Interpretation and Compilation of Programming Languages
Part 1 - Overview

João Costa Seco

March 5, 2014

This course is part of the curriculum of the third year of the "Mestrado Integrado em Engenharia Informática". It aims to introduce the fundamentals of design and implementation of programming languages. Many of the concepts presented in this course are essencial in a computational approach, as argued in this text about computational thinking [Win06], they appear in the context of programming languages in a pungent way. Concepts such as abstraction, parameterisation, specification, verification, state, are part of the computational framework one needs to resort in general.

This course follows, what may be called, a vertical structure. It requires 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 [FW08])

Programming languages are usually divided into classes like functional, imperative, object-oriented, logic, concurrent, etc., based on their main abstractions and mechanisms. The borders defined by this taxonomy are getting more and dimmer, 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).

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.

In this first lecture we describe the general architecture of interpreters and compilers, and its integration in a software system. Further reading is recommended using [AP02, ALSU06].

1 Compilers and interpreters

The core questions in this topic have the following form:

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

Each one of the tools we are referring to is built using a class of methods and techniques, and each has a specific role in either the development, or the runtime support of a software system.

Interpreters and compilers are essentially programs, whose input data, and sometimes their output data, are programs. In general, the actual behaviour 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.

Within the class of software tools where interpreters and compilers are in, there are other kinds, such as verification tools (type systems), code instrumentation tools, debuggers, integrated programming environments, database

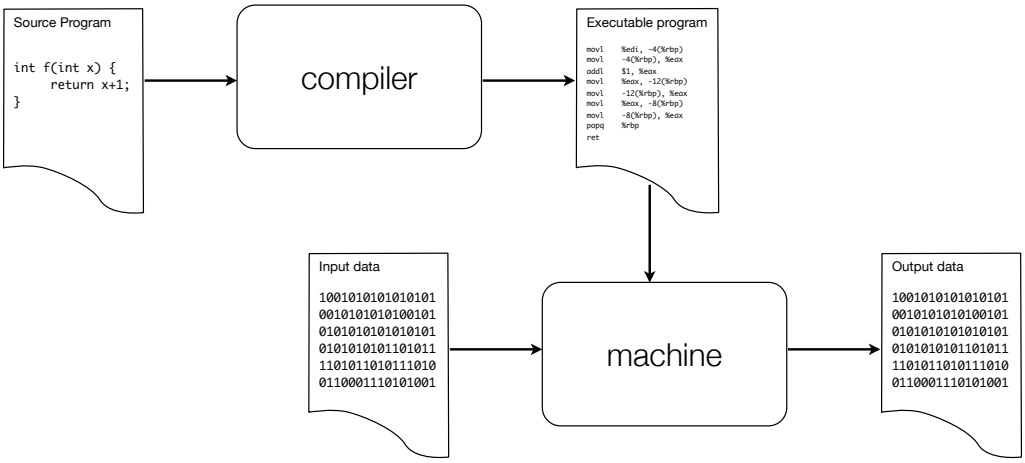

Figure 1: Compiler context diagram.

management systems (SQL interpreters). To distinguish them quickly, interpreters can be seen as virtual machines, whereas compilers are program translators.

The set of possible input values of this kind of tools, which are programs, is defined by a programming language, the actual representation of a program can be textual, or diagrammatic (visual). The behaviour of an interpreter or compiler is defined based on an abstract representation of a program. The base framework we are following starts with the definition of the data type that represents such abstract representation.

There are several ways of combining program transformation and verification tools. They vary basically in the kind of interaction and interfaces they present. A compiler, whose architecture is shown in Figure 1, transforms and optimizes a program written at a higher abstraction level (the source language) into a program written in a lower level of abstraction language (the target language). The target program is written in a way that it can be executed by a machine (virtual or real), and whose instructions are of a finer grain and adapted to its internal data structures.

Consider the simplified representations in Figures 1 and 2, 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..

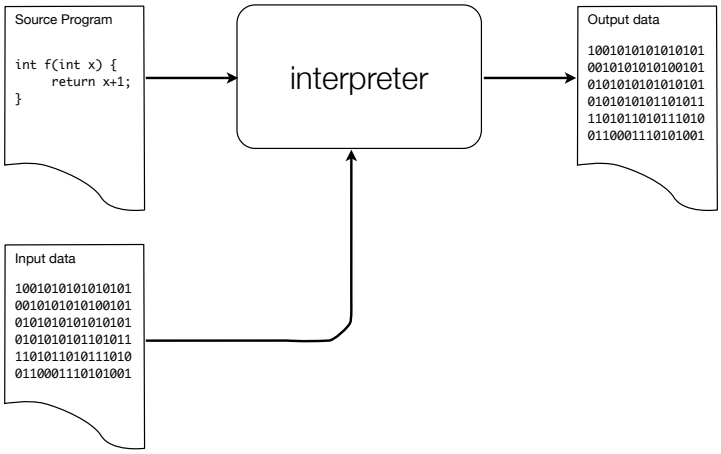

Figure 2: Interpreter context diagram.

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 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 .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 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 3. Another popular architecture is 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.

Efficiency By comparing the two approaches presented here, the usual balance is that the execution of compiler generated code runs faster and safer than the interpretation of the same code. Compilers process the code "offline", and can take advantage of optimisation and verification opportunities. Besides that, the fetch and execution cycle is hardware based, hence faster. Interpreted languages are, many times, dynamically typed, which means more time spent at runtime checking the true nature of operations' operands. Static typing offers an opportunity for faster execution because

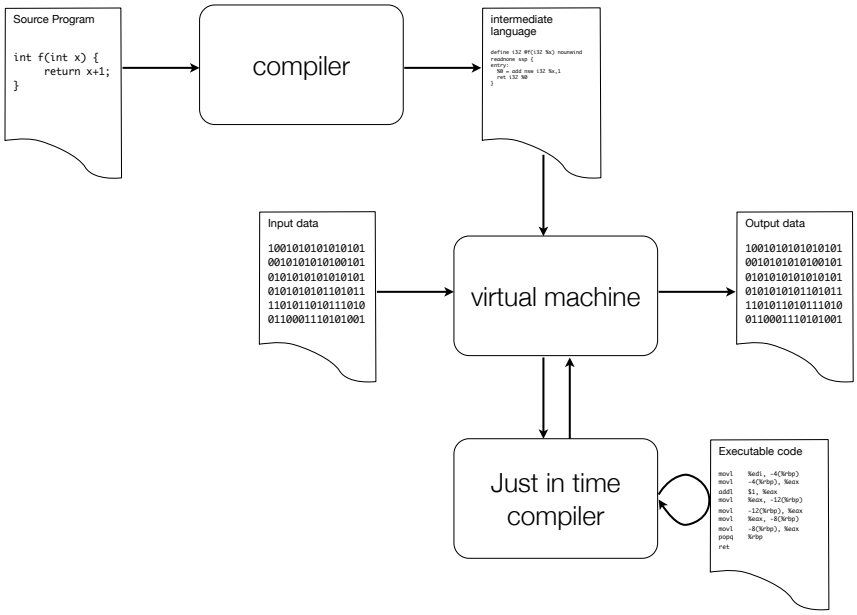

Figure 3: Interpreter with JIT compiler.

no runtime checks are needed.

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).

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

Figure 4: Sample intermediate language (JVM Bytecode)

1.1 Architecture

The internal structure of compilers and interpreters share some components and tasks, namely the treatment of input data, the *front-end*. Figure 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 recognised tokens, that is followed by a syntactical analysis, which 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.

Figure 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, and yet, is agnostic with relation to the target native

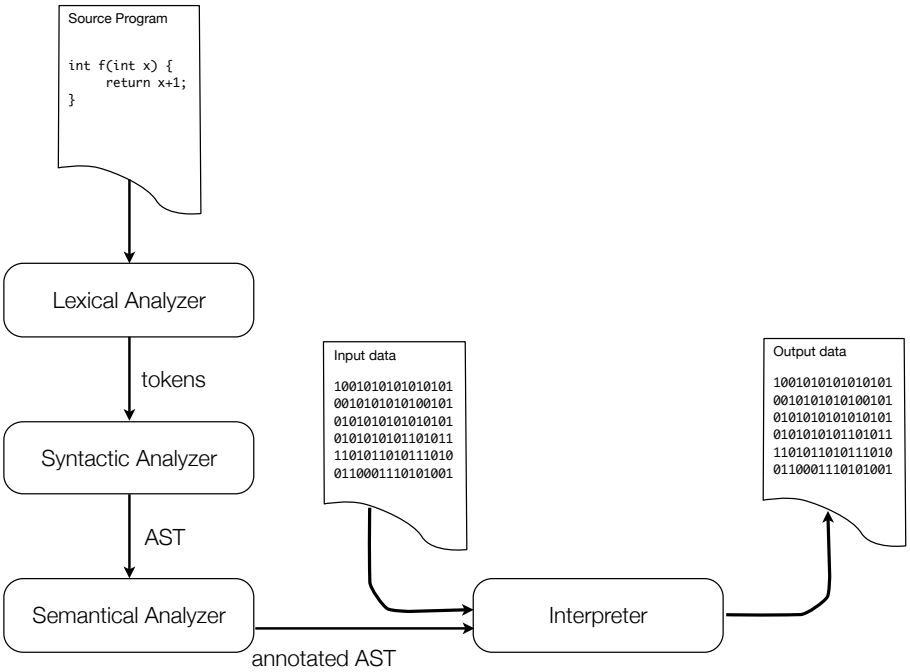

Figure 5: The architecture of an interpreter.

architecture. See for example the program example in Figure 7, that gets compiled using clang (from the LLVM framework) using the command:

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

to produce the file shown in Figure 8. This LLVM intermediate code can be optimized using a variety of techniques, using the optimiser command

```
opt -f -S -O3 toCelcius.s > toCelcius_opt.s
```

and produce the optimized code for function `main` depicted in Figure 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.

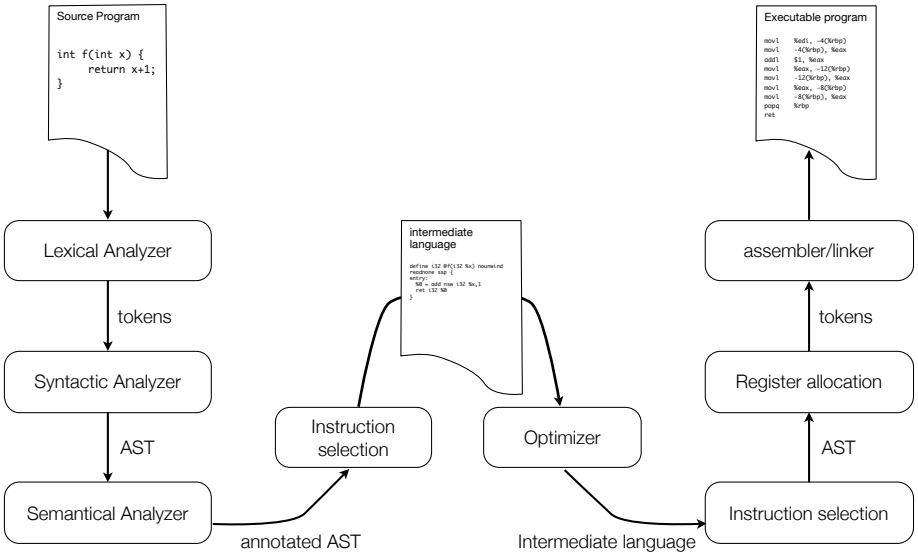

Figure 6: Compiler architecture (adapted from [Pfe])

References

[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.

[FW08] D.P. Friedman and M. Wand. *Essentials of programming languages*. Essentials of Programming Languages. MIT Press, 2008.

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

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

```c
#include <stdio.h>

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

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

Figure 7: Example of C code

```
; ModuleID = 'a.c'
target datalayout = "..."
target triple = "x86_64-apple-macosx10.9.0"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str1 = private unnamed_addr constant [7 x i8] c"%f\C2\BAF\0A\00", align 1

define float @toCelcius(i32 %f) nounwind ssp uwtable {
  %1 = alloca i32, align 4
  store i32 %f, i32* %1, align 4
  %2 = load i32* %1, align 4
  %3 = sitofp i32 %2 to float
  %4 = fsub float %3, 3.200000e+01
  %5 = fmul float %4, 5.000000e+00
  %6 = fdiv float %5, 9.000000e+00
  ret float %6
}

define i32 @main() nounwind ssp uwtable {
  %f = alloca i32, align 4
  %1 = call i32 (i8*, ...)* @scanf(i8* getelementptr inbounds ([3 x i8]* @.str,\
                                                                i32 0, i32 0), i32* %f)
  %2 = load i32* %f, align 4
  %3 = call float @toCelcius(i32 %2)
  %4 = fpext float %3 to double
  %5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([7 x i8]* @.str1,\
                                                                i32 0, i32 0), double %4)
  ret i32 0
}

declare i32 @scanf(i8*, ...)

declare i32 @printf(i8*, ...)
```

Figure 8: Example LLVM intermediate code

```
define i32 @main() nounwind uwtable ssp {
  %f = alloca i32, align 4
  %1 = call i32 (i8*, ...)* @scanf(i8* getelementptr inbounds ([3 x i8]* @.str,\
                                                  i64 0, i64 0), i32* %f) nounwind
  %2 = load i32* %f, align 4
  %3 = sitofp i32 %2 to float
  %4 = fadd float %3, -3.200000e+01
  %5 = fmul float %4, 5.000000e+00
  %6 = fdiv float %5, 9.000000e+00
  %7 = fpext float %6 to double
  %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([7 x i8]* @.str1,\
                                                  i64 0, i64 0), double %7) nounwind
  ret i32 0
}
```

Figure 9: Example of optimised LLVM intermediate code