Interpretation and Compilation of Programming Languages

Draft lecture notes - please do not make public

João Costa Seco

PUBLISHED BY NO ONE YET

http://ctp.di.fct.unl.pt/~jcs

First printing, Somewhere in time

Contents

Preface

Programming languages are one of the most important tools in computer science. They allow the expression of meaningful computational solutions for all different sorts of problems. These lecture notes aims at introducing the fundamentals of design and implementation of programming languages, and present the main implementation techniques for interpretation and reasoning tools. It serves the course "Interpretation and Compilation of Programming Languages" from the "Mestrado Integrado em Engenharia Informática" programme at FCT NOVA, Lisbon.

Many of the concepts presented in this course are essential in a wider computational approach, as it is illustrated in the seminal text on computational thinking by Jeannette M. Wing [Win06]. Many common problems appear in the context of programming languages in a pungent way, and are distilled in very canonical terms in a compiler or interpreter. Concepts such as abstraction, parametrisation, specification, verification, state, are part of the computational framework one needs to resort in general.

This course follows a vertical structure, working from a more abstract setting to a very practical and concrete project implementation. It provides some degree of theoretical knowledge about computing models, to define the runtime and type semantics of a programming language, about the pragmatics of language design, which helps on defining practical syntax and choice of operations, and the practice to be able to build tools that analyse and produce executable artefacts. Our main goal is to deepen the knowledge about existing programming languages, by means of studying of their basic building blocks and construction mechanisms. Hal Abelson, author of [AS96], once wrote:

> *If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master.* (Hal Abelson, *in* the preface of [FW08b])

Programming languages are usually divided into classes, called language paradigms, like functional, imperative, object-oriented, logic, concurrent, etc., based on their main abstractions and mechanisms. The distinctions in this taxonomy are getting thinner, in a scenario where new languages are being defined and existing languages are being extended with a rich mixture of concepts, allowing the programmer to better tackle different kinds of problems. Notable examples can be found in the Scala programming language, that provides a sound mixture of object-oriented, functional and concurrent features; the C# programming language that include (lazy) data query language mechanisms (LINQ); or the latest versions of the Java programming language (that will also include functional features). We are also witnessing changes in the target languages, that vary from the classical assembly languages (x86), compiler intermediate languages (e.g. LLVM), virtual machine intermediate languages (cf. JVM and CLR), or the more recent browser-compatible *transpilers* like babel that target Javascript.

During this course we study the fundamentals of programming language construction, their execution and verification model, and transformation functions. It is crucial that each abstraction is studied in isolation, defined in a compositional way, and combined in a sound way. To achieve these goals, students are challenged to incrementally develop interpreter algorithms, which are the perfect vehicle for learning of programming language semantics, type systems, and compiling techniques.

Part One

1. Introduction

In these notes, language-based tools – interpreters, compilers and type checkers – are used as vehicles to motivate the principled design and construction of programming languages. This introductory chapter is divided in two separate parts. We start with a visit to the general features of a programming language, its trends and origins, and then describe the general architecture of interpreter- and compiler-like tools.

1.1 Programming Languages

Programming languages represent the core of all specification, programming, analysing, and data processing tools. They are essential tools to express solutions for computational problems, ranging from data storage problems, interfacing or data exchanging between servers, to pure calculation algorithms. They are conspicuous in compilers and interpreters, but they are also at the core of important applications like integrated development environments, browsers, web-service interfaces, or databases (query languages).

The study of programming languages is many times guided by the so-called programming paradigms, which are based on a certain number of characteristics. Programming languages can also be studied from the perspective of the nature of the values they manipulate and the operations they support. Common language paradigms are imperative, functional, object-oriented, logic, reactive, etc.. A common "base" categorisation is based on a division between imperative and declarative languages, where a variable in the language has one of two meanings, the mathematical meaning or the stateful meaning [Har12]. A more traditional division is based on the pragmatic manner how the languages are used. Imperative languages describe "how" results are computed, as in declarative languages it is promoted to describe algorithms by defining precisely "what" are the results (by means of functions or logical expressions).

Low-level (assembly) languages, and languages derived directly (cf. C), are naturally imperative, mapping variables to the state of the machine, and languages like LISP, derived from the initial mathematical models like the lambda-calculus are naturally declared as declarative.

Following a large variety of language and language characteristics, there is also a big variety of tools that use language and language-based techniques. Besides the interpretation, checking

and compilation of any language we may find many other tools. From browsers whose DOM is an abstract representation of a web page and integrate just in time Javascript compilers, to software validation tools that check protocol configurations in distributed systems. Many integrated development environments also provide language-based development-time verification for Javascript, Python, or Ruby programs, which are not statically typed by the corresponding interpreter or compiler.

All this topics motivate a syllabus on the understanding of programming language semantics as a way to increase the proficiency of software developers. There are many sources of information available in books and online, but we believe to have gathered them in a unique and useful way. As a development principle, we will study programs whose data are other programs, so we will first study how programs are represented in an abstract way, as an abstract data type. We then proceed by defining properties and algorithms over values of this sort of data-types (programs).

1.2 Compilers and interpreters

A programming language is a mathematical (formal) definition, but its real purpose is to specify actual artefacts that can be executed and produce effects in the real systems. In this section we will explore the answers to the following questions, all in the domain of programming language-based tools:

- What is an interpreter?
- What is a compiler?
- What's the difference between an interpreted language and a compiled language?
- What is a JIT compiler? and an optimising compiler?
- What is a source-to-source compiler?

Each tool is designed and implemented using the methods and techniques that we address in this course. Each one of these tools has a specific role in either the development, or runtime support of a software system.

Interpreters and compilers are essentially programs, whose input data, and many times also their output data, are other programs. In general, the actual behaviour of any kind of programs is defined by (or at least dependent on) its input. The abstract behaviour of a program is defined with relation to a conventional set of possible input values, formally defined by types and conditions, or informally defined by the programmers.

Software tools like interpreters and compilers define a class of tools where we can find verification tools (type systems), code instrumentation tools, debuggers, integrated programming environments, database management systems (SQL interpreters). To distinguish them quickly, interpreters can be seen as virtual machines that operate on instructions, whereas compilers are program translators that produce programs for other machines.

The set of possible input values of this kind of tools is defined by a programming language, the actual representation of a program can be textual, or visual (via diagrams or models). The behaviour of an interpreter or compiler is defined based on an abstract representation of such a program, based on the constituents of the language and the composition of their partial behaviours. Hence, we follow a principled development that starts with the definition of an abstract data type that captures the abstract representation of a program.

There are several ways of combining program transformation and verification tools. They vary in their architecture and in the kind of interaction and interfaces they have. A compiler, whose architecture is shown in Figure 1.1, transforms and optimizes a program written at a higher abstraction level (the source language) into a program written at a lower level of abstraction language (the target language). The target program is generated in a way so that it can be executed by a machine – either virtual or real – and whose instructions are of at a finer grain of detail and adapted to the machine's internal data structures. Consider the simplified representations of a compiler and

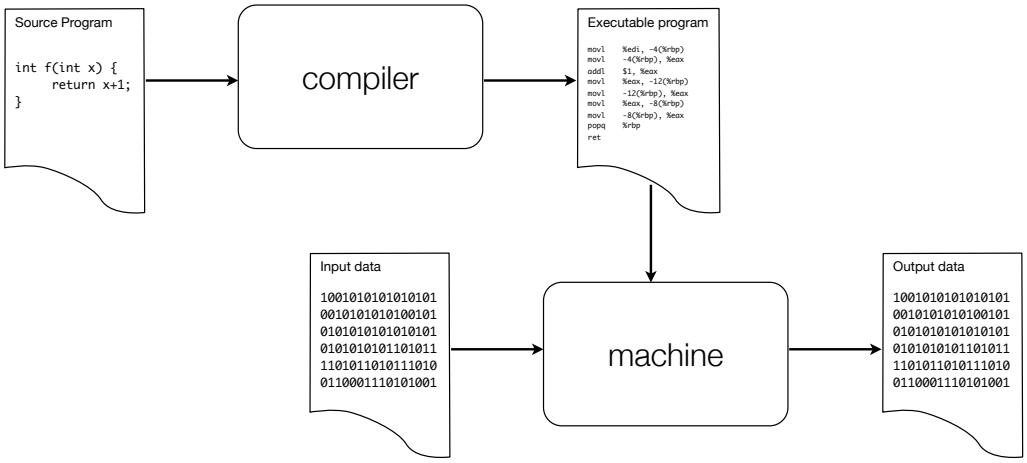


Figure 1.1: Compiler context diagram.

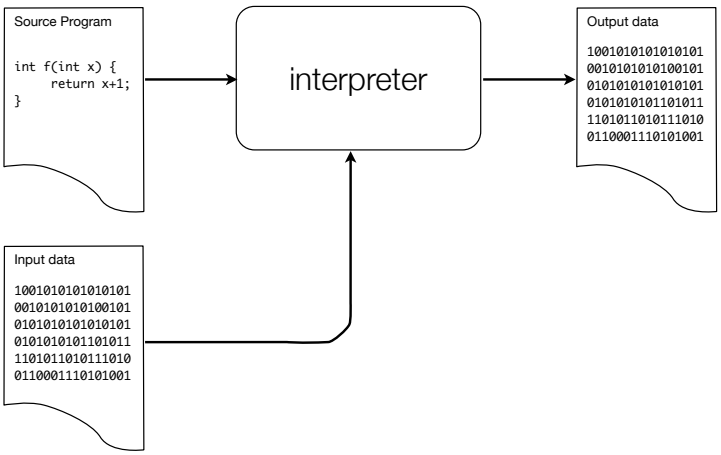


Figure 1.2: Interpreter context diagram.

an interpreter, in Figures 1.1 and 1.2 respectively, that depict their data transformation pipeline. This image can be extended without loss of generality to applications with richer interactions and interfaces, such as integrated development environments, partial compilation tools, etc..

Interpreters take as input the abstract representation of the input program (in the source language), and evaluate it with relation to additional input data, Figure 1.2. The output of the interpreter is the output produced by the program being executed. A more sophisticated, and realistic, combination of these concepts is present in the Java Virtual Machine (JVM), or the Common Language Runtime (of the .NET framework). These architectures combine compilation and interpretation of languages. They include a compiler, that translates Java programs to an intermediate language (as depicted in Figure 1.4), which is interpreted by the virtual machine. At that level, the virtual machine may resort to a compiler (*just-in-time* compiler), that transforms bytecode into native machine code that gets directly executed by the physical machine, see Figure 1.3. Other popular architectures consist in the integration of interpreters in other kinds of tools, like a Javascript interpreter in a browser, or a SQL interpreter in a database management system. The integration of a compiler into web based tools is also a current approach when developing new languages that use

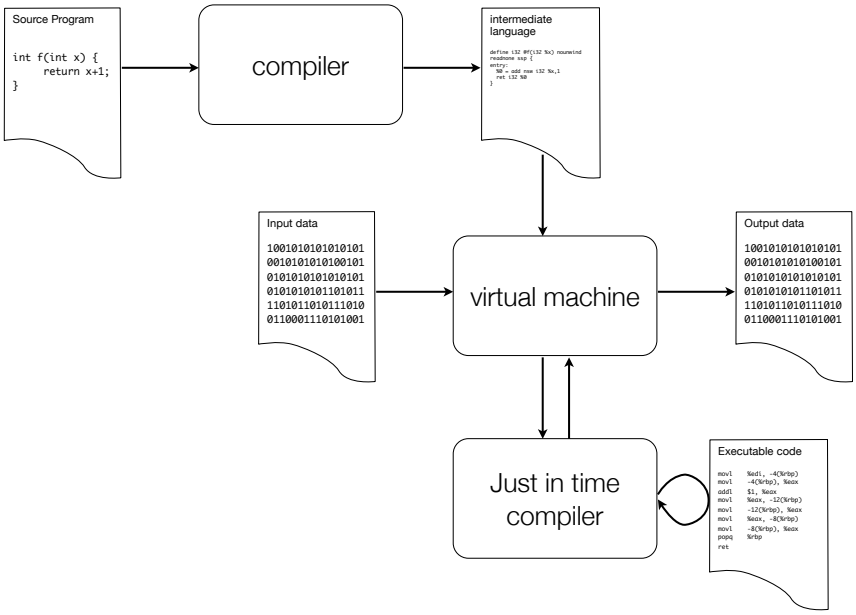


Figure 1.3: Interpreter with JIT compiler.

```
int f(int);
  Code:
    0:   iload_1
    1:   aload_0
    2:   getfield #2;
    5:   iadd
    6:   ireturn
```

Figure 1.4: Sample intermediate language (JVM Bytecode)

Javascript as an intermediate language. Popular projects include TypeScript[1] started by Microsoft, Dart[2] started by Google, Coffescript[3] inspired by other languages like Ruby, Python and Haskell, or the purely functional language Elm[4] used to develop client-side applications in a strongly typed and declarative style.

There is also a JIT compiler in the *Chrome V8* engine that compiles Javascript into native code before executing it and uses efficient garbage collection algorithms to speed up programs. The same engine is then used in projects like the MongoDB and CouchBase databases and the Node.js runtime environment[5].

1.2.1 Compilers vs. Interpreters

The discussion between interpreted and compiled languages, statically and dynamically typed language is a very common one. One that can be guided by different perspectives, some of those we list below:

[1] see `http://typescriptlang.org/` for more details.
[2] see `https://www.dartlang.org` for more details.
[3] see `http://coffeescript.org` for more details.
[4] see `http://elm-lang.org` for more details.
[5] see `https://en.wikipedia.org/wiki/Chrome_V8` for more details.

Efficiency

Comparing interpreted and compiled languages tends to go in favour of compiled languages. The execution of compiler generated code runs faster than the interpretation of the same code. Compilers process the code "offline", hence can take advantage of optimisation and verification opportunities resulting from algorithms that one cannot afford running at runtime. Besides that, the fetch and execution cycle is hardware based, hence faster. Interpreted languages are also, usually, dynamically typed, which means spending more time spent at runtime checking the true nature of operations' operands. Static typing offers an opportunity for faster execution of compiler generated code because runtime checks can usually be skipped. Languages like Javascript are nowadays being compiled at loading time, and transformed in a form of bytecode, makes them faster in some engines (e.g. Chrome and Node).

Safety

Interpreted languages are sometimes tagged as unsafe, because errors are only detected at runtime (trapped errors). However, the combination of compiler checks and the absence of runtime checks can also be gateway to build unsafe languages, which allows unexpected errors to occur (e.g. `C`, and even `Java` and the use of `Object` type). Languages like `OCaml`, `Haskell`, and `Scala` are languages that are many times interpreted (and can also be compiled), but combine it with a strong static type verification and code optimisation (e.g. *tail recursion*). The discussion about type safety, and its grey areas will be addressed in later stages of the course.

Flexibility

Interpreters usually provide a more flexible development process, taking advantage, for instance, of the dynamic loading of modules, allowing for the immediate update of new code in a running installation. Moreover, the use of an intermediate language provides code portability and cross-compilation features. In cases like `Java` and `C#`, it allows the compiled code to be executed in any platform (for which there is a bytecode interpreter, sometimes equipped with a JIT compiler).

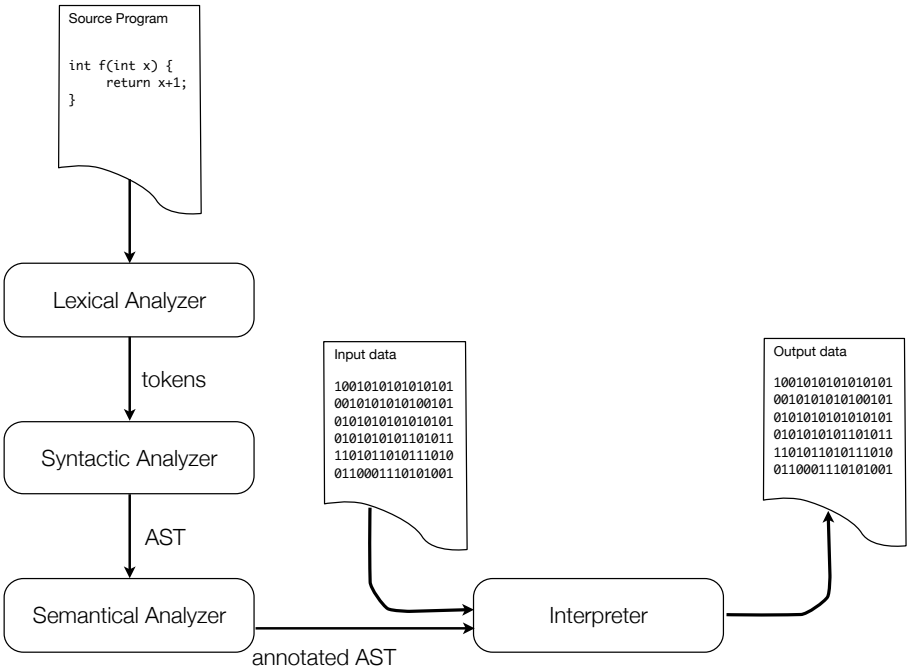


Figure 1.5: The architecture of an interpreter.

1.2.2 Software Architecture of a Language-based tool

Compilers and interpreters share some internal components and parts of their inner structure, namely the ones that treat the input data, the *front-end*. Figure 1.5 depicts the typical structure of an interpreter for a textual programming language. The first steps are a lexical analysis, where a stream of characters is transformed into a stream of recognisable tokens, which is then followed by a syntactical analysis, which then produces an abstract representation of a program. Abstract representations are the convenient format to design algorithms that interpret, analyse, and transform programs. At this stage, it is possible to analyse the semantics of a program, based on the constructions of the source language, and produce a annotated representation (e.g., with types). The abstract representation can then be evaluated on some given input data, the result of executing the interpreter is the program's output data.

The internal architecture of a compiler consists of a *front-end* layer, similar to the one found in interpreters, and two more layers. An intermediate layer, and a *back-end*. The intermediate layer of a compiler comprises the treatment of an intermediate language, which acts as a pivot in the architecture of a compiler. This split between a *front-end*, and a *back-end*, allows for, with a unique tool, to have a compiler family. This highly contributes to the portability between hardware architectures and also integration of different languages. Notable examples are `gcc`, `clang`, and the `.NET` framework and compiler family.

The use of visual languages, where programs are defined by interconnected blocks of programs, varies only on the front-end. Usually, an editor is capable of interpreting a program model, and show it as diagrams, and that same program model is an abstract representation that can manipulated by a back-end component.

Figure 1.6 shows the main blocks that comprise a compiler. The composition of a compiler's *front-end* are roughly the same as in an interpreter. Differences start where a compiler connects to a middle layer through the generation of intermediate code.

Notice that the use of an intermediate language allows further analyses and code optimisations,

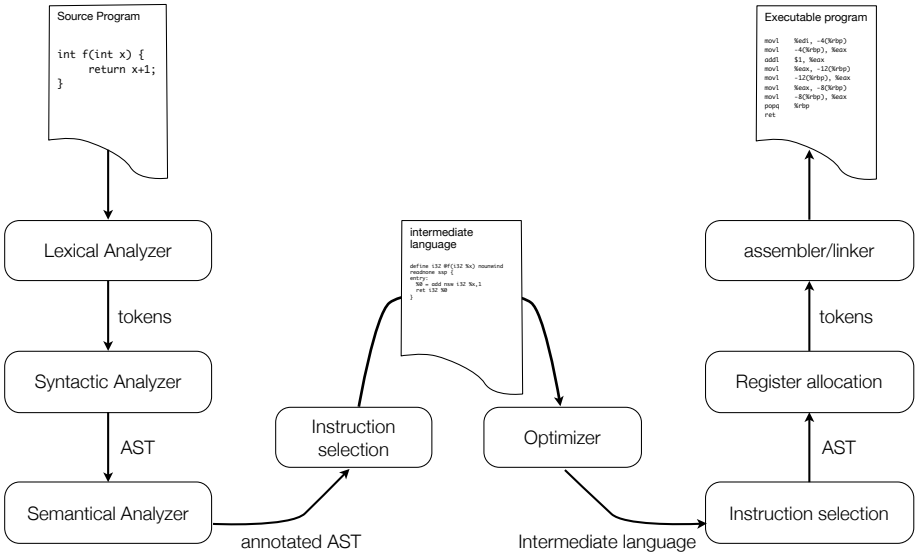


Figure 1.6: Compiler architecture (adapted from [Pfe])

and yet, is agnostic with relation to the target native architecture. See for example the program example in Figure 1.7, that gets compiled using clang (which uses the LLVM framework) using the command:

```
clang -c -S -emit-llvm celsius.c
```

to produce the file shown in Figure 1.8. This LLVM intermediate code can be optimized using a variety of techniques, using the optimiser command

```
clang -c -S -emit-llvm -O3 celsius.ll -o celsius.opt.ll
```

and produce the optimized code for function `main` depicted in Figure 1.9. This code can be directly interpreted using the LLVM virtual machine (`lli`), or it can be linked and translated to native machine code using the compiler's *back-end*. Whilst the *front-end* is (source) language specific, the *back-end* is specific to the target hardware architecture (or virtual machine). This layer can therefore be used to take advantage of the particular characteristics of each processor. This modular approach is also at the base of the so-called generic cross-compilers.

Notice that the function call from function `main` to function `toCelcius` in Figure 1.8 is replaced by the entire function body, we say that the function was inlined, resulting in a more efficient execution.

```
#include <stdio.h>

float toCelsius(int f) {
    return ((float)f - 32 )*5/9;
}

int main() {
    int f;
    scanf ( "%d" , &f );
    printf ( "%f␣oF␣\n" , toCelsius( f ) );
}
```

Figure 1.7: Example of C code

```
; Function Attrs: nounwind ssp uwtable
define float @toCelsius(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = sitofp i32 %3 to float
  %5 = fsub float %4, 3.200000e+01
  %6 = fmul float %5, 5.000000e+00
  %7 = fdiv float %6, 9.000000e+00
  ret float %7
}

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
  %3 = load i32, i32* %1, align 4
  %4 = call float @toCelsius(i32 %3)
  %5 = fpext float %4 to double
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
  ret i32 0
}
```

Figure 1.8: Example LLVM intermediate code

```
; Function Attrs: nounwind ssp uwtable
define i32 @main() local_unnamed_addr #1 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
  %3 = load i32, i32* %1, align 4
  %4 = sitofp i32 %3 to float
  %5 = fadd float %4, -3.200000e+01
  %6 = fmul float %5, 5.000000e+00
  %7 = fdiv float %6, 9.000000e+00
  %8 = fpext float %7 to double
  %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
  ret i32 0
}
```

Figure 1.9: Example of optimised LLVM intermediate code

1.3 Further reading

Many of the topics related with the classification of programming languages and language-based tools can be found in the introductory parts of many good textbooks like [Mit02]. Compiler construction books can also be used as a reference to a course similar to this, although with a different approach [AP02, ALSU06, FW08a].

2. Programs as data

In this lecture we discuss two fundamental aspects of programming language design: the syntax and the semantics of programming languages. We discuss how the syntax of a programming language can be represented, in an abstract and machine interpreted way, via an inductive data-type, hence defining programs as structured values. We then proceed by example, by defining the semantics of sample programmings languages, implemented by inductive algorithms. We introduce the first programming language in this course, comprising basic values and expressions, and use it as a running example in all other course lectures.

Syntax and semantics are two important aspects of the design process for a programming language. They define the set of valid programs, the programs that have meaning in some context. The set of syntactically valid programs is defined by combining a set of syntactic constructs, to which is then assigned a semantic meaning. The most important issue in programming languages, at the abstract level, consists in devising the set of constructs (the abstract syntax) whose semantics is better suited to solve the problem in hands. Nevertheless, when developing a concrete programming language, one has to carefully consider both the abstract and concrete syntaxes. Things like keyword choice, operator priority, indentation sensitivity, among others, must be considered.

The syntax of a language can be studied from two complementary perspectives. The **concrete** syntax of a programming language refers to the actual (formal and precise) representation of programs, as text files, box diagrams, flow charts, or some UI builder screens. The **abstract** syntax of a programming language defines the essence and structure of each language construct. This can immediately be mapped into data types that can be manipulated by programs (interpreters and compilers).

The syntax of a language is the first mechanism of establishing its expressiveness, by allowing or limiting the use of a certain constructs and their combinations. In this course we will primarily focus on text-based languages, at the concrete syntax level, but note that all concepts, techniques, and tools can be applied to other kind of languages.

```
S ::= ... | if E then S else S | if E then S | ...
```

Figure 2.1: Ambiguous grammar for conditional statements

```
1    if ( x > 0 )
2      if ( x < 10 )
3        return x;
4    else return 10;
```

Figure 2.2: Syntactic ambiguity in a C-like language.

2.1 Concrete Syntax

The concrete syntax of a programming language defines the set of artefacts accepted as syntactically correct programs of the language. The notion of a "legal" program of a given programming language is incrementally refined by other layers. Instances of such refinements can be well-typed programs, programs that satisfy a formal specification, programs that succeed on unit or integration tests, etc..

We now focus on having a precise definition of the concrete syntax for a programming language, which can be used to produce the first layer of a compiler or interpreter *front-end*. Issues like syntactic ambiguity are relevant at this stage. For instance, the code fragment in Figure 2.2 needs to be explicitly disambiguated. A grammar fragment for conditional statements, like the one in Figure 2.1, does not clearly identify which conditional statement contains the *else* branch in line 5. In languages like C or Java, the rule is to associate the `else` branch to the nearest `if` statement. If indentation is considered, in languages like Python or Haskell, the resulting derivation tree would be different. For instance, when using Python and its indentation based syntax, the program in Figure 2.3 is not ambiguous.

Consider another example in the Ruby language code fragment of Figure 2.4, where `name` can be used in distinct ways. An identifier with the form `@name` is syntactically different from an identifier with the form `name`. The former denotes an object instance variable, and the latter denotes a local variable, finally, the occurrence of `name` in a string literal, using a `#{...}` section, is in fact an occurrence of the object instance variable. Both situations regard syntactic issues of a particular language, which we will now start to deal with.

Consider our first programming language ($\mathscr{L}_0$), that allows basic operations on integer and boolean values. The concrete syntax of language $\mathscr{L}_0$ includes the terminal tokens that represent integer literals (*num*), and boolean literals (*true*). Recall the BNF specification language, and inspect the grammar given in Figure 2.5 for language $\mathscr{L}_0$.

Our first concern is to define the language concrete syntax and corresponding parser. We first need to define all the terminals (operators and keywords), and its non-terminals, which define all the allowed structures (expressions or programs). We also need to eliminate, by design, all the ambiguities of the grammar. In the language above, it is necessary to carefully define the priorities and associativity of all the operators and language constructs. Consider for instance, an expression like `1 == 2 + 3`, written without parenthesis. According to the grammar in Figure 2.5, it has two possible derivation trees, that correspond to either `(1 == 2) + 3`, or `1 == (2 + 3)`. The former is not the intuitive structure for the expression, the latter is clearly the intended expression. The same can be said about an expression like `1-2+3`, where the associativity of the operators is important. In this case, we consider that the correct derivation tree is given by `(1-2)+3`. We can express that kind of priority and associativity properties by rewriting the grammar as depicted in Figure 2.6. It accepts the same language, but ensures the absence of ambiguity. We use `E` for expressions, `D` for

```
1    if x > 0:
2      if x < 10:
3        return x
4    else:
5      return 10
```

Figure 2.3: Non ambiguous program in Python.

```
class Hello
  def init(name)
    @name = name
  end

  def hello
    puts "Hello #{@name}!"
  end
end
```

Figure 2.4: Syntactic detail of use of identifiers in Ruby.

disjunctions, `C` for conjunctions, `N` for negations, `R` for relations, `A` for addictive operations, `T` for terms (multiplicative), `F` for factors. Notice the use of a left recursive grammar in order to match the derivation of the grammar with the left associative operators of arithmetics and logic.

Notice that keywords and operators are used inside `'...'`, *num* denotes all integer literals, and non-terminals are given by capital letters. Terminals are accepted, and translated into a data type (token), by the lexical analyser (lexer). The language sentences, or programs, are accepted and translated into a data type by the syntactical analyser (parser). Lexing and parsing techniques are out of the scope of this course, and further reading are advised using for instance [AP02]. To bootstrap further work on lexing and parsing, refer to lab assignment 1.

2.2 Abstract syntax

The result of the syntactical analysis is, not only the acceptance or rejection of a program as part of a given language, but an abstract representation that can be manipulated and transformed. This representation is fairly independent from the actual form of the language constructs. It retains the essential characteristics of the language, by representing its structure and subcomponents of each language construct.

We call it the abstract syntax of the language, an abstract data type, or a data structure whose operations allow for the definition of algorithms over programs. We represent it as an inductively defined abstract data type, a recursive and compositional definition, that allows the definition of recursive and compositional algorithms over it. Interpreters, compilers, and program verification algorithms are instances of inductive (recursively defined) algorithms.

2.2.1 Inductive data types

An inductive data type definition, as a list of elements of type T, can be defined by a set of construction rules, like the following:

Definition 2.2.1 — List Data Type. The type "lists of elements of type T", is inductively defined by the cases:

1. `nil` is a list of elements of type T (the empty list)

```
E ::= num | E '+' E | E '-' E | E '*' E | E '/' E
    |  E '<=' E | E '<' E | E '>' E | E '>=' E | E '==' E
    | 'true | 'false' | E 'and' E | E 'or' E | 'not' E
    | 'if' E 'then' E 'else' E
    | '(' E ')'
```

Figure 2.5: $\mathscr{L}_0$ concrete syntax.

```
E ::= D | 'if' E 'then' E 'else' E
D ::= C | D 'or' C
C ::= N | C 'and' N
N ::= R | 'not' R
R ::= A | R '<=' A | R '<' A | R '>' A | R '>=' A | R '==' A
A ::= T | A '+' T | A '-' T
T ::= F | T '*' F | T '/' F
F ::= num | 'true | 'false' | '(' E ')'
```

Figure 2.6: Rewritten grammar for $\mathscr{L}_0$ language

2. if x is a value of type T and L is a list of elements of type T, then $\texttt{cons}(x,L)$ is a list of elements of type T.
3. there are no other values of type "lists of elements of type T", except the ones defined by rules 1 and 2.

This type definition has some components that deserve to be highlighted: a name for the type (*List*), and a set of value constructors (`nil` and `cons`). The values of a list of integers, *ListInt*, can be built using the following constructors (functions that produce values):

- `nil` : $() \rightarrow$ *ListInt*
- `cons` : *Integer* $\times$ *ListInt* $\rightarrow$ *ListInt*

That corresponds, for instance, to a `C` module with the interface described in Figure 2.7, to an OCaml module as in Figure 2.8, or to a Java interfaces as the one in Figure 2.9.

2.2.2 Inductive algorithms and properties

Inductive algorithms that process and analise a value can be defined over the inductively defined values, by analysing the construction cases in their types. We use the notion of mathematical induction when defining a function that computes the length of a list, like the one in Figure 2.10, or when proving a property about an operation like the ordered insertion of an element. Function `length`, the recursive call returns the length for the tail of the list, which is a part of the problem, a sub-problem, of the induction hypothesis, and the whole solution is then obtained from the partial solution and a rule for the general case (to add one to the length of the size of the tail).

Consider an arbitrary property about the results of function `orderedInsert` and expressed in the assertion

```
length (orderedInsert x l) = 1 + length l
```

To prove that such assertion holds one must consider the expansion of function `orderedInsert` which gives us two possible cases for variable `l`, either `Nil` or `Cons(y,ys)` for some value `y` and list `ys`. In the case where `l = Nil`, we have that `orderedInsert x Nil = Cons(x,Nil)`, hence, by inspection of function length, we have that `length (orderedInsert x Nil)` is the same as `length (Cons(x,Nil))` which is `1 + length Nil`. This proves our property in the case where `l = Nil`. In the other case for the input of function `orderedInsert`, you have `l = Cons(y,ys)`. By inspecting the function body, we identify two additional cases, depending on the condition `x <= y`.

```
/* ListInt.h */

typedef struct ListInt ListInt;

ListInt* nil();
ListInt* cons(int elem, ListInt *tail);
```

Figure 2.7: Interface of the module `ListInt` in C

```
module type List = Sig
  type intList
  val nil:intList
  val cons: int -> intList -> intList
end
```

Figure 2.8: Interface of the module `ListInt` in OCaml.

In the first case (where `x <= y`) we have that the answer is `Cons(x,Cons(y,ys))`, by inspection of function `length` we know that `length (Cons(x,Cons(y,ys))) = 1 + length (Cons(y,ys))` which matches our conclusion.

The property is proven valid in the last, inductive, case of function `orderedInsert`, by considering the induction hypothesis for `l = Cons(y,ys)` with `length (Cons(y,ys)) = 1 + length ys`

```
length(orderedInsert x ys) = 1 + length ys
```

In this case, the result of the insertion is `Cons(y,orderedInsert x ys)` and

```
length(Cons(y,orderedInsert x ys)) = 1 + length (orderedInsert x ys)
```

and

```
length(Cons(y,orderedInsert x ys)) = 2 + length ys
```

which is equivalent to `1 + length (Cons(y,ys))`, our conclusion.

This small exercise using mathematical illustrate the language-based inductive reasoning that is implemented by another class of tools, based on the axiomatic of programs introduces by Hoare in [Hoa69]. It also illustrates the way in which properties about programming languages are proven. One important property is the soundness of the execution rules of a language with relation to a typing discipline.

2.3 Abstract Syntax Trees

An abstract syntax tree is a value of an inductive data type, that represents programs in a given programming language. The derivation tree of a grammar defining a language, corresponds roughly to the structure of an abstract syntax tree.

The role of the syntactical analyser (parser) is to transform a sequence of tokens into a value of the data type that represents the programming language. A fragment of language $\mathscr{L}_0$ with expressions on integer values can be represented by the abstract data type *CALC*, and the following type constructors:

- `num` : $Integer \rightarrow CALC$
- `add` : $CALC \times CALC \rightarrow CALC$
- `sub` : $CALC \times CALC \rightarrow CALC$
- `mul` : $CALC \times CALC \rightarrow CALC$
- `div` : $CALC \times CALC \rightarrow CALC$

```
interface ListInt {...};

interface ListFactory {
  static ListInt nil();
  static ListInt cons(int elem, ListInt tail);
};
```

Figure 2.9: Interface of module `ListInt` in Java.

```
type list = Nil | Cons of int * list

let rec length l = match l with
    Nil -> 0
  | Cons(x,l) -> 1+(length l)

let orderedInsert x l = match l with
    Nil -> cons(x,Nil)
  | Cons(y,ys) -> if x <= y
                  then Cons(x,Cons(y,ys))
                  else Cons(y,orderedInsert x ys)
```

Figure 2.10: OCaml inductively defined data type and algorithms

This abstract data type definition corresponds to the `OCaml` sum type defined in Figure 2.11, and to the set of `Java` interface and classes depicted in Figure 2.12. The abstract representation of programs using these data types allows for the definition of inductive algorithms, that express the meaning of programs (interpreters), properties of programs (type systems), or code translations (compilers).

The construction of an AST usually follows the derivation tree of the language grammar. For instance, if we consider the derivation of the grammar in Figure 2.6 for expression `1+2*(4+5)`, we have the following expansions of non-terminal nodes:

```
E -> D
  -> C
  -> N
  -> A
  -> A '+' T
  -> T '+' T
  -> F '+' T
  -> 1 '+' T
  -> 1 '+' T '*' F
  -> 1 '+' F '*' F
  -> 1 '+' 2 '*' F
  -> 1 '+' 2 '*' '(' E ')'
  -> 1 '+' 2 '*' '(' D ')'
  -> 1 '+' 2 '*' '(' C ')'
  -> 1 '+' 2 '*' '(' N ')'
  -> 1 '+' 2 '*' '(' A ')'
  -> 1 '+' 2 '*' '(' A '+' T ')'
  -> 1 '+' 2 '*' '(' T '+' T ')'
  -> 1 '+' 2 '*' '(' F '+' T ')'
  -> 1 '+' 2 '*' '(' 4 '+' T ')'
  -> 1 '+' 2 '*' '(' 4 '+' F ')'
  -> 1 '+' 2 '*' '(' 4 '+' 5 ')'
```

```
type calc =
    Num of int
  | Add of calc * calc
  | Sub of calc * calc
  | Mul of calc * calc
  | Div of calc * calc
```

Figure 2.11: OCaml data-type, of programs of language *CALC*

```
interface ASTNode {...}

class ASTAdd {
  ASTNode left, right;
  ...
}

class ASTSub {
  ASTNode left, right;
  ...
}

class ASTMul {
  ASTNode left, right;
  ...
}

class ASTDiv {
  ASTNode left, right;
  ...
}
```

Figure 2.12: Java data-type `ASTNode`, of programs of language *CALC*

If we create a node for each case on the grammar rules we would have a tree like the one depicted in Figure 2.13. This is a complete derivation tree, depicting all the expansion of non-terminals of the grammar. If one takes the rule cases that actually consume tokens and ignore the remaining (redirection) nodes, then the derivation tree would look like the one in Figure 2.14. This derivation tree has the same structure as the corresponding object tree, obtained from the Java expression, using the classes in Figure 2.12:

```
new ASTAdd(new ASTNum(1),
           new ASTMul(new ASTNum(2),
                      new ASTAdd(new ASTNum(3), new ASTNum(4))))
```

or built using C-like constructors that return pointers to dynamic structures:

```
add(num(1),mul(num(2),add(num(3),num(4))))
```

or even the OCaml value of type `calc`:

```
Add(Num(1),Mul(Num(2),Add(Num(3),Num(4))))
```

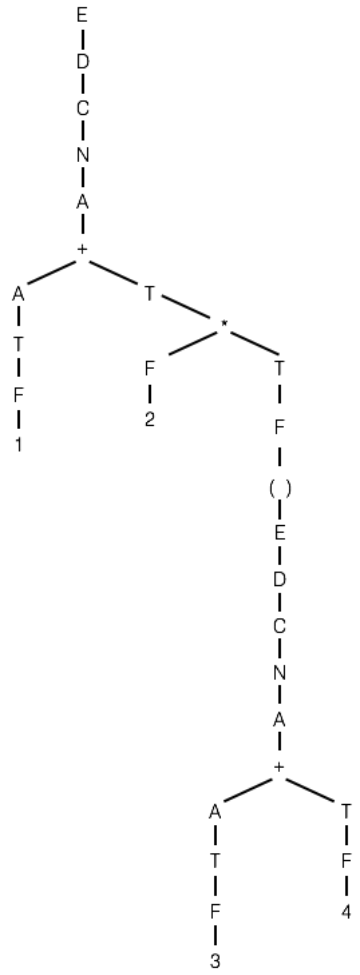


Figure 2.13: Complete grammar derivation tree

Exercise 2.1 Consider the concrete syntax of the language constructions presented next. Identify and present the necessary abstract syntax constructors and corresponding abstract syntax tree.

1. `x`
2. `0::[1;2;3]`
3. `f(0)`
4. `x = x + 1;`
5. `for(var i = 0; i < N; i++) { a += a * i; }`
6. `for(Integer i : sizes) { s += s + i; }`
7. `function (x) { return x*3; }`
8.
```
class A {
    int a;

    A() { a = 0 }
}
```

■

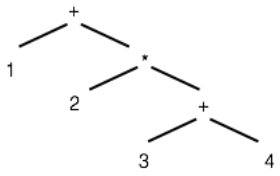


Figure 2.14: Abstract tree obtained from the complete grammar derivation

3. Semantics

The semantics of a programming language is the universal definition of the meaning of every valid program written in that language. There are several ways of defining such meanings, but here we adopt a more practical and direct kind of definitions. In the literature, the kind of approach explained in the chapter is called a big-step, structural operation semantics. Big-step because it directly yields as result of a function a value for a given program. Structural because it is defined inductively on the syntactic structure of each expression of the language.

3.1 Structural Operational Semantics

The definition of a structural operational semantics is a technique to give a precise meaning to programs of a programming language. It consists in a function, defined by case analysis, that associates a denotation to programs, based on the denotations of its subcomponents. One can define several different semantic functions for a given programming language, hence obtaining different kinds of denotations in different domains of analysis. Interpreters, compilers, and type systems are instances of semantic functions, of different natures, defined over the elements of a programming language (programs).

Given the representation of programs as values explored in the previous chapter, using a data type to define the set of all possible programs, we can now build and manipulate values of such data types. We will use functions on programs to compute their meaning (denotation). The semantics of a language can be characterised by a function whose domain is the set of programs (e.g. *PROG*), and target set that is the set of all possible denotations (e.g. the set of all integer values). Such a function assigns a (single) denotation to each (syntactically correct) program, or program fragment.

Different kinds of interpreting functions will necessarily have different target sets (denotation sets). For instance, a type system is an interpretation function that assigns a denotation of the set of possible value types to programs (expressions). A compiler is a function whose target set are programs in an intermediate/machine language. We call structural operational semantics to a syntax-directed structural definition of the semantic function that assigns value denotations to programs. The basic principle in this kind of definition, is that the semantic denotation of a construct is based on the semantic denotation of its components.

```
let rec eval e =
  match e with
    Number n -> n
  | Add(l,r) -> (eval l)+(eval r)
  | Sub(l,r) -> (eval l)-(eval r)
  | Mul(l,r) -> (eval l)*(eval r)
  | Div(l,r) -> (eval l)/(eval r)
```

Figure 3.1: Evaluation function for *CALC* in OCaml

```
interface ASTNode {
  int eval();
}

class ASTAdd implements ASTNode {
  ASTNode left, right;
  int eval() { return left.eval() + right.eval(); }
}
```

Figure 3.2: Evaluation function for *CALC* in Java.

The structural operational semantics of the programming language *CALC* is defined by a semantic function, called *eval*, that assigns an expression its value. The domain of such function is the set of arithmetic expressions *CALC* and its target set is the set of integer values:

$$eval : CALC \rightarrow Integer$$

The function `eval` is inductively defined in the cases of the data type *CALC*, as it is the case of the OCaml code in Figure 3.1. The OCaml type of function `eval` is `calc->int`. This definition is compositional, which means that we can extend the language with new language constructs, by defining the semantics of such constructs separately. The same interpretation function can be defined, in an object-oriented style, by implementing a method `eval` in all the classes implementing the `ASTNode` interface (Figure 2.12), as depicted in Figure 3.2. Check the starter code given to learn yet another strategy to implement the `eval` function using the Visitor pattern[1].

Notice that the function in Figure 3.1 yields a result for all syntactically correct expressions, except in the case of a division by zero. This is an unexpected error in our implementation, i.e. the interpreter does not detect the error and crashes. Also, the target set of the `eval` function is directly mapped to the `int` data type.

Consider language $\mathscr{L}_0$ (Figure 2.5) where the set of valid results is $Integer \cup \{\texttt{true}, \texttt{false}\}$. In this case the signature of method `eval` must be different to also consider boolean values as results. A simple way of implementing an interpreter is to return a sum type as it is defined in Java by interface `IValue` and classes `IntValue` and `BoolValue`, in Figure 3.3. Notice that the return type of method `eval` changed to `IValue`.

In this case, there is an infinite number of syntactically valid expressions for which the result is not defined. For example, the denotation expression `1+true` is not defined. Hence, this demands that a trapping mechanism is implemented as part of the semantic function to avoid a catastrophic stop of the runtime environment. Dynamically typed languages like Python, JavaScript, or Ruby constantly check for the appropriate types of the operands, on every operation of a program. Statically typed languages, usually compiled, can skip runtime type verifications, with obvious performance advantages.

[1] For more on design patterns see [GHJV95]

```
interface IValue {}

interface ASTNode { IValue eval() throws TypeMismatchException ; }

class IntValue implements IValue {
  int val;
  IntValue(int val) { this.val = val; }
  int getValue() { return val; }
}

class BoolValue implements IValue {
  boolean val;
  BoolValue(boolean val) { this.val = val; }
  boolean getValue() { return val; }
}

class ASTAdd implements ASTNode {
  ASTNode left, right;
  IValue eval() throws TypeMismatchException {
    IValue l = left.eval();
    IValue r = right.eval();
    if( l instanceOf IntValue && r instanceOf IntValue )
      return new IntValue(((IntValue) l).getValue()+((IntValue)r).getValue());
    else
      throw new TypeMismatchException();
  }
}
```

Figure 3.3: Evaluation function for $\mathscr{L}_0$ in Java with dynamic typing.

```
type result = IntValue of int | BoolValue of bool
```

Figure 3.4: Results of eval function using sum types.

This technique demands the constant boxing and unboxing of integer and boolean values. Languages that already implement sum types natively, like in OCaml, the return type of function `eval` would be given by a type such as the one declared in Figure 3.4. The use of tagged unions in the C programming language would also be an alternative, like in Figure 3.5.

Exercises

Exercise 3.1 — $\star$. Consider the language of regular expressions, whose concrete syntax contains the terminal symbols `*`, `+`, `?`, `|`, `(`, and `)`, and is defined by the following grammar:

```
typedef struct {
  int kind;
  union { bool b_value; int i_value; } value;
} result_t;
```

Figure 3.5: Results of eval function using union types.

$$
\begin{array}{rcl}
E & ::= & E\,\langle * \rangle \\
 & | & E\,\langle + \rangle \\
 & | & E\,\langle ? \rangle \\
 & | & E\,\langle | \rangle\,E \\
 & | & E\,E \\
 & | & \langle (\rangle\,E\,\langle) \rangle \\
 & | & \langle char \rangle
\end{array}
$$

Represent the abstract syntax of the language above using:

1. an hierarchy of Java classes.
2. an inductive data type in Ocaml

■

Exercise 3.2 — ⋆⋆⋆. Implement an inductive algorithm to translate a regular expression into an automaton.

Note: For the purpose of this exercise, an automaton can be characterized by a Java object or an OCaml function that recursively analyses a string given as input. Consider that each operation produces an automaton with a pair of starting and final states. An automaton for each non-terminal is obtained by combining the automatons generated from the sub-expressions (λ transitions are crucial to connect (sub)automatons). ■

3.2 Typing Semantics

The structural operational semantics for language $\mathscr{L}_0$ is a partial function that yields a result for some syntactically valid programs. By inspecting the implementation of function `eval` we see that there are situations where the meaning of a program is not defined. All language operators that destruct a value (inspect its actual meaning) have a clear specification for the operand's nature. So, the evaluation of ill-formed expressions like `1+true` or `1 && false` is interrupted by throwing a `TypeMismatchException`. If no traps were placed, the interpreter would try to convert the values to the expected type and would crash. If all traps of the kind illustrated in Figure 3.3 are in place, we have a clear identification of the programs for which the results are undefined. So, we may identify the set of possible runtime errors of a language by listing the possible (untrapped) errors of the underlying operators. Division by zero and type argument mismatch are amongst the most common runtime errors.

Languages that verify the operand's compatibility at runtime (prior to each operation execution) are classified as dynamically typed languages. An untyped language would be a language that would not yield an error on any type mismatch and would proceed in an erroneous path anyway. The real problem of dynamic type verification is that errors are detected only after a program starts being used. With high impact on the user experience.

A safer and more conservative approach is to statically type programs and ensure (prior to program execution or compilation) that a set of runtime errors, so-called type errors, do not occur at runtime.

3.2.1 Static typing

A general result of computation is that, in a Turing complete programming language, it is not possible to predict the outcome of a program without actually executing it, and the execution of a program may not terminate. It is impossible to predict if a program will terminate at all.

So, to predict if a program will terminate with an error, we approximate the result by defining a kind of semantic function using as target a more abstract set than the set of integer and boolean values.

We will use types as results, and each type represents the set of all possible values of that type. So, when we say that an expression is of type `int` (respectively `bool`), this means that its result will be for sure a member of the set of integer values (respectively the set of boolean values). We will call this function, `typecheck`.

A common result on programming languages is the soundness of the typing function with relation to the evaluation function. This result was first enunciated by Robin Milner in the sentence "Well-typed programs do not go wrong", where "wrong" is represented here by throwing a `TypeMismatchException` exception. This property can be stated as:

For all expressions `E`, if `typecheck(E) = T` and `eval(E) = V` then `V` is of type `T`.

This means that, for all programs that would end in error (the exception being a kind of result), the typing would be incorrect. However, type systems, as well as all other static analysis, are over approximations. This means that there are programs that will not end with an error and for which the typing function is not defined. We will get back to this idea after defining a type system for the language $\mathscr{L}_0$.

We define a function that maps expressions of language $\mathscr{L}_0$ to values in the set of types $Type \triangleq \{\texttt{int}, \texttt{bool}, \texttt{none}\}$. The signature of the function is:

$$typecheck : \mathscr{L}_0 \rightarrow Type$$

We now define the function inductively, and give the example for an `ASTAdd` expression in the Java code fragment in Figure 3.6. The base case for this induction is in the case for numbers and boolean values, in Figure 3.7. Notice that there are obvious optimizations that can be done, namely using a singleton pattern for the types that have only one possible value.

Given these two functions, it only makes sense to evaluate programs (function `eval`) for which there is a defined type, i.e. function `typecheck` yields a result different from `none`. Typing is also a pre-condition to compile a program. In the next chapter we present the functioning of a virtual machine (stack based) and compile language $\mathscr{L}_0$ to the bytecode.

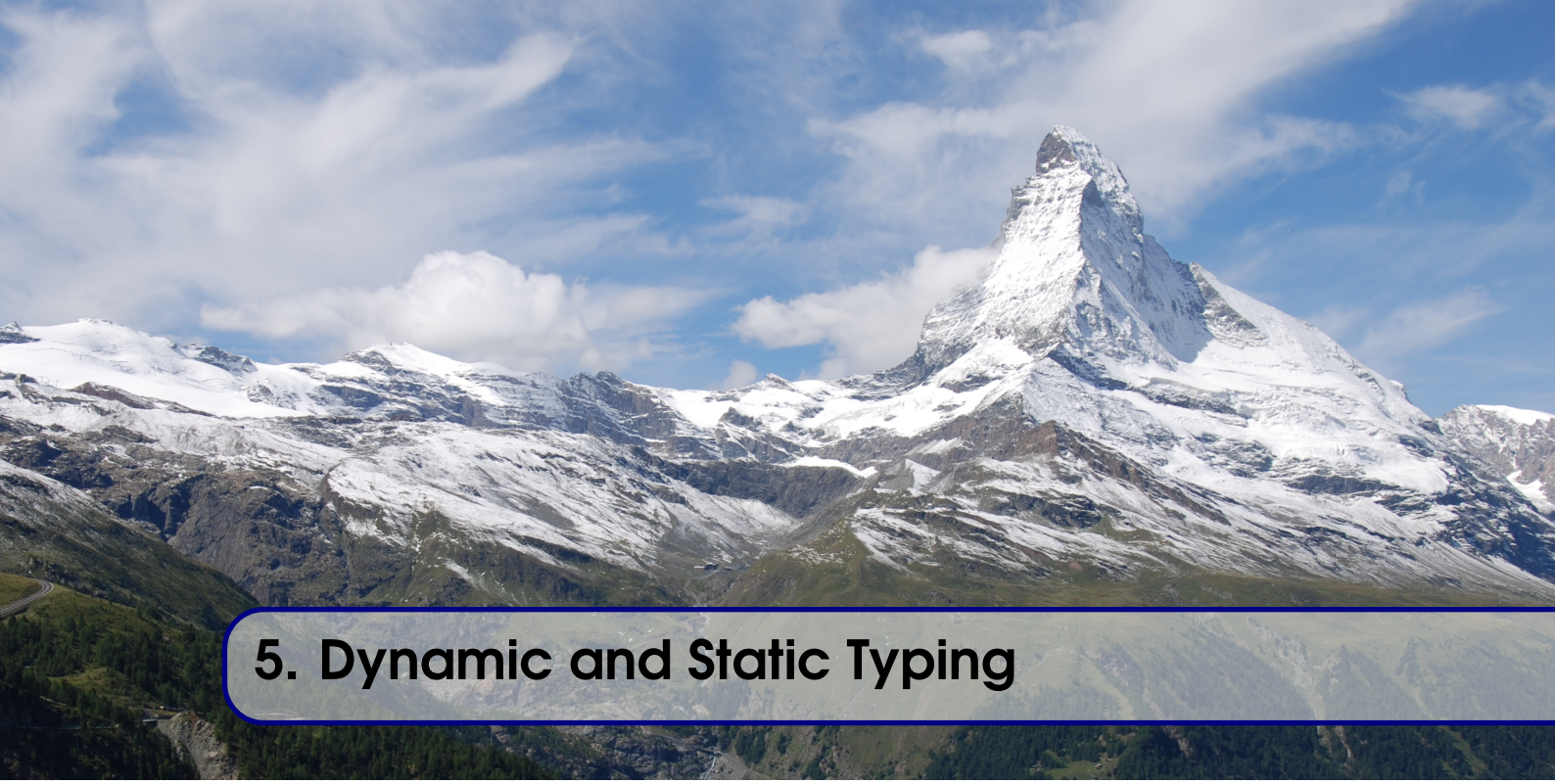
```
interface IType {}

interface ASTNode {
  ...
  IType typecheck();
}

class NoneType implements IType {}

class IntType implements IType {}

class BoolType implements IType {}

class ASTAdd implements ASTNode {
  ASTNode left, right;
  IValue eval() throws TypeMismatchException { ... }
  IType typecheck() {
    IType l = left.typecheck();
    IType r = right.typecheck();
    if( l instanceOf IntType && r instanceOf IntType )
      return new IntType();
    else
      return new NoneType();
  }
}
```

Figure 3.6: Static type checking function for language $\mathscr{L}_0$ in Java.

```
class ASTNum implements ASTNode {
  int val;
  IValue eval() throws TypeMismatchException { ... }
  IType typecheck() throws TypingException {
    return new IntType();
  }
}
class ASTBool implements ASTNode {
  boolean val;
  IValue eval() throws TypeMismatchException { ... }
  IType typecheck() throws TypingException {
    return new BoolType();
  }
}
```

Figure 3.7: Base cases for static type checking function.

4. The JVM stack machine



5. Dynamic and Static Typing

In this lecture we discuss the fundamental aspect of runtime errors and safety of programming languages as given by type verification mechanisms. We proceed by defining the semantics of a small untyped programming language, and check its runtime error situations. We then study dynamic typing mechanisms that allow runtime detection of erroneous situations, and static typing mechanisms that conservatively identify a larger set of runtime errors.

We define the static typing mechanism by means of a semantic function from programs to types, which guarantees that a set of runtime errors, so-called type errors, do not occur at runtime.

5.1 Error handling

Consider the abstract syntax for language *CALC* already presented in the previous lecture, here defined in Figure 5.1 using Haskell, and the corresponding semantic function, in Figure 5.2. The function `eval` yields a result for all syntactically correct expression, except in the case of a division by zero. This is an unexpected error in our implementation, i.e. the interpreter does not detect the error and crashes. Thus, the Haskell function `eval` defines a partial function from programs to integers, since it is not defined for all the elements of its domain (the set of programs or expressions), given by the data type `CALC`. Also, the target set of the eval function is directly mapped to the `Int` data type, which does not allow us to encode any form of error handling.

Undefined results correspond to what we commonly know as execution errors. A common approach to handling execution errors is to trap them, and treat them within the language domain, by dynamically analysing the operands of each language operation. For instance, in Java, an *array-out-of-bounds* error or a wrong cast operation is detected and handled using the exception mechanism of the language. A simpler approach, adopted by Javascript, is to extend the set of denotations, the target set of function `eval`, to include the special value `Undefined`, and define it by

$$\mathit{Integer} \cup \{\texttt{Undefined}\}.$$

Where the `Undefined` value is the denotation given to the programs for which no integer value can be computed due to an error.

```
data CALC =
    Num Int
  | Add CALC CALC
  | Sub CALC CALC
  | Mul CALC CALC
  | Div CALC CALC
  deriving Show
```

Figure 5.1: Haskell data type for *CALC* expressions.

```
eval :: CALC -> Int
eval (Num n) = n
eval (Add e e') = (eval e) + (eval e')
eval (Sub e e') = (eval e) - (eval e')
eval (Mul e e') = (eval e) * (eval e')
eval (Div e e') = div (eval e) (eval e')
```

Figure 5.2: Operational semantics for *CALC* expressions.

The extended result set can be defined as an abstract data type, and implemented by an `Haskell` inductive data type `Result`, like the one in Figure 5.3, or by a `Java` class hierarchy, like the one in Figure 5.4. To extend our interpreter to handle errors, one needs to anticipate all runtime errors and give the appropriate denotation. Check the `Haskell` code in Figure 5.5. Notice that there is an explicit check to ensure that both operands are indeed valid (`Value n`), otherwise the result is the special value `Undefined`. In the case of the division operation, the division by zero is detected and handled using the value `Undefined`. In the other cases of the language, an error occurring in one of the operand expressions is propagated to the result.

Exercise 5.1 Are there more special situations where this language semantics is not defined. Can you enumerate them? ■

5.2 Dynamic Typing

Runtime errors are usually due to a mismatch between given operands and the valid domain of the operations. For instance, the domain of division operation is the set of pairs that belong to $\mathbb{N} \times \mathbb{N}_{\setminus\{0\}}$ and the target set is $\mathbb{N}$.

When considering a language like $\mathscr{L}_0$, with the abstract syntax given in Figure 5.6, the set of valid results is

$$\mathit{Integer} \cup \{\texttt{true}, \texttt{false}\}.$$

In this language, there are many expressions for which there is not an integer or boolean denotation, which result from a mismatch in the operators domain and the nature of its operands. Adding a boolean value with an integer value makes no sense. These undefined results should be classified as execution errors, and can be trapped, as before, by analysing the operands of each language operation before the operation is erroneously applied. Given the approach above, We can extend the target set of function `eval` to

$$\mathit{Integer} \cup \{\texttt{true}, \texttt{false}\} \cup \{\texttt{Undefined}\}.$$

Hence, the result set can be defined by the `Haskell` data type in Figure 5.7 or by the corresponding `Java` class hierarchy in Figure 5.8.

```
data Result =
    Integer Int
  | Undefined
  deriving (Eq,Show)
```

Figure 5.3: Result set for language $\mathscr{L}_0$ in Haskell

```
interface Result {}

class IntegerValue { int value; }

class Undefined {}
```

Figure 5.4: Result set for language $\mathscr{L}_0$ in Java

We can now extend our interpreter definition to detect all possible execution errors and yield an `Undefined` result, as in the OCaml code fragment in Figure 5.10, or the Haskell code fragment in Figure 5.11 This strategy can be found in many interpreted languages, whose interpreters do not crash on invalid programs, but instead yield some distinguishable invalid result (e.g. Javascript). An alternative would be to generate some trapped exception that can be dealt by language mechanisms. This is a form of dynamic typing, where the interpreter code is able to detect the nature of the operand values, and act accordingly. Possible actions of the interpreter can be to convert some value from boolean to integer, or integer to a real number, or they can be to yield some undefined result.

Dynamic typing is a technique that demands a typed dynamic representation of values at runtime It allows for the design of more flexible languages, but it can be a factor of significant impact in the efficiency of the interpreter. An alternative to representing and checking type information at runtime, prior to each operation, is to use static typing.

5.3 Type Systems

Static typing works by analysing the source code, and conservatively consider all the possible runtime scenarios. Static typing consists in interpreting a program using representatives for the runtime values, i.e. types. Unlike computations over runtime values, computation over types is (on most type systems) decidable, and can be computed efficiently at development time.

The main goal of static typing is to avoid runtime errors, by rejecting, *à priori*, a program that does not conform to a type specification. Ill-typed programs may crash due to type errors at runtime. Not all execution errors can be easily predicted, and there is a wide range of type systems, to avoid an equally wide range of runtime errors.

The use of static typing also allows for a more efficient use of resources (memory and time), by totally (or partially) erasing the type information from the runtime representation of values. For instance, the runtime information, in languages like C, is totally erased from the runtime, while in Java, the produced bytecode is typed with basic and object types, but generic type variables are erased and replaced by the `Object` type. Also in Java, non-essencial information about identifiers used by the developer are simply erased from the compiled code. Static typing also allows for the generation of more efficient target code operations, by choosing in advance the operations suitable for a given value representation. Dynamic type information is usually used to choose, at runtime, the actual operations to apply, or which method to call (in OO languages), based on the actual nature of the operands (or arguments).

Type systems ensure basic properties like the declaration and initialisation of all used local variables' identifiers, or the implementation of all methods declared in some class interface. Type systems impose a programming discipline on the developer which is not enforced in the more

```
eval :: AST -> Result

eval (Num n) = Value n

eval (Add e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n+n')
              (_ , _) -> Undefined

eval (Sub e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n-n')
              (_ , _) -> Undefined

eval (Mul e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n*n')
              (_ , _) -> Undefined

eval (Div e e') =
         case (eval e, eval e') of
              (Value n, Value 0) -> Undefined
              (Value n, Value n') -> Value (div n n')
              (_ , _) -> Undefined
```

Figure 5.5: Evaluation function for language $\mathscr{L}_0$ in Haskell

flexible, and dynamically typed languages like Ruby, Python, or JavaScript.

A type system is a semantic function, and can be implemented by an interpreter working at a different abstraction level. We will use a technique similar to the definition of the operational semantics studied in the previous lecture. Denotations for programs, in this context, belong to the set of possible types of values. In the case of language $\mathscr{L}_0$ the set is:

$$\{Integer, Boolean, None\}$$

where type *Integer* is given to all programs always yielding integer values, *Boolean* is given to all programs always yielding boolean values, and *None* is given to all programs for which it is not possible to determine the nature of their result values.

The signature of the type checking semantic function on programs of language $\mathscr{L}_0$, called *typecheck*, is the following:

$$typecheck : VAL \to \{Integer, Boolean, None\}$$

Once again the target set can be represented by the abstract data type defined in Figure 5.12, and the abstraction function is given by the Haskell code in Figure 5.13.

Using the `Java` language, we can express the typecheck function in the way used to expressed the operational semantics. Types are expressed using a class hierarchy (see Figure 5.14) and the typing function is expressed as a new method of interface `ASTNode` and implemented in all classes of the AST (see Figure 5.15). Notice that, for the sake of simplicity, type classes are here represented as singleton classes, which makes comparisons be implemented by the comparator of base values.

```
data AST =
    Num Int
  | Add AST AST
  | Sub AST AST
  | Mul AST AST
  | Div AST AST
  | VTrue
  | VFalse
  | IF AST AST AST
  | And AST AST
  | Or AST AST
  | Geq AST AST
  | Gt AST AST
  | Leq AST AST
  | Lt AST AST
  | Eq AST AST
  deriving (Eq,Show)
```

Figure 5.6: Haskell data type for $\mathscr{L}_0$ expressions.

```
data Result =
    Integer Int
  | Boolean bool
  | Undefined
  deriving (Eq,Show)
```

Figure 5.7: Result set for language $\mathscr{L}_0$ in Haskell

We present in the lectures a discussion about the guaranties that are associated to well-typed programs, which will be appended to these notes afterwards.

5.4 Summary

In this lecture we've studied two different ways of trapping execution errors in programs. The first, dynamic type verification that avoids execution erros that cause the interpreter to crash, and also provides a structured way of dealing with errors (exception mechanisms, undefined values, etc).

The static type verification allows for a more conservative approach, by detecting execution errors without actually executing the code. One may inspect the definitions of Figures 5.11 and 5.13 and conclude that, all dynamic type verifications in the operational semantics function will always succeed.

5.5 Exercises

Exercise 5.2 Does the type system avoid all execution errors? Enumerate 5 examples of errors that are not usually trapped by a type system, and 5 execution errors that are. ■

Exercise 5.3 Write the soundness proof for language $\mathscr{L}_0$. ■

```
interface Result {}

class IntegerValue { int value; }

class BooleanValue { int value; }

class Undefined {}
```

Figure 5.8: Result set for language $\mathscr{L}_0$ in Java

```
public class ASTAdd implements ASTNode {

  ...

  public IValue eval() {
    IValue l = left.eval();
    IValue r = right.eval();

    if( l instanceof IntValue &&
        r instanceof IntValue ) {

      return new IntValue(
          ((IntValue) l).getValue() +
          ((IntValue)r).getValue());
    }
    else
      throw new TypeMismatchException();
  }
```

Figure 5.9: Dynamically typed interpreter for $\mathscr{L}_0$ (Java).

```
let rec eval e =
  match e with
    Number n -> Integer n
  | Add(l,r) ->
      (match eval l, eval r with
       | Integer n, Integer m -> Integer m+n
       | _, _ -> Undefined)
...
```

Figure 5.10: Dynamically typed interpreter for $\mathscr{L}_0$ (OCaml).

```
...

eval VTrue = Boolean True

eval VFalse = Boolean False

eval (IF e e' e'') =
         case eval e of
              Boolean b -> if b then eval e' else eval e''
              _ -> Undefined

eval (And e e') =
          case (eval e, eval e') of
              (Boolean b, Boolean b') -> Boolean (b && b')
              (_ , _) -> Undefined

...

eval (Lt e e') =
          case (eval e, eval e') of
              (Integer n, Integer m) -> Boolean (n < m)
              (_ , _) -> Undefined

eval (Eq e e') =
          case (eval e, eval e') of
              (Integer n, Integer m) -> Boolean (n == m)
              (Boolean b, Boolean b') -> Boolean (b == b')
              (_ , _) -> Undefined
```

Figure 5.11: Dynamically typed interpreter for $\mathscr{L}_0$ (Haskell).

```
data Type =
    IntType
  | BoolType
  | None
  deriving (Eq,Show)
```

Figure 5.12: Abstract data type for types in language $\mathscr{L}_0$.

```
typecheck :: AST -> Type

typecheck (Num n) = IntType

typecheck (Add e e') =
         case (typecheck e, typecheck e') of
              (IntType, IntType) -> IntType
              (_ , _) -> None
...
typecheck (Eq e e') =
          case (typecheck e, typecheck e') of
              (IntType, IntType) -> BoolType
              (BoolType, BoolType) -> BoolType
              (_ , _) -> None
```

Figure 5.13: Interpretation function for types in language $\mathscr{L}_0$.

```
interface Type {}

class IntType implements Type {
  static public final IntType value = new IntType();

  private IntType() {}
}

class BoolType implements Type {
  static public final BoolType value = new BoolType();

  private BoolType() {}
}
```

Figure 5.14: Types for language $\mathscr{L}_0$ in Java.

```
class ASTNum implements ASTNode {
  ...
  @Override
  public Type typecheck() throws TypeException {
    return IntType.value;
  }
}

class ASTAdd implements ASTNode {
  public final ASTNode left;
  public final ASTNode right;
  ...
  @Override
  public Type typecheck() throws TypeException {
    if( left.typecheck() == IntType.value &&
        right.typecheck() == IntType.value )
      return IntType.value;
    throw new TypeException();
  }
}
```

Figure 5.15: Typechecking fragment for $\mathscr{L}_0$ in Java.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[AP02] A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[FW08a] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

[FW08b] D.P. Friedman and M. Wand. *Essentials of programming languages*. Essentials of Programming Languages. MIT Press, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Har12] Robert Harper. *Practical Foundations for Programming Languages (second edition)*. Cambridge University Press, New York, NY, USA, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[Mit02] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002.

[Pfe] Frank Pfenning. Lecture notes: compiler construction, overview.

[Win06] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006.