Implementation of Programming Languages

An Overview

Gabriele Keller

March 2, 2005

1 Introduction

These notes give a brief overview of the implementation of programming languages. Since we only spend a week on this topic, which by itself provides enough material for a complete course, we will not go into any detail. The purpose of the first week is simply to give you a rough understanding of how compilers and interpreters work.

Programs written in high-level languages, or more precisely, any language apart from machine or assembly code, cannot be directly executed on a computer. There are two main ways to deal with this: either, by using a compiler to translate the source level program into an executable program in machine language; or, by using an interpreter which “understands” the source language and executes the commands and evaluates the expression of the program. However, some language implementors choose a middle way: a compiler translates the source program into an intermediate language, and then the resulting program is run by the interpreter. Let us have a look how interpreters and compilers work, and what the advantages and disadvantages are.

1.1 Compiler

Translating programs from a high-level language into executable code is a complex process, and is quite different from translating, for example assembly into machine code and vice versa. In the case of assembly code, each instruction directly corresponds to a machine code instruction. Therefore, an Assembler or Disassembler does not need to look at the whole program, it just translates instructions one after the other into the corresponding instruction of the target language. It is also straightforward to check if an instruction is legal, because there are only a fixed number of registers the instructions can operate on, and the type of an operand can be easily established.

The situation in high-level languages is quite different: let us take a simple expression like \( i + j \), as an example. Is this a legal expression? We can’t tell. In most languages, the expression is only accepted if \( i \) and \( j \) are both declared, and some might even require that both variables are also initialised. Also, it depends on the type of the variables: if, for example, one of them represents a string, the expression would be rejected in most languages. And finally, without knowing whether \( i \) and \( j \) are floating point or integer values, it is not even clear into which instruction it should be translated to. So, a compiler needs a whole lot of context information, and to get this information, it has to look at the whole program and understand it to a certain extend. Let us have a closer look at the different stages involved in this process. We use the simple C-program fragment listed in Figure 1 as running example.

1.1.1 Lexer

For the compiler, the source level program is just one big, unstructured string, and while humans are good at extracting meaning from such a representation, computers are not. When we read a
program, we automatically decompose the string into variable names, keywords, numerical constants and the like. In the same way, the first part of a compiler, called a **lexer**, extracts the meaningful components (**token**) of the program string and converts them into a more meaningful representation of an adequate data type. A program also contains a lot of information which is not important for the compilation process, but helpful for the programmer or anybody else who reads the program: for example additional spaces, tabs, comments. The lexer can safely ignore them.

For example, a lexer which processes the program displayed in Figure 1, it might produce the following sequence of tokens:

```
"int foo() \n int i;...
```

The exact internal representation of the tokens, of course, depends on the language the compiler is implemented in, and the way the programmer models the token data type in this language. A possible Haskell definition of the Token data type could look something like:\footnote{Type constructors are displayed in blue to distinguish them from types}

```haskell
data Token = Ident String | FloatConst Float | IntConst Int | Semicolon | LBrace | RBrace | LParent | RParent |
```

You can see that different constructors of the new type **Token** data type carry different information with them: it is, for example, not sufficient to record that a token is an identifier (**Ident**), but we also have to store the string that describes the name of the identifier, "int". Therefore, the constructor **Ident** has one argument of type **String**. Similarly, if the lexer finds an integer constant, the value this constant represents has to be stored as well.

The lexer is already able to detect a limited set of program errors: for example, if the source language does not permit anything but numbers to start why a digit, a program containing an identifier "3count" can already be rejected.

The lexer already deconstructs the string into meaningful components, and converts them into a more meaningful representation of an adequate data type. This process is called **tokenization**. A possible Haskell definition of the Token data type could look something like:

```haskell
data Token = Ident String | FloatConst Float | IntConst Int | Semicolon | LBrace | RBrace | LParent | RParent |
```

You can see that different constructors of the new type **Token** data type carry different information with them: it is, for example, not sufficient to record that a token is an identifier (**Ident**), but we also have to store the string that describes the name of the identifier, "int". Therefore, the constructor **Ident** has one argument of type **String**. Similarly, if the lexer finds an integer constant, the value this constant represents has to be stored as well.

The lexer is already able to detect a limited set of program errors: for example, if the source language does not permit anything but numbers to start why a digit, a program containing an identifier "3count" can already be rejected.
1.1.2 Syntactic Analysis

The lexer converts the source program into a list of tokens. The next component, parser, tries to identify structures, like variable declarations, loops, conditionals, in this sequence, and returns a so-called parse tree. To do this, the parser needs to know the grammar of the source language, which is usually specified in EBNF form (or extended Bacchus-Naur form). Consider the following fragment of a simplified C grammar:

\[
\begin{align*}
\text{funDef} &::= \text{Ident}_1 \text{Ident}_2 (\text{arguments}) \text{stmt} \\
\text{stmt} &::= \text{expr} ; | \text{if} \ \text{expr} \ \text{then} \ \text{stmt}_1 \ \text{else} \ \text{stmt}_2 ; | \text{return} \ \text{expr} ; |
\{ \text{locDec} \ \text{stmts} \} | \text{while} (\ \text{expr} \ ) \ \text{stmt} \\
\text{stmts} &::= \epsilon | \text{stmt} \ \text{stmts} \\
\text{expr} &::= \text{Num} | \text{Ident} | \text{expr}_1 + \text{expr}_2 | \text{expr}_1 - \text{expr}_2 | \\
&\quad \text{Ident} = \text{expr} | \text{Ident} (\text{exprs}) \\
\text{locDec} &::= \text{Ident}_1 \text{Ident}_2 ; \\
\text{arguments} &::= \epsilon | \ldots
\end{align*}
\]

It says that a function definition (funDef) consists of two identifiers, an opening parenthesis, arguments, closing parenthesis, followed by a statement stmt. In this grammar, funDef, arguments, and stmt are so-called non-terminals: place holders, whose exact form is specified in the grammar. Semicolon, parenthesis, and keywords are examples for terminals. Ident and Num are also a terminals — although they are place holders as well, their exact form is usually not defined in the grammar usually, but by additional lexical rules, which state what kind of symbols are allowed in an identifier, an integer constant, a floating point constant, and so on. In the grammar above, all terminals are displayed in blue, non-terminals in red, and symbols which belong the EBNF formalism in black.

From the second line of the grammar we learn that a statement can be an expression, followed by a semicolon. Separated by the bar symbol, which can be read as or, different alternatives are listed. The third line simply says that a list of statements stmts can be either empty (denoted by the symbol \(\epsilon\), or consist of a single statement followed by a (possibly empty) list of statements.

The parser checks if the source program adheres to the grammar of the language, and converts the flat sequence of tokens into a tree structure, the so-called parse tree. The tree has different kinds of nodes and leaves: generally speaking, the nodes in the tree correspond to the alternatives in the grammar, and the leaves to terminals, as you can see in our example tree in Figure 2. Note that some terminals in the source program are missing in the parse tree: separators like semicolon, parenthesis and brackets do not appear anymore. In the source program, they add structural information by grouping or separating expressions or statements. Once the parser gathers this information and encodes it in the tree, they are not necessary anymore for further compilation steps.

Again, there are different ways to model the parse tree. Let us have a look at the following (incomplete) Haskell definition:

\[
\begin{align*}
data \ \text{FunDef} &::= \text{FunDef} \ \text{Ident} \ \text{Ident} \ \text{Arguments} \ \text{Stmt} \\
data \ \text{Stmt} &::= \text{ExprStmt} \ \text{Expr} | \text{IfStmt} \ \text{Expr} \ \text{Stmt} \ \text{Stmt} \\
&\quad | \text{ReturnStmt} \ \text{Expr} | \text{Block} \ \text{LocDec} \ \text{Stmts} \\
&\quad | \text{WhileStmt} \ \text{Expr} \ \text{Stmt} \\
type \ \text{Stmts} &::= [\text{Stmts}] \\
data \ \text{Expr} &::= \ldots
\end{align*}
\]

According to the grammar, a function definition always consists of the same components: name of the result type, name of the function, arguments, and a statement, which is the function body. So, the data type FunDef has only one constructor, incidentally called FunDef as well. The constructor FunDef has exactly one argument for each of these components, with the corresponding type. A statement can have five different forms, and for each form we chose a different type constructor with an appropriate name (ExprStmt, IfStmt, ReturnStmt, Block, and WhileStmt). Since Stmts
Figure 2: Parse-tree of the example program

are a possibly list of zero or more Stmt, well, we model them as a list. How would you now define the data types Expr and locDec?

A parser can detect structural errors: for example, when the compiler is expecting to find a function definition, but the type of the return argument is missing, the parser will usually complain and trigger an appropriate error message.

1.1.3 Semantic Analysis

Lexer and parser only analyse the structure of a program, but do not check if the program is meaningful. Of course, a compiler cannot check if a program is correct in the sense that it produces the result or shows the behaviour expected by the programmer. It can, however, check some static properties, and make sure the program is consistent. For example, identifiers might have to be declared before use, and have to have the correct type. This check is done during the semantic analysis. It takes the parse tree as input, and attaches additional information to it: for example, for each identifier a pointer to its declaration, and the type of the variable. For this step, the compiler often computes a symbol table which provides a mapping between the symbols (function names, variable names, type names, method names) and their definition.

1.1.4 Optimisation and Code Generation

Ideally, a programmer writing in a high-level language should not have to know anything about the architecture the code will run on, and should not need to compromise the clarity of the code for the sake of efficiency. A perfect compiler should recognise these opportunities for optimisation. However, in reality, optimisations in compilers are a extremely hard problem.

Let us have a look at some simple optimisations to get an idea of the difficulties which arise. Procedure calls, for example, are fairly costly on some architectures. However, it is considered good style and improves the readability of a program to decompose a complex computation into a number of simple procedures. Therefore, the compiler will try and inline functions like these

```c
int bar (int x, int y) {
    return (x + y);
}
```

which means that a call to bar
\[ z = \text{bar}(4,z) \times 2; \]

is replaced by the body of the function definition, taking into account the values of the function parameter:

\[ z = (4 + z) \times 2; \]

This seems straight forward. Inlining, however, does not always make sense: what happens if the procedure is recursive, or there are many calls to a lengthy procedure? Even for such a seemingly simply optimisation, it is hard to specify hard and fast rules when they should be applied. Also, different optimisations might interact with each other and lead to unexpected effects, and might even lead to a slowdown of the resulting code.

### 1.2 Interpreter

Instead of writing a compiler to translate the source program into machine language, it is also possible to write an interpreter, which directly executes the program executing its instructions one by one. Like a compiler, an interpreter needs a lexer and syntactic and semantic analyser, which perform exactly the same function. However, an interpreter is not able to apply the same optimisations as a compiler, and does, obviously, not require a code generator.

An interpreter can be seen as abstract or virtual machine, a software layer on top of the concrete machine, which understands the commands of the high-level language and executes them. Although interpreted programs run slower than compiled programs, there are a number of advantages to interpreters as well: a interpreter starts executing the program directly, without having to completely analyse and translate it first. This is convenient while developing and debugging a program. Portability is also an issue: say, you have an interpreter for some language written in C. It means you can execute programs written in this language on any machine and operating system for which you have a C compiler. If you have a compiler for this language, you have to write a whole code generation phase, which is not a trivial task.

### 1.3 Hybrid Approach

Some languages are implemented using an approach between compiler and interpreter: they compile the source program into some intermediate code, which then is interpreted. One example for such a language is E-Lisp, a Lisp dialect used by GNU Emacs and XEmacs. ELisp is compiled into a bytecode, a fairly low-level intermediate language which is still completely architecture independent. Since bytecode is so close to machine code already, an alternative option to interpreting is Just in Time or Dynamic Compilation, where the bytecode is translated into machine code just before the execution of the program. The first major programming language for which JIT was implemented is SmallTalk. More recently, it has been taken up for languages such as Java, Python, C# (.Net bytecode).