CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

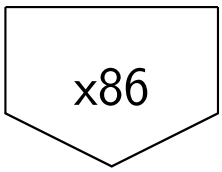
Chapter 1

Language Processing

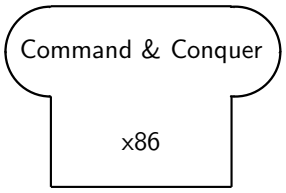
In this chapter, we take a look at programming languages from the point of view of programming tools called language processors. We develop an informal notation that allows to visualize the processing of programs by language processors, which leads to insights to the universality of certain language processing techniques.

1.1 Hardware and Native Programs

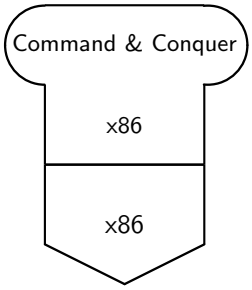
Throughout this course, we shall use so-called T-diagrams for visualizing the relationship between programs, tools and processors. T-diagrams were introduced by Earley and Sturgis in 1970 [ES70]. The processor of a computer is represented by a rectangle with a sharp point at the bottom. The type of processor is indicated in the rectangle. For example, a processor of the x86 family (IBM PC) like this.



A program is represented by a mushroom shape, where the foot indicates the language in which the program is written—its *source language*. For example, the program "Command & Conquer" written in machine code for the x86 processor is represented as follows.



When represented like this, programs can be placed on top of processors to indicate that the programs are run on the processors. A condition for this is that the source language of the program matches the processor type.



Programs that are written in machine code of a hardware architecture are called *native* to the architecture.

1.2 Translators

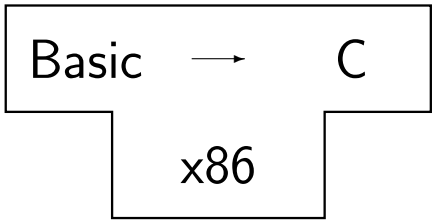
Translation from one programming language into another one is a central language processing technique. The concept of "abstraction level" of programming languages is debatable. High-level programming languages aim at describing software "close to" the application domain that is being modeled whereas low-level programming languages allow to describe solutions "close to" the actual physical processes that occur inside the computer's processors. Following this distinction, we distinguish between two kinds of translation tools:

- Compilers translate from a high-level language into a lower level language, and
- decompilers translate from a low-level language into a higher one.

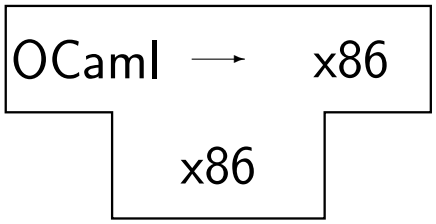
Special cases of compilers are assemblers, where the "high-level" language is assembler code and the low-level language is machine code, and disassembler, which are decompilers that translate from machine code to assembler code.

Translators translate programs written in one language—we call it the *from-language*—into programs written in another language—called the *to-language*. The corresponding T-diagram has the shape of a "T" (thus the name T-diagram)

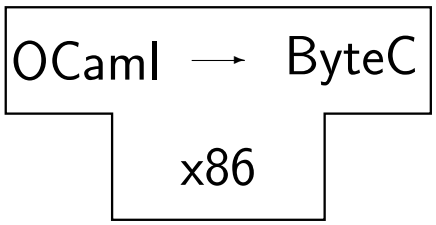
and indicates the from-language on the left and the to-language on the right. Of course, translators are themselves programs written in some language (its source language), which is indicated—as for all programs—in the lower part of the "T" symbol. For example, a compiler that translates Basic programs to C programs written in x86 machine code looks like this.



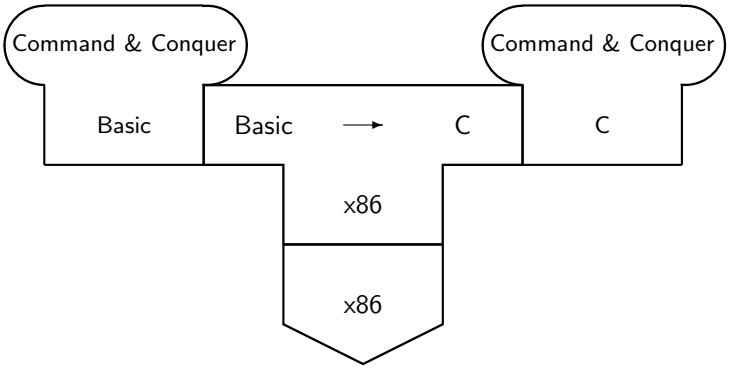
The OCaml compilation system that we will be using sports a native compiler in the following format.



There is also a byte-code OCaml compiler that generates more portable code as target program (in a similar spirit as JVM bytecode), but the overheads may cause an order of magnitude slowdown.

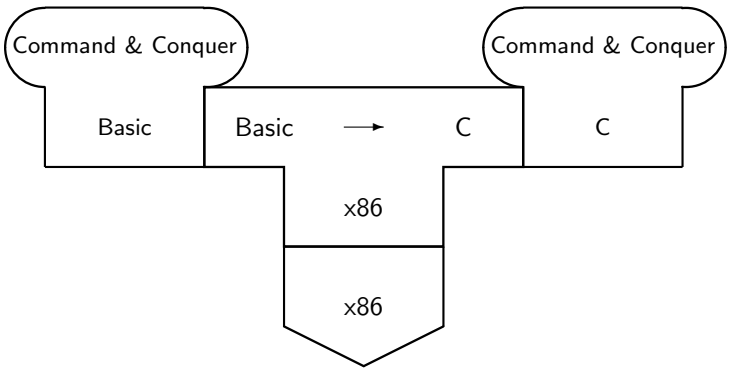


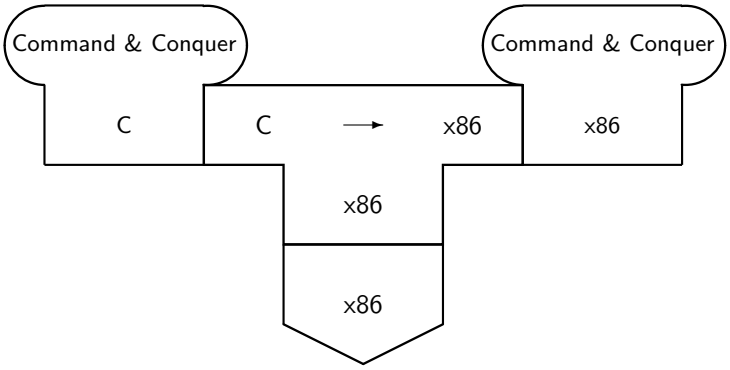
Translators are used to translate programs from one language to another. This process is being executed on a computer by running the translator program. We can indicate the process of translation by a side-ways combination of symbols.



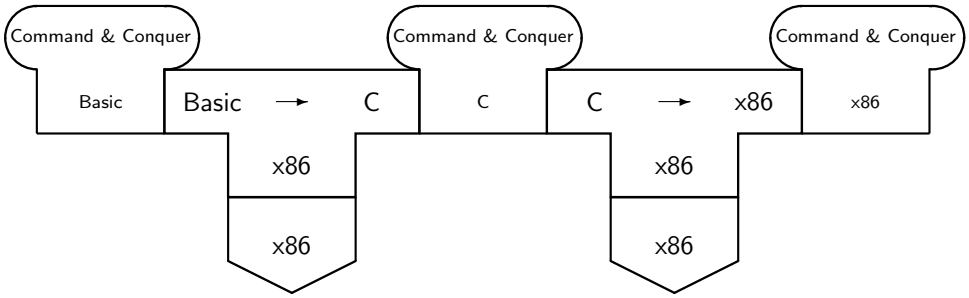
Again, this works only if the languages at the connection between the programs, processor and the translator match. The language of the program that we want to translate needs to match the from-language of the translator (here Basic), the language of the desired output of the translation needs to match the to-language of the translator (here C), and finally, the source language of the translator needs to match the processor type (here x86).

Of course, the output of one translator can serve as input of another one, leading to a chain of translation steps. For example, the program "Command & Conquer" can be translated from Basic to C, and then from C to x86 machine code, as highlighted next.



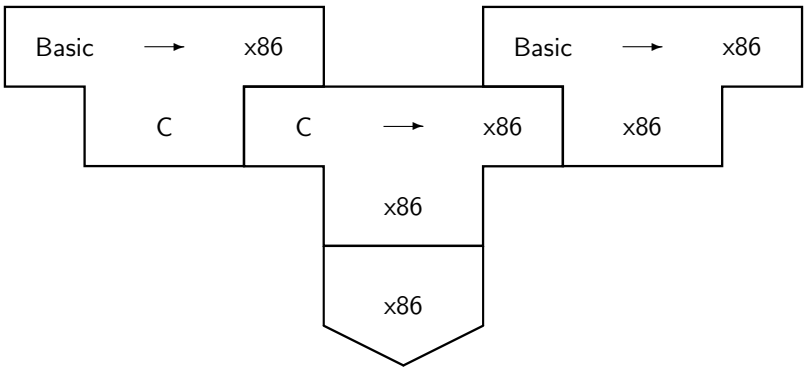


For convenience, we often combine two sequences of compilation into one single T-diagram.



Note, however, that this diagram denotes two distinct compiler runs, namely first the run of the Basic-to-C compiler to compile the Basic program to C, and then the run of the C-to-x86 compiler to compile the resulting C program to x86 machine code.

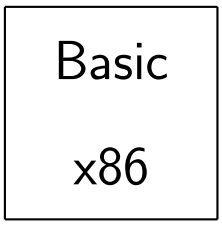
Translators are programs, which need to be executed on a processor. Being sophisticated pieces of software, translators are typically written in a high-level language and then compiled to machine code. The diagram below describes the compilation of a Basic-to-x86 compiler from C to x86 machine code.



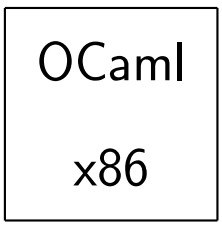
1.3 Interpreters

Apart from translators, the second kind of language processing tool are interpreters. An interpreter is a program that take another program (and the other program's input data) as input and directly executes it, typically by reading and executing single instructions or groups of instructions at a time.

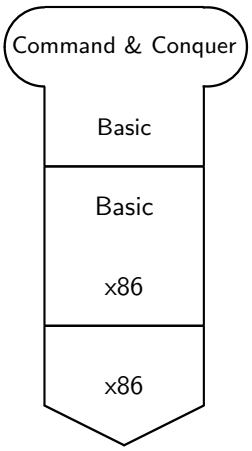
We represent an interpreter by a rectangle, where the language being interpreted—the *target language* of the interpreter—is indicated on the top. The language, in which the interpreter is written—its *source language*—is indicated on the bottom. For example, a Basic interpreter written in x86 machine code looks like this.



There is also an OCaml interpreter that can be run either interactively (where the commands are interpreted as they are being typed) or in batch mode (where the commands are taken from a file). This OCaml interpreter would be denoted, as follows:

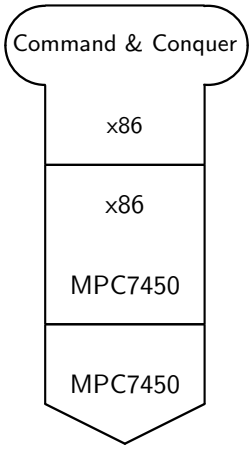


Now programs can be executed using an interpreter, which in turn is executed by a processor. This works when the target language of the interpreter is the same as the source language of the program, and when the source language of the interpreter matches the processor type. For example, a program "Command & Conquer" written in Basic can be executed as follows.



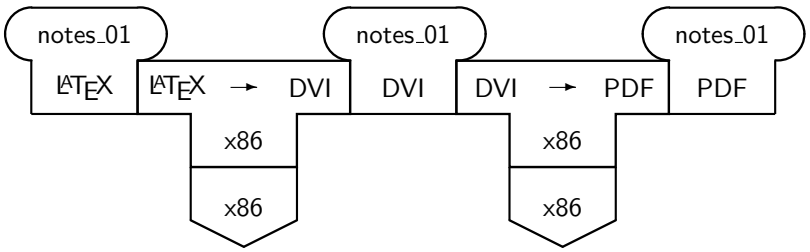
Note that the vertical stacking of language processors denotes *single program executions*. In the diagram above, the "Command & Conquer" program in Basic, the Basic interpreter, and the x86 machine all "run" at the same time, at different levels of abstraction.

The process of interpretation is not confined to high-level languages. Machine code can be interpreted as well, a process called hardware emulation. Hardware emulation allows us to run programs that are written in machine code for one machine on another machine. For example, an interpreter for x86 machine code written in PowerPC (MPC7450) machine code can be used to run the program "Command & Conquer" written in x86 machine code on a PowerPC.



1.4 Combining Compilation and Interpretation

The process of translation and interpretation is not confined to conventional programming languages. Other kinds of data can be compiled from one format to another, or interpreted by programs. For example, this text was written in the typesetting language LaTeX, translated into a format called DVI (device-independent format), and from there to the PDF format, all on a Ubuntu Linux x86 Virtual Box platform, as shown below.

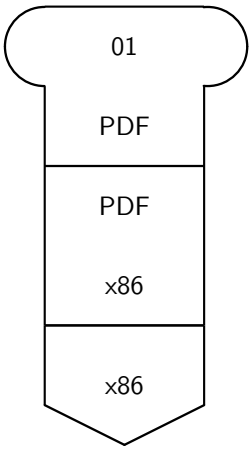


The corresponding sequence of Unix instructions that the author carried out are as follows:

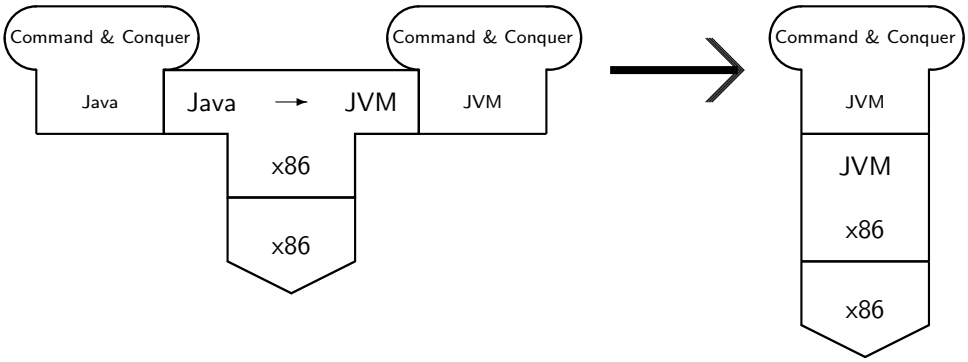
```
henz$ latex notes_01.tex
henz$ dvipdf notes_01.dvi notes_01.pdf
```

The first command causes the typesetting program LaTeX to translate the source file `01.tex` to the DVI file `01.dvi`, and the second command translates `01.dvi` to the PDF file `01.pdf`.

The resulting PDF file can be viewed using a program that reads the PDF data and interprets it, displaying the data visually, as indicated below. The company Adobe that developed the PDF format provides such an interpreter, called Acrobat Reader.

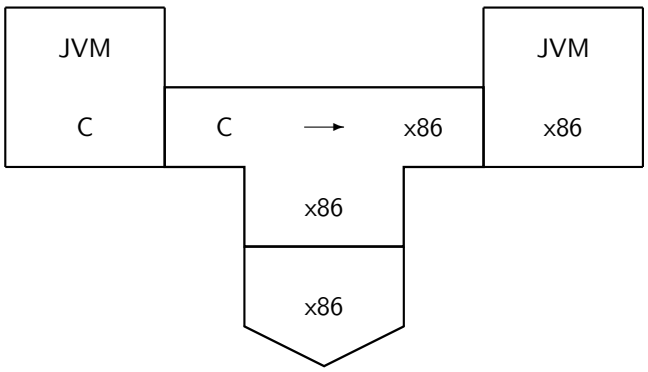


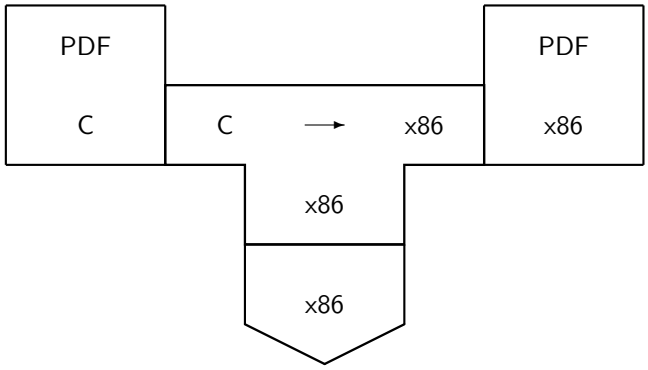
An example for combining compilation and interpretation for the execution of computer programs is the typical execution of Java programs. Java programs are usually compiled into a format called Java Virtual Machine code. Usually, this code is not executed directly by existing hardware, but interpreted using a Java Virtual Machine. The picture below illustrates the process.



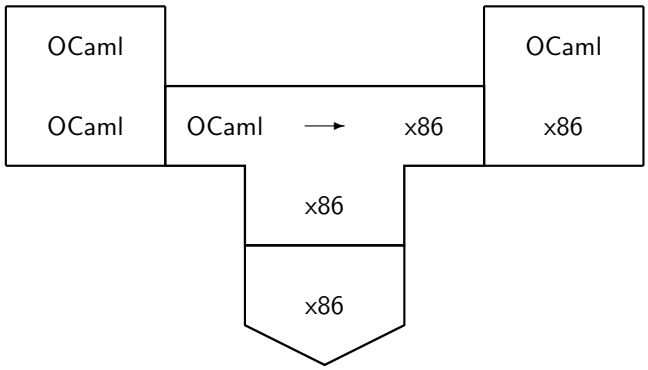
Compiling Interpreters

Just like translators, interpreters are programs, typically written in higher programming languages. In order to get an executable, we need to compile them. As examples, we give the compilation of a JVM from C to x86 machine code, and the compilation of Acrobat Reader (assuming it is written in C).

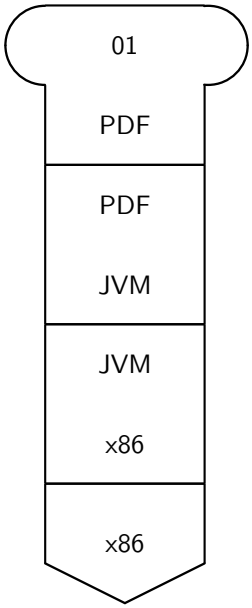




It is possible to write an interpreter in the same language that it is meant to interpret, before compiling it to native code. For this, we need a compiler for the said language to some native code.



Finally, it is possible to combine different interpreters with each other by "stacking" them. Let us say Acrobat Reader is written in Java instead of C. We can compile it to JVM code, and use a Java virtual machine to run it. The picture below shows two interpreters in action to view a PDF file.



1.5 Summary

We have seen two distinct families of language processing tools. Translators translate programs written in one language to another language, and interpreters execute programs on a given underlying architecture. This architecture can consist of a chain of interpreters, finally ending on a hardware platform. The notation of T-diagrams allows us to visualize the different language processing steps involved in executing computer programs.

Chapter 2

Inductive Definitions

2.1 Motivating Examples

It is common in the study of programming languages to define a set by a collection of rules that specify the members of the set. Each rule has zero or more premises, or requirements, and one conclusion.

Example 2.1 (Numerals, first attempt) *We may define the set of unary numerals (i.e., numerals in base 1) for the natural numbers as follows:*

- *Zero is a numeral.*
- *If n is a numeral, then $Succ(n)$ is also a numeral.*

Equivalently, we might say that the set Num of numerals is defined by the following rules:

- *$Zero \in Num$.*
- *If $n \in Num$, then $Succ(n) \in Num$.*

Observe that in each formulation the first rule has no premises, whereas the second has one. Examples for elements of Num are:
$Zero$ and $Succ(Succ(Zero))$.

Example 2.2 (Binary trees, first attempt) *We may define the set of binary trees by the following rules:*

- *The empty tree, Empty, is a binary tree.*
- *If t_l and t_r are binary trees, then $Node(t_l, t_r)$ is a binary tree.*

Equivalently, we might say that the set Tree of binary trees is defined by the following rules:

- *$Empty \in Tree$.*

- *If* $t_l, t_r \in$ *Tree, then* $Node(t_l, t_r) \in$ *Tree.*

The first rule has no premises; the second has two. Examples for elements of T are:
Empty and $Node(Empty, Node(Node(Empty, Empty), Empty))$.

Notice the similarity between these two examples. The empty tree is analogous to the number 0, and the node formation operation is analogous to the successor operation (except that it has two predecessors!).

Excursion: Defining Sets in Java and then OCaml

We can directly translate the concept of defining sets by rules into Java. Our examples translated to Java look like this:

```
interface Num {}
class Zero implements Num {}
class Succ implements Num {
   public Num pred;
   Succ(Num p) {pred = p;}
}

interface Tree {}
class Empty implements Tree {}
class Node implements Tree {
   public Tree left, right;
   Node(Tree l,Tree r) {left = l; right = r;}
}
```

These class definitions introduce the types `Num` and `Tree`, respectively, from a given set of constructors. Each constructor defines a rule for membership in that type. The (implicitly defined) constructors `Zero()` and `Empty()` have no arguments; this corresponds to a rule with no premises. The constructor `Succ` has one argument, corresponding to the single premise in the inductive definition; the constructor `Node` has two arguments, corresponding to the two premises in the inductive definition. We can construct the example instances of type `Num` by

```
Num my_num = new Zero();
Num my_other_num = new Succ(new Succ(new Zero()));
```

and the example instances of type `Tree` by

```
Tree my_tree = new Empty();
Tree my_other_tree =
   new Node(new Empty,
            new Node(new Node(new Empty(),
                              new Empty()),
                     new Empty()));
```

For the OCaml language, we can directly define algebraic data type for our sets. In the case of `Num` and `Tree` types, we can define the following new data types (inductively):

```
type num =
  | Zero
  | Succ of num

type tree =
  | Empty
  | Node of tree * tree
```

We can also define specific values by using a `let` construct which gives a name to some value. For example:

```
let my_num = Zero
let my_other_num:num = Succ (Succ Zero)
let my_tree =
  Node(Empty,
       Node(Node(Empty,Empty),
            Empty))
```

Types do not need to be declared for each expression that we wrote as there is a type inference process used by the language processor. However, if the user desires, he/she could ascribe a type to a given expression using he notation `(exp:ty)`. This declares that expression expression `exp` has type `ty`. It act as a documentation to remind us of the type of a value, and it is also used by the type system to check for type consistency.

Exercise 2.1 *Give a collection of rules defining the set of strings over characters. Give the corresponding Java class definitions (without using Java's* `string` *class) for the same set. Similarly, do the same thing for OCaml.*

2.2 The Extremal Clause

When we say that a set is defined by a set of rules, what precisely do we mean? Which set do we consider to be defined by those rules? To see why this is an important question, consider the set

$$\begin{aligned} \mathit{StrangeNum} &= \{\mathit{Zero}, \mathit{Succ}(\mathit{Zero}), \mathit{Succ}(\mathit{Succ}(\mathit{Zero})), \ldots\} \\ &\quad \cup \{\infty, \mathit{Succ}(\infty), \mathit{Succ}(\mathit{Succ}(\infty)), \ldots\} \end{aligned}$$

where ∞ is an arbitrary new symbol. Observe that $\mathit{Zero} \in \mathit{StrangeNum}$; and that if $n \in \mathit{StrangeNum}$, then $\mathit{Succ}(n) \in \mathit{StrangeNum}$—that is, *StrangeNum* meets the requirements of the rules we gave to define the set *Num*. This means

that the rules alone are not sufficient to pick out the intended set *Num*, since the strictly bigger set[1] *StrangeNum* also satisfies these same rules.

To use a set of rules to define a set, we must say something more than just that the set must obey these rules. What more is needed? We need an extremal clause that states that *nothing else is in the set except those elements that are required to be there by the rules.* This may sound like a bit of legalese, but mathematically it is essential to include the extremal clause, for otherwise the rules do not determine a unique set. Thus, the definition of *Num* should really be stated as follows.

Example 2.3 (Numerals, revised) *The set Num is defined by the following rules:*

- *Zero is a numeral.*
- *If n is a numeral, then $Succ(n)$ is also a numeral.*
- *nothing else is a numeral.*

Equivalently, we may say that Num is the least set that fulfills the first two rules, by which we mean precisely that nothing else is in the set except as is forced by the rules. Here "least" refers to the subset relation over sets; if X is the least set that fulfills some rules, then for any set Y that fulfills the rules, we have $X \subseteq Y$.

To see that StrangeNum is ruled out by the extremal clause, observe that ∞ has no business being in the specified set because it is not forced to be in there by the rules. Observe that StrangeNum is not the least set that fulfills the first two rules because StrangeNum $\supsetneq$ Num and Num obeys these rules. Thus StrangeNum is not defined by the rules.

Similarly, we may revise the definition of binary trees by adding an extremal clause as follows.

Example 2.4 (Binary trees, revised) *We may define the set of binary trees by the following rules:*

- *The empty tree, Empty is a binary tree.*
- *If t_l and t_r are binary trees, then $Node(t_l, t_r)$ is a binary tree.*
- *Nothing else is a binary tree.*

Equivalently, we might say that the set Tree of binary trees is the least set defined by the following two rules

- *$Empty \in Tree$.*
- *If $t_l, t_r \in Tree$, then $Node(t_l, t_r) \in Tree$.*

[1] bigger in the sense that *StrangeNum* is a strict superset of *Num*

The extremal clause ensures that a collection of rules of the kind given above determines a unique set. In practice we do not explicitly state the extremal clause, but rather we state that the set in question is inductively defined by a given collection of rules. For example, we may say that the set *Num* is inductively defined by the two rules membership rules given in Example 2.1. In doing so we are implicitly stating that nothing else is to be a member of that set unless it is *forced* to be there by the rules.

2.3 Inductive Definition with Inference Rules

It is quite common to give an inductive definition by a set of inference rules. For example, we might say that the set *Num* is inductively defined by the following rules:

$$\frac{}{Zero \in Num} \qquad \frac{n \in Num}{Succ(n) \in Num}$$

Similarly, we might say that the set *Tree* is inductively defined by the following rules:

$$\frac{}{Empty \in Tree} \qquad \frac{t_l \in Tree \qquad t_r \in Tree}{Node(t_l, t_r) \in Tree}$$

The horizontal line plays the role of "if ... then ..." in our earlier presentations of the rules. In general, in an inductive definition of a set X, an inference rule of the form

$$\frac{x_1 \in X \quad \cdots \quad x_n \in X}{x \in X}$$

stands for the rule "if $x_1 \ldots x_n \in X$, then $x \in X$. Why do we say that the least set defined by a collection of rules is *inductively* defined by it? As the terminology suggests, the answer is that there is a close connection with reasoning by mathematical induction. Here's why. Suppose that we wish to prove that every binary tree t has a height h satisfying the following two requirements:

- The height of *Empty* is 0.
- If t_l and t_r have heights h_l and h_r, respectively, then the height of $Node(t_l, t_r)$ is $1 + max(h_l, h_r)$.

We call these two conditions the specification of the height of a binary tree. The question is this: how do we know that every binary tree in fact has a height? This might seem like an odd question at first, but consider that the *infinite* binary tree $Node(Node(\ldots,\ldots), Node(\ldots,\ldots))$ has no height in the sense specified! Luckily,

the extremal clause rules out such "infinite trees", which makes it possible to assign a height to each binary tree.

How do we prove that every binary tree has a height? By induction! We have to prove that for every binary tree $t \in \mathit{Tree}$, there exists a number h satisfying the specification of height given above. Given the inductive definition of binary trees, what might t be? By the first rule defining binary trees, t might be *Empty*. In that case, t clearly has a height, namely 0, in accordance with the specification. By the second rule defining binary trees, t might have the form $\mathit{Node}(t_l, t_r)$, where t_l and t_r are also binary trees. By induction we may assume that each of them has a height, say h_l and h_r, respectively. But then the height h of t is uniquely determined by the equation $h = 1 + \mathit{max}(h_l, h_r)$ as required by the specification. Since *Tree* contains no other elements than are given by these two rules, we have demonstrated that *every* binary tree $t \in \mathit{Tree}$ has a height h.

Height of Trees in Java

Here is another point of view on the same question. Let look at the definition of the `height` function for the Java type `Tree`.

```
interface Tree {
   public int height();
}
class Empty implements Tree {
   public int height() {return 0;}
}
class Node implements Tree {
   public Tree left, right;
   Node(Tree l,Tree r) {left = l; right = r;}
   public int height() {
      return 1 + Math.max(height(left),height(right));
   }
}
```

The question is: why does the call `height(t)` terminate for every `t` of type `Tree`? Once again, the proof is by induction on the structure of `t`. If `t` is an instance of `Empty`, then `height(t)` terminates returning 0, as required. If, on the other hand, `t` is an instance of `Node`, then inductively we may assume that `height(t.left)` terminates (returning h_l) and that `height(t.right)` terminates (returning h_r), from which it follows that `height(t)` terminates with $1 + \mathit{max}(h_l, h_r)$. This completes the proof.

Expressing height and parity functions in OCaml

For writing codes, we typically need to use recursive functions in OCaml. As functions are first-class values in OCaml, we use the same let annotation to

declare functions, as we would declare values. We may also choose to ascribe types to our parameters and results (to make the code more readable), or leave it unspecified out of convenience, as illustrated by two examples below.

```
let rec parity n =
  match n with
    | Zero -> 0
    | Succ m -> 1 - (parity m)

let rec height (t:tree) : int =
  match t with
    | Empty -> 0
    | Node (x,y) -> 1 + (max (height x) (height y))
```

The keyword `rec` indicates that we are defining a possibly recursive value/function. The pattern-matching construct `match` is a generalization of the the conditional expression. The conditional `if e then e1 else e2` can be viewed as a shorthand for:

```
  match e with
    | True -> e1
    | False -> e2
```

It matches its argument `e` to a series of patterns, and chooses the first matched branch to be executed with a value returned. Unlike a procedural (or statement-based) programming languages that operates by side-effects, OCaml is different in that it is an expression-oriented language that computes the value of each given expression before it is returned.

You can do a similar termination reasoning for the following question : why does the call `(height t)` terminate for every value `t` of type `tree`? Similarly, why does call `(parity n)` terminate for every value `n` of type `num`? They can be proven by an inductive proof.

We will give a survey on the basic features of OCaml in the next chapter.

Bibliography

[ES70] J. Earley and H. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13:607–617, 1970.