusions

In this chapter we have seen how to define the operational semantics for a programming language. Usually this semantics is written on paper formally, but the elegance and clarity of the formal model is lost in the implementation phase. This is due to the fact that compilers are written using commercial general purpose programming languages which do not provide constructs to ease the implementation of such

formal models. Even when using functional programming languages which offer more expressiveness than imperative languages it is not possible to have a one-to-one mapping from semantics rule to the implementation. The meta-compiler instead offers a way to directly map these rules into code, making it more compact and clear. Of course we must trade off simplicity and compactness for performances, but static analysis techniques on the generated code might drastically improve the performance of the code generated by the meta-compiler.