Chapter 1 Programming Languages and their Processors

Programming languages are highly stylized, completely unambiguous artifacts that are in essence very different from the natural languages we use to communicate with each other even though they perhaps look very familiar. In fact, programming languages have more in common with the languages studied in formal language theory where sequences of symbols are investigated as a way to encode and represent computations. It turns out that most of today's programming languages are related to a class of formal languages called context-free languages. That programming languages are related to formal languages, *i.e.*, they can be viewed as unambiguous sequences of symbols representing computations, is important because that enables us to write programs that interpret them as well as translate them into other programming languages – the topic of this book.

There is an incredible variety of programming languages that is in use today. A quick survey of google.com groups comp.lang.* at the time of the writing of this book shows over 300 different programming languages actively being discussed and used. In order to make sense of this many languages we typically assign languages to different classes. Here we classify programming languages as high-level versus low-level languages. We define high-level programming languages as,

> *A high-level programming language is a language that supports data abstraction and structured programming.*

Data abstraction is the ability to encapsulate the implementation details of a data type and only expose an interface that makes the behavior of the data type visible. In object-oriented languages like C++ and Java this is accomplished by defining objects with an internal state that can be accessed via methods. In procedural languages like C or Pascal this is accomplished (albeit less elegantly) by creating data structures using the `struct` (C) or `record` (Pascal) commands and only allowing accessor functions to modify the data structures. Structured programming refers to programming language features that enforce a structured flow of control through a program (as opposed to unstructured programming using `goto` statements). Most high-level programming languages have structured programming constructs such as `while` loops and `if-then-else` statements.

On the other hand we define low-level languages as,

> *A low-level programming language is a language that does* **not** *support data abstraction and structured programming.*

Most assembly and bytecode languages are considered low-level languages. They typically do not possess any kind of data abstraction facility; byte or word level data is simply mapped onto physical memory which is then globally accessible. Flow of control is managed via conditional and unconditional jump instructions.

Two additional classifications have recently come into use. First we have the domain specific language defined as,

> *A domain specific language is a programming language dedicated to a particular problem domain, a particular problem representation technique, or a particular solution technique.*

Prime examples of domain specific languages are languages such as Html and PHP. Configuration managers such as the `make` facility under Linux/Unix is another example of a domain specific language. Languages in this class are geared towards solving one particular problem well. Html and PHP were designed to format and display web content. You would probably not use either language to tackle some sort of scientific computational problem. The same holds for `make`, it was specifically designed to manage the configuration of software systems, it would be difficult to use it for anything else since the language lacks any notion of arithmetic for example.

We also have the general purpose programming language that we define as,

> *A general purpose programming language is a language designed to be used for writing software in a wide variety of application domains.*

Most of our high-level programming languages such as Python, C, Pascal, Java, FORTRAN, *etc.* fall into this category. Another way of viewing this category is that a general purpose programming language is a language void of any domain specific features. For example, general purpose programming languages do not include features for page layout or software system configuration management (although they might possess libraries that accomplish that but these features are not built directly into the languages themselves).

Despite their differences and individual classifications all programming languages share two things:

1. A structural definition that specifies what programs in a particular programming language look like. By that we mean the shape of the sequences of symbols that make up valid programs.

2. A definition of the meaning or behavior of programs written in a particular language. For example, as programmers we are interested in the kinds of computations a particular program represents.

1.1 The Structure of Programming Languages

We already mentioned that programming languages are related to formal languages that consist of sequences of symbols encoding and representing computations. Being a bit more specific we can say the following,

> *A programming language is a formal system of symbols where the symbols are combined according to certain rules to make up larger structures of the language. This formal system of symbols is referred to as the syntax of the programming language.*

The syntax of a programming language is what we see in our editors when we write programs. You have probably experienced the strict nature of the formal system defining the syntax of a programming language when you forgot a semicolon at the end of a statement in Java for example. The Java compiler flagged an error in your program and probably suggested that it was expecting a semicolon in the program text. This means that your program violated the rules of the formal system defining the syntax of Java and therefore it was not considered to be a correct Java program.

The rules of the formal system that define the syntax of a programming language combine individual symbols to form larger and larger structures of the language. In most programming language we have a hierarchy of these structures,

Character – A single character.

Word – A word consists of one or more characters.

Phrase – A phrase consists of multiple words. A common phrase in many programming languages is an arithmetic expression such as `Var1 + Var2`.

Sentence – A sentence consists of one or more phrases and represents a program.

As outlined above we see that characters are combined to form words, words are combined to form phrases, and phrases are combined to form sentences. We say that a sentence is valid if the characters, words, and phrases are combined according to the rules defining the syntax of the programming language. In this view a program is simply a valid sentence.

One way to express the rules of the syntax of a programming language is to simply list all the possible legal combinations of characters, words, and phrases that make up valid sentences. Surprisingly we now have a precise definition of what we mean by a programming language,

A programming language is a collection of valid sentences.

In other words, a programming language is defined as the collection of all possible programs you can write in this language. For example, the Python programming language is defined as the collection of all possible Python programs. Although this is a precise and interesting definition of a programming language it is not very practical, since, as in the case of Python, the collection of valid sentences of most programming languages is infinite.

A more practical definition of a programming language is via rules that explicitly specify how to combine characters, words, and phrases into valid sentences. Programming languages are related to context-free formal languages and therefore we can use context-free grammars to succinctly state the rules that define the syntax of a programming language. Grammars have the property that every valid sentence of a language can be derived from the rules of the grammar and therefore the definition of a programming language via the collection of valid sentences and the definition of a programming language with a grammar are equivalent. However, the latter is much more practical because a grammar is a finite structure.

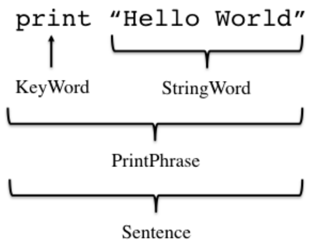


Figure 1.1: An example of the phrase structure of a programming language.

Figure 1.1 is an example of how characters, words, and phrases comprise programs in a programming language. The program shown in the figure is a sentence that consists of a phrase. The phrase in turn consists of the word `print` followed by the word `"Hello World"`. Given this hierarchical nature it is pretty straight forward to write down some rules that define this structure. Here is one way to do this,

$$Sentence \rightarrow PrintPhrase \tag{1.1}$$
$$PrintPhrase \rightarrow KeyWord\ StringWord \tag{1.2}$$
$$KeyWord \rightarrow \texttt{print} \tag{1.3}$$
$$StringWord \rightarrow \texttt{"Hello World"} \tag{1.4}$$

These rules state that a *Sentence* consists of a *PrintPhrase*, that a *PrintPhrase* consists of a *KeyWord* followed by a *StringWord*; that a *KeyWord* is the word `print`, and that a *StringWord* is the word `"Hello World"`. Notice that we can recover the original program from these rules by replacing the left side of a rule with the right side starting with *Sentence* as follows,

$Sentence$	[apply rule (1.1)]
$PrintPhrase$	[apply rule (1.2)]
$KeyWord\ StringWord$	[apply rule (1.3) to $KeyWord$]
`print` $StringWord$	[apply rule (1.4) to $StringWord$]
`print "Hello World"`	

A set of rules as shown in (1.1) through (1.4) that captures the structure of a programming language is called a grammar. Applying the rules to the symbol *Sentence* in such a way that we obtain the original program text is called a derivation. We will have much more to say about grammars and derivations when we look at actual programming language implementations.

We as software developers perceive the structure of programming languages in two dimensions. Consider this simple program that prints out a sequence of numbers,

```
i=0
while i < 10 do
    print i
    i=i+1
enddo
```

We perceive the structure of this program pretty much in the way it is shown here: The program consists of two statements - an assignment statement and a while loop. The body of the loop consists of two assignment statements. The indentation of the program text clearly indicates the hierarchical nature of the overall program. A language processor perceives the program quite differently in one dimension, namely as a stream of characters. To a language processor the above program would look something like this,

`i=0`**nl**`while`**sp**`i`**sp**`<`**sp**`10`**sp**`do`**nlspspsp**`print`**sp**`i`**nlspspsp**`i=i+1`**nl**`enddo`**nleof**

Here **nl**, **sp**, and **eof** are special characters denoting a new line, a space, and the end of file, respectively. The language processor will read this stream and use the grammar rules that define the structure of the language to group the characters into words, the words into phrases, and finally it will group the phrases into a sentence. If this grouping fails anywhere along those lines then the character stream does not represent a syntactically correct program and the language processor will flag a syntax error. This grouping of input characters into larger and larger language structures is called parsing and we will see later on that there are a number of algorithms that allow us to perform this grouping. Parsing a program file is also often referred to as syntax analysis.

1.2 The Behavior of Programming Languages

Of course it is not enough to know the syntax of a programming language, we also need to know the meaning of each syntactic structure of a programming language in order to effectively use a programming language. We usually associate meaning of a syntactic structure with the behavior of that structure when the program is executed. Consider the program in Figure 1.1, the meaning of this program is the behavior that the program will actually print `Hello World` to the screen. Now consider our program that prints out a sequence of integers,

```
i=0
while i < 10 do
    print i
    i=i+1
enddo
```

The meaning of this program is the behavior of printing,

```
0
1
2
3
4
5
6
7
8
9
```

to the screen. The overall behavior of our program is a composition of the behavior of the individual phrases or statements in the program. In this case the overall behavior is composed of the behavior of the assignment statement

```
i=0
```

which has the behavior of storing the value `0` in the variable `i` and the loop statement

```
while i < 10 do
    print i
    i=i+1
enddo
```

which has the behavior of executing its loop body as long as the loop expression `i < 10` remains true. The behavior of the loop body, in turn, is composed of the behavior of the print statement,

```
print i
```

The **While Loop**
Syntax:

$$WhileLoop \rightarrow \texttt{while}\ Expression\ \texttt{do}\ StatementList\ \texttt{enddo}$$

Semantics:
The while loop executes an *Expression* and a *StatementList* repeatedly until the value of the *Expression* is false. More specifically, a while loop is executed by first evaluating the *Expression*:

1. If the value is true, then the contained *StatementList* is executed. If execution of the *StatementList* completes normally, then the entire while loop is executed again, beginning by re-evaluating the *Expression*.

2. If the value is false, no further action is taken and the while loop terminates.

Figure 1.2: The specification of a while loop as a programming language feature.

and the assignment statement,

```
i=i+1
```

Therefore, in order to understand the overall behavior of a program all we need to do is look at the behavior of the individual parts or phrases of the program. This allows us to state that,

> *The meaning of a program is its behavior which is a composition of the behaviors of all the subparts of a program.*

We often refer to the meaning of a program as the semantics of a program.

Now, if we are interested in implementing a programming language then we need to somehow define the semantics of the individual syntactic structures of a programming language. Unfortunately, there exists no such nice mechanism as a set of grammar rules for specifying the semantics of syntactic structures. In most cases the semantics of a programming language is described in English prose. Consider the specification in Figure 1.2 that defines a while loop. This is a very typical specification of a programming language feature. It starts out by describing the syntax of the feature. In this case it provides a grammar rule for a while loop phrase that states that a while loop starts with the word `while` followed by an *Expression* phrase, followed by the word `do`, followed by a *StatementList* phrase, and finally followed by the word `enddo`. After the syntax definition we have the semantic definition. Here we define the behavior of the while loop with a description in plain English. This description gives a precise account of the expected behavior of the while loop in terms of evaluating the loop expression and executing the loop body.

A programming language implementer will use these kinds of descriptions to implement the semantics of a language. The part of a language processor that deals with the behavioral aspects of a programming language is called the semantic analysis and this is performed after the syntactic analysis of a program was successful. Not only does the semantic analysis phase of a language processor implement the behavioral aspects of a language but it also tries to verify that the behavior is correct as much as possible. Clearly, the semantic analysis of a language processor will not be able to tell you whether the algorithm that you wrote in that particular language is correct, but it will be able to tell you whether the program you wrote conforms to the semantic rules of the language. For example, if the loop expression of a while loop always evaluates to false, something like this $2 < 1$, then the language processor can issue a warning stating that your loop body will never execute. It is highly unlikely that you wrote a loop that never executes and therefore this behavior is probably due to a programming error and the language processor can assist you in writing better programs by analyzing the behavior of your program. Furthermore, in programming languages like Java or C++ variables that are used in a program need to be explicitly declared. Consequently, the semantic analysis phase of a Java or C++ compiler checks whether your program conforms to this behavior; if it doesn't the compiler will issue an error.

For a complete language specification the language designer has to give a specification as in Figure 1.2 for each syntactic construct of the programming language. For complicated languages like Java this is a non-trivial task; the full language specification of Java is a 700 page book!

There are exceptions to the rule that the semantics of a programming language is only defined in terms of English prose. The designers of the programming language ML constructed a mathematical model of the behavior of ML using the lambda calculus. The behavior of each syntactic construct of ML is given by a mathematical expression in this calculus. This means, given a program written in ML we could compute the behavior of this program using these mathematical expressions without having to execute the program. The benefit

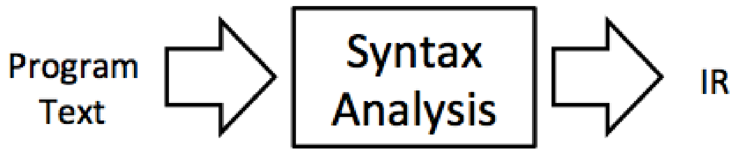


Figure 1.3: The syntax analysis module of a language processor.

of defining the behavior of a programming language mathematically is that it is an unambiguous specification and that we can easily check whether an implementation of the programming language is correct by comparing the behavior of the implementation with the computed behavior based on the semantic specification.

1.3 Language Processors

We are interested in implementing languages, that means we are interested in implementing language processors. The overall structure of language processors can be understood in terms of a few architectural modules or building blocks. As we will see, in many cases the architecture of language processors mirrors the structure of language specifications in that we have syntax and semantics analysis modules to deal with the syntactic and the behavioral aspects of programming language implementations.

1.3.1 Building Blocks

Syntax Analysis

Figure 1.3 shows a diagram of a syntax analysis module for language processors. This module uses the rules defining the syntax of a programming language to verify that the input program is syntactically correct. If it syntactically correct it will read the program and construct an intermediate representation (IR) of the program. The IR is an abstract representation of the program designed to facilitate follow-on processing of the program, such semantic analysis. The most common IR for programs is a tree based representation. A tree base representation is often referred to as an abstract syntax tree (AST).

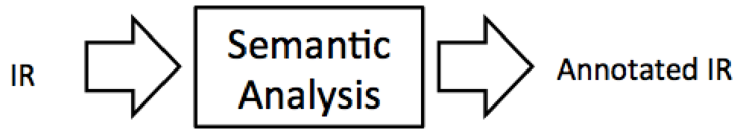


Figure 1.4: The semantic analysis module of a language processor.

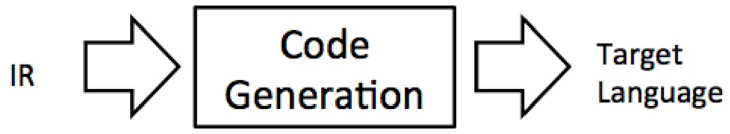


Figure 1.5: The semantic analysis module of a language processor.

It is interesting to note that the syntax module could be viewed as a stand-alone module as in the case of reading and processing configuration files as part of a larger program. Here, the syntax analysis module will simply read the configuration file and represent its contents in an abstract representation that is easily accessible for the rest of the program.

Semantic Analysis

The semantic analysis module shown in Figure 1.4 reads an IR of a program. If the IR of the program conforms to the semantic specification of the programming language then the module produces an annotated version of the IR. The annotation of the IR might include such things as the type and size of declared variables and wether or not a function call references a function defined in the current program file or if it is a call to an externally defined function. These and many other kinds of annotations are possible and are typically included in the annotated IR. The semantic module essentially attaches information to the validated IR of the program that will make later processing such as code generation easier.

Code Generation

The module shown in Figure 1.5 is the code generator module. It reads a (possibly annotated) IR of a program and converts this representation into code in some target language. The target language can be another language at the same level as the input language as in the case of pretty printers or it can be a lower level language compared to the language represented by the IR. The latter is the case in compilers where high-level languages are translated into low-level languages.

An interesting and perhaps non-intuitive application of the code generator module is report generation. Here a program assembles information and represents it as some sort of IR and calls upon the code generator module to generate human readable output.

1.3.2 Architectures

We obtain different kinds of language processors by snapping our building blocks together in different configurations.

The Reader

A reader consists of a single building block, namely the syntax analysis shown in Figure 1.3. The reader reads a file and produces an IR of that file. As mentioned above, one example of a reader is the configuration file reader. But there are other examples such as program analysis tools (*e.g.* word and line counters) as well as the Java class file loader. In the case of word and line counters the IR these programs produce is a single number, the number of words or lines, respectively.

The Generator

Similar to the reader, a generator only consists of a single building block: the code generator shown in Figure 1.5. A generator processes an IR and produces text. Above we mentioned report generation as an example of a generator. Other examples include object serializers and webpage generators.

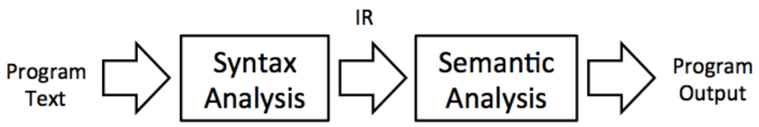


Figure 1.6: The architecture of an interpreter.

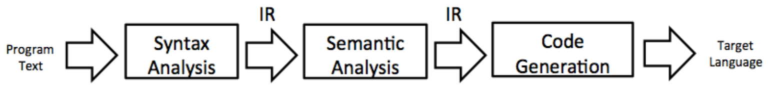


Figure 1.7: The architecture of a translator.

The Interpreter

The interpreter is a language processor that we would more typically associate with programming languages. It consists of a syntax analysis building block and a semantic analysis building block as shown in Figure 1.6. The syntax analysis reads the program text and produces an IR which the semantic analysis in turn will process and execute producing the output of the program. In the case of the interpreter we modify the semantic analysis slightly extending it from simply checking whether the input IR conforms to the desired behavior and generating an annotated IR to checking and then executing the IR. Executing the IR will produce the desired program output.

Examples of interpreted languages include simple programmable calculators as well as programming languages such as Lisp and Javascript.

The Translator

The translator uses all three of our building blocks as shown in Figure 1.7: syntax analysis, semantic analysis, and code generation. The translator reads a program text file, verifies the syntactical structure of that program, and constructs an IR of the source program. The IR is then processed by the semantic analysis insuring that the program conforms to the semantic rules of the language and then constructs the annotated IR. The annotated IR in turns is used by the code

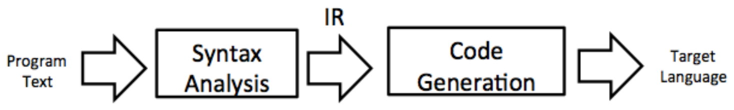


Figure 1.8: The architecture of a simple translator.

generator to emit code in the target language.

Examples of translators include assemblers, weblog analyzers, source-to-source translators (*e.g.* C++ to C translators), and of course compilers such as the Java compiler `javac`. A compiler is a translator that translates a high-level language into a low-level language.

The Simple Translator

The last architecture we consider is the simple translator. A simple translator consists of a syntax analysis building block and a code generating building block (Figure 1.8). The simple translator works similar to the translator but it does not perform any semantic checking. It is a purely syntactic translator. Examples include pretty printers and syntax highlighters.

1.3.3 An Example: Processing the Java Language

When processing a programming language, *i.e.* reading and executing a program, it is sometimes necessary to use more than one language processor. Consider the Java programming language. We can think of a Java processor in terms of a pipeline of multiple smaller processors. Figure 1.9 illustrates this pipeline. In order to run a program we first have to compile it into Java bytecode with the Java compiler and then interpret the generated bytecode in the Java virtual machine (JVM). This is shown in Figure 1.10. Here the Java compiler is a translator that reads a Java program and translates it into bytecode . The java compiler is translator as we defined earlier, Figure 1.11. The JVM is an interpreter that reads bytecode and executes it (Figure 1.12). In the Java language system bytecode is stored in class files.

If you have installed Java on your system so that it accessible from the shell (MacOS/Unix/Linux) or from the console (Windows) then you can follow the progress of a program through this pipeline. Assume that you have written a

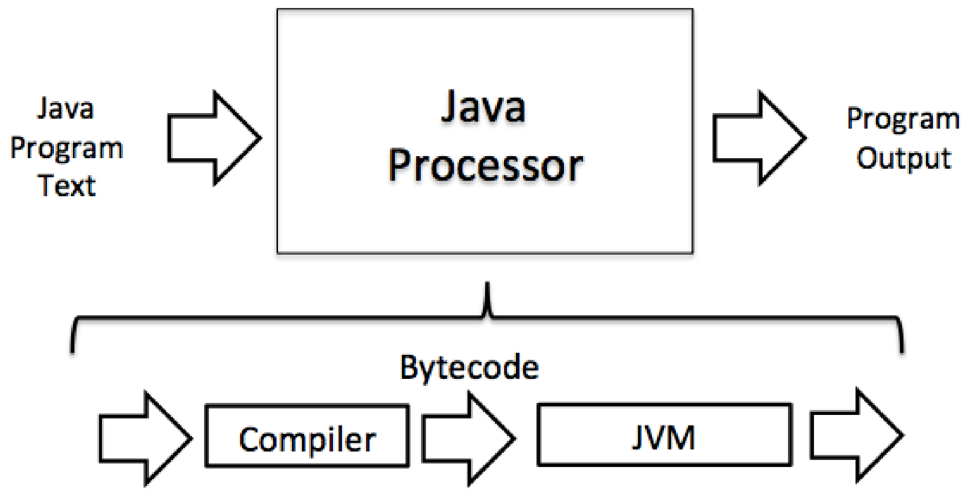


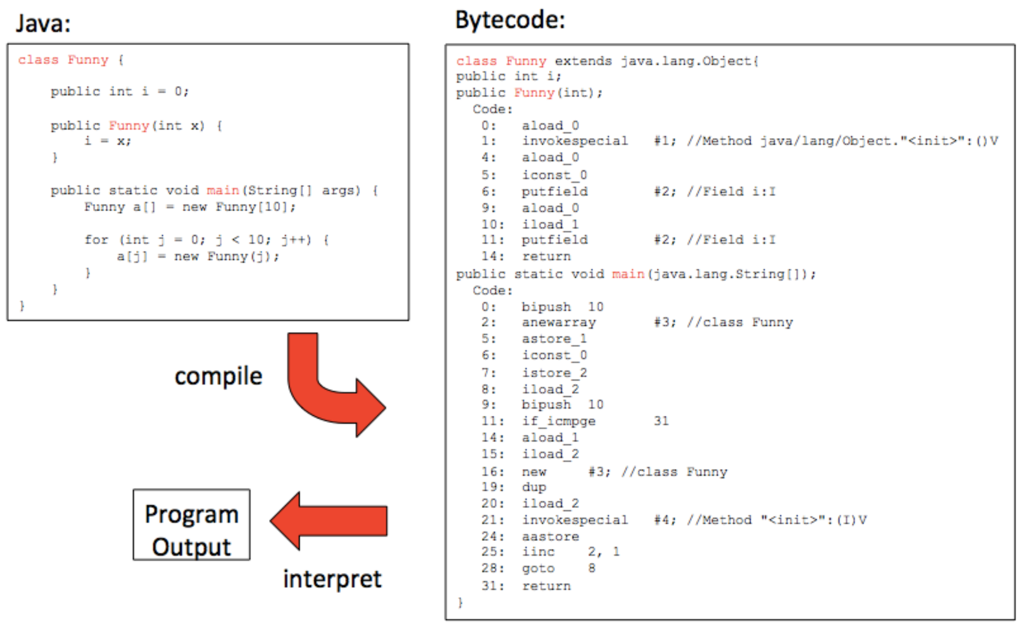
Figure 1.9: The Java processing pipeline.

simple Java program and that you stored it in the file `Funny.java`, then you can process your file as follows:

```
% javac Funny.java // compile Funny.java into Funny.class
% javap -c Funny.class // look at the bytecode  in the class file
% java Funny.class  // start the JVM and execute the bytecode
```

Chapter Summary

Programming languages are formal, unambiguous artifacts related to the structure studied in formal language theory. The are related to a class of formal languages called context-free languages and therefore can be specified using context-free grammars. Grammars are systems of rules that allow us to specify the syntactic structure of a programming language. However, in addition to the syntactic structure we also have a semantic aspect to programming languages. The semantics of a programming language is usually associated with the intended behavior of a program. Unfortunately, no formalisms such as grammar rules exist for the definition of the semantics of a programming language and



```
class Funny {

    public int i = 0;

    public Funny(int x) {
        i = x;
    }

    public static void main(String[] args) {
        Funny a[] = new Funny[10];

        for (int j = 0; j < 10; j++) {
            a[j] = new Funny(j);
        }
    }
}
```

```
class Funny extends java.lang.Object{
public int i;
public Funny(int);
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   iconst_0
   6:   putfield        #2; //Field i:I
   9:   aload_0
   10:  iload_1
   11:  putfield        #2; //Field i:I
   14:  return
public static void main(java.lang.String[]);
  Code:
   0:   bipush  10
   2:   anewarray       #3; //class Funny
   5:   astore_1
   6:   iconst_0
   7:   istore_2
   8:   iload_2
   9:   bipush  10
   11:  if_icmpge       31
   14:  aload_1
   15:  iload_2
   16:  new     #3; //class Funny
   19:  dup
   20:  iload_2
   21:  invokespecial   #4; //Method "<init>":(I)V
   24:  aastore
   25:  iinc    2, 1
   28:  goto    8
   31:  return
}
```

Figure 1.10: Following a source program through the Java processing pipeline.

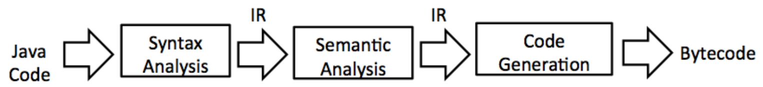


Figure 1.11: The Java compiler architecture.

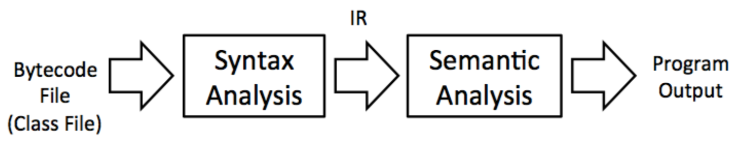


Figure 1.12: The Java virtual machine.

therefore the semantic aspect of a programming language is usually defined in English prose.

We concluded the chapter by looking at specific language processor architectures such as the interpreter and the compiler with the specific example of the Java language processor.

Bibliographic Notes

A nice introduction to formal language theory is Webber [17]. Our approach to language processor building blocks and architectures was heavily influenced by the work of Parr [11]. Exercise 7 is based on an example given in [18]. The mathematical definition of the ML programming language can be found in [10].

The following is a list of books and documents (in no particular order) that define and specify the various languages mentioned in the chapter: Java [5], C [7], C++ [16], Lisp [15], Pascal [6], HTML [13], PHP [9], FORTRAN [1], Make [4]. The Python Software Foundation keeps an up to date version of the Python language specification on their website: python.org. Oracle keeps an online version of the Java specifications: docs.oracle.com/javase/specs.

Exercises

1. Take your favorite programming language and try to classify it according to the language categories we established in the chapter. Justify your classification.

2. Find a programming language in each of the following categories:

   (a) A high-level language.

   (b) A low-level language.

   (c) A general purpose language.

   (d) A domain specific language.

   In each case justify your classification.

3. Write a very simple program in your favorite programming language and then try to define its syntactic structure with a set of rules similar to what we did in Section 1.1 with the rule set (1.1) through (1.4). Show that you can derive the original program from your rules.

4. Write down a set of rules similar to what we did in Section 1.1 with the rule set (1.1) through (1.4) that defines the syntactic structure of the following program,

   ```
   i=0
   while i < 10 do
       print i
       i=i+1
   enddo
   ```

   Show that you can derive the program above from your rules.

5. Write a simple program in your favorite programming language and then write a specification similar to the specification appearing in Figure 1.2 for each syntactic construct in the program.

6. Try to find a full specification in your library/on the web for

   (a) your favorite programming language (other than Java),

   (b) a programming language that you are interested in but not familiar with.

7. Rule sets as in (1.1) through (1.4) can also be used to define unambiguous snippets of natural language. Consider the following rule set,

$$\begin{array}{rcl}
Sentence & \rightarrow & NounPhrase\ Verb\ NounPhrase \\
NounPhrase & \rightarrow & Article\ Noun \\
Verb & \rightarrow & \texttt{loves} \\
Verb & \rightarrow & \texttt{chases} \\
Article & \rightarrow & \texttt{the} \\
Noun & \rightarrow & \texttt{dog} \\
Noun & \rightarrow & \texttt{cat}
\end{array}$$

   If we have multiple rules for the same structure we have a choice. Given these rules, do the following:

   (a) Starting with the symbol $Sentence$, derive the phrase '`the dog chases the cat`'. Show your derivation.

   (b) Is the phrase '`the cat chases the rat`' a sentence in the language defined by the rules above? Why? Why not?

   (c) Derive all possible sentences in the language defined by the rules above.

8. Try to determine the language processor architecture for

   (a) your favorite programming language (other than Java),
   (b) a programming language that you are interested in but not familiar with.

   Is it compiled or interpreted? Does the language processor consist of a pipeline of multiple language processors? If so, what are the architectures of the individual processors?

9. Find a programming language whose language processor consists of pipeline of multiple language processors. Identify the architectures of the language processors in the pipeline and follow a program through the pipeline.