Interpretation of Computer Programs

Expert programmers control the complexity of their designs with the same general techniques used by designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure. As programmers we use languages for constructing computational data objects and processes to model complex phenomena in the real world. However, as we confront increasingly complex problems, we will find that any fixed programming language is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.

Programming is endowed with a multitude of languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as procedure definition, that are appropriate to the larger-scale organization of systems.

Metalinguistic abstraction —establishing new languages— plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An *evaluator* (or *interpreter*) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

It is no exaggeration to regard this as the most fundamental idea in programming: *The evaluator, which determines the meaning of expressions in a programming language, is just another program*.

It sounds obvious, doesn't it? But the implications are profound. If you are a computational theorist, the interpreter idea recalls Gödel's discovery of the limitations of formal logical systems, Turing's concept of a universal computer, and von Neumann's basic notion of the stored-program machine. If you are a programmer, mastering the idea of an interpreter is a source of great power. It provokes a real shift in mindset, a basic change in the way you think about programming. 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.

There are three reasons why as a programmer you should learn about interpreters.

First, you will need at some point to implement interpreters, perhaps not interpreters for full-blown general-purpose languages, but interpreters just the same. Almost every complex computer system with which people interact in flexible ways —a computer drawing tool or an information-retrieval system, for example— includes some sort of interpreter that structures the interaction. These programs may include complex individual operations —shading a region on the display screen, or performing a database search— but the interpreter is the glue that lets you combine individual operations into useful patterns. Can you use the result of one operation as the input to another operation? Can you name a sequence of operations? Is the name local or global? Can you parameterize a sequence of operations, and give names to its inputs? And so on. No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system. It's easy to find examples of programs with good individual operations, but lousy glue.

Second, even programs that are not themselves interpreters have important interpreter-like pieces. Look inside a sophisticated computer-aided design system and you're likely to find a geometric recognition language, a graphics interpreter, a rule-based control interpreter, and an object-oriented language interpreter all working together. One of the most powerful ways to structure a complex program is as a collection of languages, each of which provides a different perspective, a different way of working with the program elements. Choosing the right kind of language for the right purpose, and understanding the implementation trade-offs involved: that's what the study of interpreters is about.

The third reason for learning about interpreters is that programming techniques that explicitly involve the structure of language are becoming increasingly important. Today's concern with designing and manipulating class hierarchies in object-oriented systems is only one example of this trend. Perhaps this is an inevitable consequence of the fact that our programs are becoming increasingly complex — thinking more explicitly about languages may be our best tool for dealing with this complexity. Consider again the basic idea: the interpreter itself is just a program. But that program is written in some language, whose interpreter is itself just a program written in some language whose interpreter is itself... Perhaps the whole distinction between program and programming language is a misleading idea, and future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application.

Mastery of interpreters does not come easily, and for good reason. The language designer is a further level removed from the end user than is the ordinary application programmer. In designing an application program, you think about the specific tasks to be performed, and consider what features to include. But in designing a language, you consider the various applications people might want to

implement, and the ways in which they might implement them. Should your language have static or dynamic scope, or a mixture? Should it have inheritance? Should it pass parameters by reference or by value? It all depends on how you expect your language to be used, which kinds of programs should be easy to write, and which you can afford to make more difficult.

Also, interpreters really *are* subtle programs. A simple change to a line of code in an interpreter can make an enormous difference in the behavior of the resulting language. Don't think that you can just skim these programs —very few people in the world can glance at a new interpreter and predict from that how it will behave even on relatively simple programs. So study these programs. Better yet, *run* them —this is working code. Try interpreting some simple expressions, then more complex ones. Add error messages. Modify the interpreters. Design your own variations. Try to really master these programs, not just get a vague feeling for how they work.

If you do this, you will change your view of your programming, and your view of yourself as a programmer. You'll come to see yourself as a designer of languages rather than only a user of languages, as a person who chooses the rules by which languages are put together, rather than only a follower of rules that other people have chosen.

In the last decade, information applications and services have entered the lives of people around the world in ways that hardly seemed possible in the '90s. They are powered by an ever-growing collection of languages and frameworks —all erected on an ever-expanding platform of interpreters.

Do you want to create Web pages? In 1995, that meant formatting static text and graphics, in effect, creating a program to be run by browsers executing only a single "print" statement. Today's dynamic Web pages make full use of scripting languages (another name for interpreted languages) like JavaScript. The browser programs can be complex, and including asynchronous calls to a Web server that is typically running a program in a completely different programming framework possibly with a host of services, each with its own individual language.

Or maybe you're programming a massive computing cluster to do searching on a global scale. If so, you might be writing your programs using the map-reduce paradigm of functional programming to relieve you of dealing explicitly with the details of how the individual processors are scheduled.

Or perhaps you're developing new algorithms for sensor networks, and exploring the use of lazy evaluation to better deal with parallelism and data aggregation. Or exploring transformation systems like XSLT for controlling Web pages. Or designing frameworks for remixing multimedia streams. Or...

So many new applications! So many new languages! So many new interpreters! As ever, novice programmers, even capable ones, can get along viewing each new framework individually, working within its fixed set of rules. But creating new frameworks requires skills of the master: understanding the principles that run across languages, appreciating which language features are best suited for which type of application, and knowing how to craft the interpreters that bring these languages to life.

1. THE PROGRAMMING LANGUAGE LIFE CYCLE

Every programming language has a life cycle, which has some similarities to the well-known software life cycle. The language is *designed* to meet some requirement. A formal or informal *specification* of the language is written in order to communicate the design to other people. The language is then implemented by means of language processors. Initially, a *prototype* implementation might be developed so that programmers can try out the language quickly. Later, high-quality (industrial-strength) *compilers* will be developed so that realistic application programming can be undertaken.

As the term suggests, the programming language life cycle is an iterative process. Language design is a highly creative and challenging endeavor, and no designer makes a perfect job at the first attempt. The experience of specifying or implementing a new language tends to expose irregularities in the design. Implementors and programmers might discover flaws in the specification, such as ambiguity, incompleteness, or inconsistency. They might also discover unpleasant features of the language itself, features that make the language unduly hard to implement efficiently, or unsatisfactory for programming.

In any case, the language might have to be redesigned, respecified, and reimplemented, perhaps several times. This is bound to be costly, i.e., time-consuming and expensive. It is necessary, therefore, to plan the life cycle in order to minimize costs.

Figure 1 illustrates a life cycle model that has much to recommend it. Design is immediately followed by specification. (This is needed to communicate the design to implementors and programmers.) Development of a prototype follows, and development of compilers follows that. Specification, prototyping, and compiler development are successively more costly, so it makes sense to order them in this way. The designer gets the fastest possible feedback, and costly compiler development is deferred until the language design has more or less stabilized.

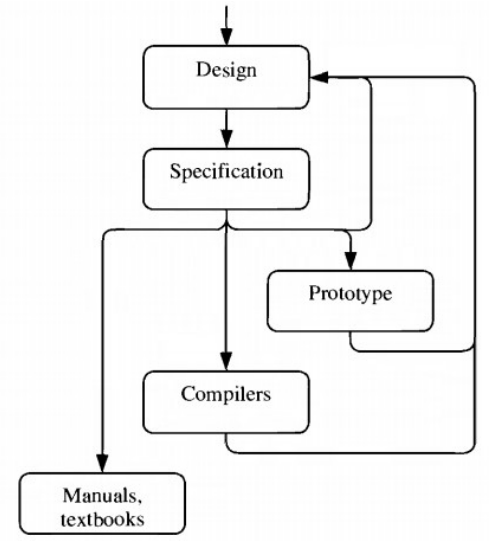


Figure 1. A programming language life cycle model

1.1. Design

The essence of programming language design is that the designer selects concepts and decides how to combine them. This selection is, of course, determined largely by the intended use of the language. A variety of concepts have found their way into programming languages: basic concepts such as values and types, storage, bindings, and abstraction; and more advanced concepts such as encapsulation, polymorphism, exceptions, and concurrency. A single language that supports all these concepts is likely to be very large and complex indeed (and its implementations will be large, complex, and costly). Therefore a judicious selection of concepts is necessary.

The designer should strive for simplicity and regularity. Simplicity implies that the language should support only the concepts essential to the applications for which the language is intended. Regularity implies that the language should combine these concepts in a systematic way, avoiding restrictions that might surprise programmers or make their task more difficult. Language irregularities also tend to make implementation more difficult.

A number of principles have been discovered that provide useful guidance to the designer:

- The *type completeness* principle suggests that no operation should be arbitrarily restricted in the types of its operands. For instance, operations like assignment and parameter passing should, ideally, be applicable to all types in the language.
- The *abstraction principle* suggests that, for each program phrase that specifies some kind of computation, there should be a way of abstracting that phrase and parameterizing it with respect to the entities on which it operates. For instance, it should be possible to abstract any expression to make a function, or (in an imperative language) to abstract any command to make a procedure.
- The *correspondence principle* suggests that, for each form of declaration, there should be a corresponding parameter mechanism. For instance, it should be possible to take a block with a constant definition and transform it into a procedure (or function) with a constant parameter.

These are principles, not dogma. Designers often have to make compromises (for example to avoid constructions that would be unduly difficult to implement). But at least the principles help the designer to make the hard design decisions rationally and fully conscious of their consequences.

The main purpose of this brief discussion has been to give an insight into why language design is so difficult.

1.2. Specification

Several groups of people have a direct interest in a programming language: the *designer* who invented the language in the first place; the *implementors*, whose task it is to write language processors; and the much larger community of ordinary *programmers*. All of these people must rely on a common understanding of the language, for which they must refer to an agreed *specification* of the language.

Several aspects of a programming language need to be specified:

- *Syntax* is concerned with the form of programs. A language's syntax defines what tokens (symbols) are used in programs, and how phrases are composed from tokens and subphrases. Examples of phrases are commands, expressions, declarations, and complete programs.
- *Contextual constraints* (sometimes called *static semantics*) are rules such as the following. *Scope rules* determine the scope of each declaration, and allow us to locate the declaration of each identifier. *Type rules* allow us to infer the type of each expression, and to ensure that each operation is supplied with operands of the correct types. Contextual constraints are so called because whether a phrase such as an expression is well-formed depends on its context.
- *Semantics* is concerned with the meanings of programs. There are various points of view on how we should specify semantics. From one point of view, we can take the meaning of a program to be a mathematical function, mapping the program's inputs to its outputs. (This is the basis of *denotational semantics*.) From another point of view, we can take the meaning of a program to be its behavior when it is run on a machine. (This is the basis of *operational semantics*.) Since we are studying language processors, i.e., systems that run programs or prepare them to be run, we shall prefer the operational point of view.

When a programming language is specified, there is a choice between formal and informal specification:

- An *informal specification* is one written in English or some other natural language. Such a specification can be readily understood by any user of the programming language, if it is well-written. Experience shows, however, that it is very hard to make an informal specification sufficiently precise for all the needs of implementors and programmers; misinterpretations are common. Even for the language designer, an informal specification is unsatisfactory because it can too easily be inconsistent or incomplete.
- A *formal specification* is one written in a precise notation. Such a specification is more likely to be unambiguous, consistent, and complete, and less likely to be misinterpreted. However, a formal specification will be intelligible only to people who understand the notation in which the specification is written.

In practice, most programming language specifications are hybrids. Nearly all language designers specify their syntax formally, using BNF, EBNF or syntax diagrams. These formalisms are widely understood and easy to use. Some older languages, such as Fortran and Cobol, did not have their syntax formalized, and it is noteworthy that their syntax is clumsy and irregular. Formal specification of syntax tends to encourage syntactic simplicity and regularity, as illustrated by Algol (the language for which BNF was invented) and its many successors. For example, the earlier versions of Fortran had several

different classes of expression, permissible in different contexts (assignment, array indexing, loop parameters); whereas Algol from the start had just one class of expression, permissible in all contexts.

Similarly, formal specification of semantics tends to encourage semantic simplicity and regularity. Unfortunately, few language designers yet attempt this. Semantic formalisms are much more difficult to master than BNF. Even then, writing a semantic specification of a real programming language (as opposed to a toy language) is a substantial task. Worst of all, the designer has to specify, not a stable well-understood language, but one that is gradually being designed and redesigned. Most semantic formalisms are ill-suited to meet the language designer's requirements, so it is not surprising that almost all designers content themselves with writing informal semantic specifications. Also contextual constraints are usually specified informally, because their formal specification is more difficult, and the available notations are not yet widely understood.

The advantages of formality and the disadvantages of informality should not be underestimated, however. Informal specifications have a strong tendency to be inconsistent or incomplete or both. Such specification errors lead to confusion when the language designer seeks feedback from colleagues, when the new language is implemented, and when programmers try to learn the new language. Of course, with sufficient investment of effort, most specification errors can be detected and corrected, but an informal specification will probably never be completely error-free. The same amount of effort could well produce a formal specification that is at least guaranteed to be precise.

The very act of writing a specification tends to focus the designer's mind on aspects of the design that are incomplete or inconsistent. Thus the specification exercise provides valuable and timely feedback to the designer. Once the design is completed, the specification (whether formal or informal) will be used to guide subsequent implementations of the new language.

1.3. Prototypes

A prototype is a cheap low-quality implementation of a new programming language. Development of a prototype helps to highlight any features of the language that are hard to implement. The prototype also gives programmers an early opportunity to try out the language. Thus the language designer gains further valuable feedback. Moreover, since a prototype can be developed relatively quickly, the feedback is timely enough to make a language revision feasible. A prototype might lack speed and good error reporting; but these qualities are deliberately sacrificed for the sake of rapid implementation.

For a suitable programming language, an *interpreter* might well be a useful prototype. An interpreter is very much easier and quicker to implement than a compiler for the same language. The drawback of an interpreter is that an interpreted program will run perhaps 100 times more slowly than an equivalent machine-code program. Programmers will quickly tire of this enormous inefficiency, once they pass the stage of trying out the language and start to use it to build real applications.

A more durable form of prototype is an interpretive compiler. This consists of a translator from the programming language to some suitable abstract machine code, together with an interpreter for the abstract machine. The interpreted object program will run 'only' about 10 times more slowly than a machine-code object program. Developing the compiler and interpreter together is still much less costly than developing a compiler that translates the programming language to real machine code. Indeed, a suitable abstract machine might be available 'off the shelf', saving the cost of writing the interpreter.

Another method of developing the prototype implementation is to implement a translator from the new language into an existing high-level language. Such a translation is usually straightforward (as long as the target language is chosen with care). Clearly the existing target language must already be supported by a suitable implementation. This was precisely the method chosen for the first implementation of C++, which used the `cfront` translator to convert the source program into C.

Development of the prototype must be guided by the language specification, whether the specification is formal or informal. The specification tells the implementor which programs are well-formed (i.e., conform to the language's syntax and contextual constraints) and what these programs should do when run.

1.4. Compilers

A prototype is not suitable for use over an extended period by a large number of programmers building real applications. When it has served its purpose of allowing programmers to try out the new language and provide feedback to the language designer, the prototype should be superseded by a higher-quality implementation. This is invariably a compiler — or, more likely, a family of compilers, generating object code for a number of target machines. Such a high-quality implementation is referred as an *industrial-strength* compiler.

The work that went into developing a prototype need not go to waste. If the prototype was an interpretive compiler, for example, we can bootstrap it to make a compiler that generates machine code.

Development of compilers must be guided by the language specification. A syntactic analyzer can be developed systematically from the source language's syntactic specification. A specification of the source language's scope rules and type rules should guide the development of a contextual analyzer. Finally, a specification of the source language's semantics should guide the development of a code specification, which should in turn be used to develop a code generator systematically.

In practice, contextual constraints and semantics are rarely specified formally. If we compare separately-developed compilers for the same language, we often find that they are consistent with respect to syntax, but inconsistent with respect to contextual constraints and semantics. This is no accident, because syntax is usually specified formally, and therefore precisely, and everything else informally, leading inevitably to misunderstanding.

2. LEVELS OF PROGRAMMING LANGUAGE

Low-level languages (*machine code*, *assembly*) are so called because they force algorithms to be expressed in terms of primitive instructions, of the kind that can be performed directly by electronic hardware. High-level languages (*C*, *Python*, *Java*, *Clojure*) are so called because they allow algorithms to be expressed in terms that are closer to the way in which we conceptualize these algorithms in our heads. The following are typical of concepts that are supported by high-level languages, but are supported only in a rudimentary form or not at all by low-level languages:

- *Expressions*: An expression is a rule for computing a value. The high-level language programmer can write expressions similar to ordinary mathematical notation, using operators such as '+', '-', '*', and '/'.
- *Data types*: Programs manipulate data of many types: primitive types such as truth values, characters, and integers, and composite types such as records and arrays. The high-level language programmer can explicitly define such types, and declare constants, variables, functions, and parameters of these types.
- *Control structures*: Control structures allow the high-level language programmer to program selective computation (e.g., by if- and case-commands) and iterative computation (e.g., by while- and for-commands).
- *Declarations*: Declarations allow the high-level language programmer to introduce identifiers to denote entities such as constant values, variables, procedures, functions, and types.
- *Abstraction*: An essential mental tool of the programmer is abstraction, or separation of concerns: separating the notion of *what* computation is to be performed from the details of *how* it is to be performed. The programmer can emphasize this separation by use of named procedures and functions. Moreover, these can be parameterized with respect to the entities on which they operate.
- *Encapsulation* (or *data abstraction*): Packages and classes allow the programmer to group together related declarations, and selectively to hide some of them. A particularly important usage of this concept is to group hidden variables together with operations on these variables, which is the essence of object-oriented programming.

2.1. High-Level Languages

Even in the earliest days of electronic computing in the 1940s it was clear that there was a need for software tools to support the programming process. Programming was done in *machine code*, it required considerable skill and was hard work, slow and error prone. Assembly languages were developed, relieving the programmer from having to deal with much of the low-level detail, but requiring an *assembler*, a piece of software to translate from assembly code to machine code. Giving symbolic names to instructions, values, storage locations, registers and so on allows the programmer to concentrate on

the coding of the algorithms rather than on the details of the binary representation required by the hardware and hence to become more productive. The *abstraction* provided by the assembly language allows the programmer to ignore the fine detail required to interact directly with the hardware.

The development of high-level languages gathered speed in the 1950s and beyond. In parallel there was a need for compilers and other tools for the implementation of these languages. The importance of formal language specifications was recognized and the correspondence between particular grammar types and straightforward implementation was understood. The extensive use of high-level languages prompted the rapid development of a wide range of new languages, some designed for particular application areas such as COBOL for business applications and FORTRAN for numerical computation. Others such as PL/I (then called NPL) tried to be very much more general-purpose. Large teams developed compilers for these languages in an environment where target machine architectures were changing fast too.

2.1.1. Advantages of High-Level Languages

The difficulties of programming in low-level languages are easy to see and the need for more user-friendly languages is obvious. A programming notation much closer to the problem specification is required. Higher level abstractions are needed so that the programmer can concentrate more on the problem rather than the details of the implementation of the solution.

High-level languages can offer such abstraction. They offer many potential advantages over low-level languages including:

- Problem solving is significantly faster. Moving from the problem specification to code is simpler using a high-level language. Debugging high-level language code is much easier. Some high-level languages are suited to rapid prototyping, making it particularly easy to try out new ideas and add debugging code.
- High-level language programs are generally easier to read, understand and hence maintain. Maintenance of code is now a huge industry where programmers are modifying code unlikely to have been written by themselves. High-level language programs can be made, at least to some extent, self-documenting, reducing the need for profuse comments and separate documentation. The reader of the code is not overwhelmed by the detail necessary in low-level language programs.
- High-level languages are easier to learn.
- High-level language programs can be structured more easily to reflect the structure of the original problem. Most current high-level languages support a wide range of program and data structuring features such as object orientation, support for asynchronous processes and parallelism.
- High-level languages can offer software portability. This demands some degree of language standardization. Most high-level languages are now fairly tightly defined so that, for example,

moving a Java program from one machine to another with different architectures and operating systems should be an easy task.

- Compile-time checking can remove many bugs at an early stage, before the program actually runs. Checking variable declarations, type checking, ensuring that variables are properly initialized, checking for compatibility in function arguments and so on are often supported by high-level languages. Furthermore, the compiler can insert runtime code such as array bound checking. The small additional runtime cost may be a small price to pay for early removal of errors.

2.1.2. Disadvantages of High-Level Languages

Despite these significant advantages, there may be circumstances where the use of a low-level language (typically an assembly language) may be more appropriate. We can identify possible advantages of the low-level language approach.

- The program may need to perform some low-level, hardware-specific operations which do not correspond to a high-level language feature. For example, the hardware may store device status information in a particular storage location—in most high-level languages there is no way to express direct machine addressing. There may be a need to perform low-level I/O, or make use of a specific machine instruction, again probably difficult to express in a high-level language.
- The use of low-level languages is often justified on the grounds of efficiency in terms of execution speed or runtime storage requirements. There are many programming applications where efficiency is a primary concern. These could be large-scale computations requiring days or weeks of processor time or even really short computations with severe real-time constraints. Efficiency is usually concerned with the minimization of computation time, but other constraints such as memory usage or power consumption could be more important. In the early development of language implementations, the issue of efficiency strongly influenced the design of compilers. The key disadvantage of high-level languages was seen as being one of poor efficiency. It was assumed that machine-generated code could never be as efficient as hand-written code. But as compiler technology steadily improved, as processors became faster and as their architectures became more suited to running compiler-generated code from high-level language programs, the efficiency argument became much less significant. Today, compiler-generated code for a wide range of programming languages and target machines is likely to be just as efficient, if not more so, than hand-written code. Does this imply that justifying the use of low-level languages on the grounds of producing efficient code is now wrong? The reality is that there may be some circumstances where coding in machine or assembly code, very carefully, by hand, will lead to better results. But, when developing software, a valuable rule to remember is that there is no need to optimize if the code is already fast enough.

3. A TAXONOMY OF PROGRAMMING LANGUAGE PROCESSORS

A *programming language processor* is any system that manipulates programs expressed in some particular programming language. With the help of language processors we can run programs, or prepare them to be run.

This definition of language processors is very general. It encompasses a variety of systems, including the following:

- *Editors*. An editor allows a program text to be entered, modified, and saved in a file. An ordinary text editor lets us edit any textual document (not necessarily a program text). A more sophisticated kind of editor is one tailored to edit programs expressed in a particular language.
- *Translators* and *compilers*. A translator translates a text from one language to another. In particular, a compiler translates a program from a high-level language to a low-level language, thus preparing it to be run on a machine. Prior to performing this translation, a compiler checks the program for syntactic and contextual errors.
- *Interpreters*. An interpreter takes a program expressed in a particular language, and runs it immediately. This mode of execution, omitting a compilation stage in favor of immediate response, is preferred in an interactive environment. Command languages and database query languages are usually interpreted.

In practice, we use all the above kinds of language processor in program development. In a conventional programming system, these language processors are usually separate tools; this is the 'software tools' philosophy. However, most systems now offer integrated language processors, in which editing, compilation, and interpretation are just options within a single system. The following examples contrast these two approaches.

Example 1. Language processors as software tools

The 'software tools' philosophy is well exemplified by the UNIX operating system. Indeed, this philosophy was fundamental to the system's design.

Consider a UNIX user developing a chess-playing application in Java, using the Java Development Kit (JDK). The user invokes an editor, such as the screen editor vi, to enter and store the program text in a file named (say) `Chess.java`:

```
vi Chess.Java
```

Then the user invokes the Java compiler, `javac`:

```
javac Chess.Java
```

This translates the stored program into object code, which it stores in a file named `Chess.class`. The user can now test the object-code program by running it using the interpreter, `java`:

```
java Chess
```

If the program fails to compile, or misbehaves when run, the user reinvokes the editor to modify the program; then reinvokes the compiler; and so on. Thus program development is an edit-compile-run cycle.

There is no direct communication between these language processors. If the program fails to compile, the compiler will generate one or more error reports, each indicating the position of the error. The user must note these error reports, and on reinvoking the editor must find the errors and correct them. This is very inconvenient, especially in the early stages of program development when errors might be numerous.

The essence of the 'software tools' philosophy is to provide a small number of common and simple tools, which can be used in various combinations to perform a large variety of tasks. Thus only a single editor need be provided, one that can be used to edit programs in a variety of languages, and indeed other textual documents too. What we have described is the 'software tools' philosophy in its purest form. In practice, the philosophy is compromised in order to make program development easier. The editor might have a facility that allows the user to compile the program (or indeed issue any system command) without leaving the editor. Some compilers go further: if the program fails to compile, the editor is automatically reinvoked and positioned at the first error.

These are *ad hoc* solutions. A fresh approach seems preferable: a fully integrated language processor, designed specifically to support the edit-compile-run cycle.

Example 2. Integrated language processor

Apache NetBeans is a fully integrated language processor for Java, consisting of an editor, a compiler, and other facilities. The user issues commands to open, edit, compile, and run the program. These commands may be selected from pull-down menus, or from the keyboard.

The editor is tailored to Java. It assists with the program layout using indentation, and it distinguishes between Java keywords, literals and comments using color. The editor is also fully integrated with the visual interface construction facilities of Apache NetBeans.

The compiler is integrated with the editor. When the user issues the 'compile' command, and the program is found to contain a compile-time error, the erroneous phrase is highlighted, ready for immediate editing. If the program contains several errors, then the compiler will list all of them, and the user can select a particular error message and have the relevant phrase highlighted.

The object program is also integrated with the editor. If the program fails at runtime, the failing phrase is highlighted. (Of course, this phrase is not necessarily the one that contains the logical error. But it would be unreasonable to expect the language processor to debug the program automatically!)

3.1. Implementations of Programming Language Processors

A simplistic but not inaccurate view of the language implementation process suggests that some sort of translator program is required (a *compiler*) to transform the high-level language program into a semantically equivalent machine code program that can run on the target machine. Other software, such as libraries, will probably also be required. As the complexity of the source language increases as the language becomes "higher and higher-level", closer to human expression, one would expect the complexity of the translator to increase too.

Many routes are possible, and it may be the characteristics of the high-level language that forces different approaches. For example, the traditional way of implementing Java makes use of the Java Virtual Machine (JVM), where the compiler translates from Java source code into JVM code and a separate program (an *interpreter*) reads these virtual machine instructions, emulating the actions of the virtual machine, effectively running the Java program. This seemingly contrary implementation method does have significant benefits. In particular it supports Java's feature of dynamic class loading. Without such an architecture-neutral virtual machine code, implementing dynamic class loading would be very much more difficult. More generally, it allows the support of *reflection*, where a Java program can examine or modify at runtime the internal properties of the executing program.

Interpreted approaches are very appropriate for the implementation of some programming languages. Compilation overheads are reduced at the cost of longer runtimes. The programming language implementation field is full of trade-offs.

To make effective use of a high-level language, it is essential to know something about its implementation. In some demanding application areas such as *embedded systems* where a computer system with a fixed function is controlling some electronic or mechanical device, there may be severe demands placed on the embedded controller and the executing code. There may be real-time constraints (for example, when controlling the ignition timing in a car engine where a predefined set of operations has to complete in the duration of a spark), memory constraints (can the whole program fit in the 64 k bytes available on the cheap version of the microcontroller chip?) or power consumption constraints (how often do I have to charge the batteries in my mobile phone?). These constraints make demands on the performance of the hardware but also on the way in which the high-level language implementing the system's functionality is actually implemented. The designers need to have an in-depth knowledge of the implementation to be able to deal with these issues.

3.1.1. Compilers

The compiler is a program translating from a *source language* to a *target language*, implemented in some *implementation language*. As we have seen, the traditional view of a compiler is to take some high-level language as input and generate machine code for some target machine.

The field of compilation is not restricted to the generation of low-level language code. Compiler technology can be developed to translate from one high-level language to another. For example, some of the early C++ compilers generated C code rather than target machine code. Existing C compilers were available to perform the final step.

The complexity of a compiler is not only influenced by the complexities of the source and target languages, but also by the requirement for optimized target code. There is a real danger in compiler development of being overwhelmed by the complexities and the details of the task. A well-structured approach to compiler development is essential.

The design of compilers is influenced by the characteristics of formal language specifications, by automata theory, by parsing algorithms, by processor design, by data structure and algorithm design, by operating system services, by the target machine instruction set and other hardware features, by implementation language characteristics and so on, as well as by the needs of the compilers' users. Coding a compiler can be a daunting software task, but the process is greatly simplified by making use of the approaches, experiences, recommendations and algorithms of other compiler writers.

It is helpful to say that a compiler can be built of two distinct phases. The first is the *analysis* phase, reading the source program, creating internal data structures reflecting its syntactic and semantic structure according to the definition of the language. The second is the *synthesis* phase, generating code for a machine from the data structures created by the analysis phase. Thinking about a compiler in terms of these two distinct phases can greatly simplify both design and implementation.

This subdivision, although simple, has major consequences for the design of compilers. The interface between these two phases is some intermediate language, loosely "mid-way" between the source and target languages. If this intermediate language is designed with care, it may then be possible to structure the compiler so that the analysis phase becomes target machine independent and the synthesis phase becomes source language independent. This, in theory, allows great potential savings in implementation effort in developing new compilers. If a compiler structured in this way needs to be retargeted to a new machine architecture, then only the synthesis phase needs to be modified or rewritten. The analysis phase should not need to be touched. Similarly, if the compiler needs to be modified to compile a different source language (targeting the same machine), then only the analysis phase needs to be modified or rewritten.

The lexical and syntax analyzer do similar things. They both group together characters or tokens into larger syntactic units. So there is an issue about whether a particular syntactic structure should be recognized by the lexical analyzer or by the syntax analyzer. The traditional approach, and it is an approach that works well, is to recognize the simpler structures in the lexical analyzer, specifically those that can be expressed in terms of a Chomsky type 3 grammar. Syntactic structures specified by a Chomsky type 2 or more complex grammar are then left for resolution by the syntax analyzer. In theory,

the syntax analyzer could deal with the lexical tokens using a Chomsky type 2 grammar parsing approach, but this would add significantly to the complexity of the syntax analyzer. Furthermore, by leaving the lexical analyzer to deal with these tokens improves compiler efficiency because simpler and faster type 3 parsing techniques can be used.

The lexical analysis, syntax analysis, semantic analysis and the machine-independent optimization phases together form the front-end of the compiler and the code generation and machine-dependent phases form the back-end. These phases all have specific and distinct roles. And to support the design of these individual modules, the interfaces between them have to be defined with care.

The language implementation does not stop at the compiler. The support of collections of library routines is always required, providing the environment in which code generated by the compiler can run. Other tools such as debuggers, linkers, documentation aids and interactive development environments are needed too. This is no easy task.

Dealing with this complexity requires a strict approach to design in the structuring of the compiler construction project. Traditional techniques of software engineering are well applied in compiler projects, ensuring appropriate modularization, testing, interface design and so on. Extensive stage-by-stage testing is vital for a compiler.

To ease the task of producing a programming language implementation, many software tools have been developed to help generate parts of a compiler or interpreter automatically. For example, lexical analyzers and syntax analyzers (two early stages of the compilation process) are often built with the help of tools taking the formal specification of the syntax of the programming language as input and generating code to be incorporated in the compiler as output. The modularization of compilers has also helped to reduce workload.

For example, many compilers have been built using a target machine independent *front-end* and a source language-independent *back-end* using a standard intermediate representation as the interface between them. Then front-ends and back-ends can be mixed and matched to produce a variety of complete compilers. Compiler projects rarely start from scratch today.

3.1.2. Interpreters

Running a high-level language program using a compiler is a two-stage process. In the first stage, the source program is translated into target machine code and in the second stage, the hardware executes or a virtual machine interprets this code to produce results. Another popular approach to language implementation generates no target code. Instead, an *interpreter* reads the source program and "executes" it directly. So if the interpreter encounters the source statement `a=b+1`, it analyses the source characters to determine that the input is an assignment statement, it extracts from its own data structures the value of `b`, adds one to this value and stores the result in its own record of `a`.

This is analogous to what you'd do if you were handed a C program and told to execute it by hand. You would read each statement in the proper sequence and mentally do what it says. You probably would keep track of the values of the program's variables on a sheet of scratch paper, and you'd write down each line of program output until you've completed the program execution. Essentially, you would have just done what a C interpreter does. A C interpreter reads in a C program and executes it. There is no object program to generate and load. Instead, an interpreter translates a program into the actions that result from executing the program immediately. *Immediacy* is the key characteristic of interpretation; there is no prior time-consuming translation of the source program into a low-level representation.

In an interactive environment, immediacy is highly advantageous. For example, the user of a command language expects an immediate response from each command; it would be absurd to expect the user to enter an entire sequence of commands before seeing the response from the first one. Similarly, the user of a database query language expects an immediate answer to each query. In this mode of working, the 'program' is entered once and then discarded.

The user of a programming language, on the other hand, is much more likely to retain the program for further use, and possibly further development. Even so, translation from the programming language to an intermediate language followed by interpretation of the intermediate language (i.e., *interpretive compilation*) is a good alternative to *full compilation*, especially during the early stages of program development.

How do you decide when to use an interpreter and when to use a compiler? When you feed a source program into the interpreter, the interpreter takes over to check and execute the program. A compiler also checks the source program but instead generates object code. After running the compiler, you need to run the linker to generate the object program, and then you have to load the object program into memory to execute it. If the compiler generates an assembly language object code, you must also run an assembler. So, an interpreter definitely requires less effort to execute a program.

Interpreters can be more versatile than compilers. You can use Java to write a C interpreter that runs on a Microsoft Windows-based PC, an Apple Macintosh, and a Linux box, so that the interpreter can execute C source programs on any of those platforms. A compiler, however, generates object code for a particular computer. Therefore, even if you took a C compiler originally written for the PC and made it run on the Mac, it would still generate object code for the PC. To make the compiler generate object code for the Mac, you would have to rewrite substantial portions of the compiler. You can get around this problem by making your compiler generate object code for the Java virtual machine. This virtual machine runs on various computer platforms.

What happens if the source program contains a logic error that doesn't show up until runtime, such as an attempt to divide by a variable whose value is zero?

Since an interpreter is in control when it is executing the source program, it can stop and tell you the line number of the offending statement and the name of the variable. It can even prompt you for some corrective action before resuming execution of the program, such as changing the value of the variable to something other than zero. An interpreter can include an interactive *source-level debugger*, also known as a *symbolic debugger*. *Symbolic* means the debugger allows you to use symbols from the source program, such as variable names.

On the other hand, the object program generated by a compiler and a linker generally runs by itself. Information from the source program such as line numbers and names of variables might not be present in the object program. When a runtime error occurs, the program may simply abort and perhaps print a message containing the address of the bad instruction. Then it's up to you to figure out the corresponding source statement and which variable's value was zero.

So when it comes to debugging, an *interpreter* is usually the way to go. Some compilers can generate extra information in the object code so that if a runtime error occurs, the object program can print out the offending statement line number or the variable name. You then fix the error, recompile, and rerun the program. Generating extra information in the object code can cause the object program to run slower. This may induce you to turn off the debugging features when you're about to compile the "production" version of your program. This situation has been compared to wearing a life jacket when you're learning to sail on a lake, and then taking the jacket off when you're about to go out into the ocean.

Suppose you've successfully debugged your program, and now your primary concern is how fast it executes. Since a computer can execute a program in its native machine language at top speed, a compiled program can run orders of magnitude faster than an interpreted source program. A compiler is definitely the winner when it comes to speed. This is certainly true in the case of an optimizing compiler that knows how to generate especially efficient code.

So, whether you should use a compiler or an interpreter depends on what aspects of program development and execution are important. The best of both worlds would include an interpreter with an interactive symbolic debugger to use during program development and a compiler to generate machine language code for fast execution after you've debugged the program.

Interpretation is sensible when most of the following circumstances exist:

- The programmer is working in interactive mode, and wishes to see the results of each instruction before entering the next instruction.
- The program is to be used once and then discarded (i.e., it is a 'throw-away' program), and therefore running speed is not very important.
- Each instruction is expected to be executed only once (or at least not very frequently).
- The instructions have simple formats, and thus can be analyzed easily and efficiently.

Interpretation is very slow. Interpretation of a source program, in a high-level language, can be up to 100 times slower than running an equivalent machine-code program. Therefore interpretation is not sensible when:

- The program is to be run in production mode, and therefore speed is important.
- The instructions are expected to be executed frequently.
- The instructions have complicated formats, and are therefore time-consuming to analyze. (This is the case in most high-level languages.)

4. PRACTICAL PROBLEMS OF INTERPRETATION

The process of source-level interpretation sounds attractive because there is no need for the potentially complex implementation of code generation. But there are practical problems. The first problem concerns performance. If a source statement is executed repeatedly it is analyzed each time, before each execution. The cost of possibly multiple statement analysis followed by the interpreter emulating the action of the statement will be many times greater than the cost of executing a few machine instructions obtained from a compilation of `a=b+1`. However this cost can be reduced fairly easily by only doing the analysis of the program once, translating it into an intermediate form that is subsequently interpreted. Many languages have been implemented in this way, using an interpreted intermediate form, despite the overhead of interpretation.

The second problem concerns the need for the presence of an interpreter at runtime. When the program is "executing" it is located in the memory of the target system in source or in a post-analysis intermediate form, together with the interpreter program. It is likely that the total memory footprint is much larger than that of equivalent compiled code. For small, embedded systems with very limited memory this may be a decisive disadvantage.

All programming language implementations are in some sense interpreted. With source code interpretation, the interpreter is complex because it has to analyses the source language statements and then emulate their execution. With intermediate code interpretation, the interpreter is simpler because the source code analysis has been done in advance. With the traditional compiled approach with the generation of target machine code, the interpretation is done entirely by the target hardware, there is no software interpretation and hence no overhead. Looking at these three levels of interpretation in greater detail, one can easily identify trade-offs:

- Source-level interpretation: interpreter complexity is high, the runtime efficiency is low (repeated analysis and emulation of the source code statements), the initial compilation cost is zero because there is no separate compiler and hence the delay in starting the program's execution is also zero.

- Intermediate code interpretation: interpreter complexity is lower, the runtime efficiency is improved (the analysis and emulation of the intermediate code statements is comparatively simple), there is an initial compilation cost and hence there is a delay in starting the program.
- Target code interpretation/full compilation: there is no need for interpreter software so interpreter complexity is zero, the runtime efficiency is high (the interpretation of the code is done directly by the hardware), there is a potentially large initial compilation cost and hence there may be a significant delay in starting the program.

The different memory requirements of the three approaches are somewhat harder to quantify and depend on implementation details. In the source-level interpretation case, a simplistic implementation would require both the text of the source code and the (complex) interpreter to be in main memory. The intermediate code interpretation case would require the intermediate code version of the program and the (simpler) interpreter to be in main memory. And in the full compilation case, just the compiled target code would need to be in main memory. This, of course, takes no account of the memory requirements of the running program —space for variables, data structures, buffers, library code, etc.

There are other trade-offs. For example, when the source code is modified, there is no additional compilation overhead in the source-level interpretation case, whereas in the full compilation case, it is usual for the entire program or module to be recompiled. In the intermediate code interpretation case, it may be possible to just recompile the source statements that have changed, avoiding a full recompilation to intermediate code.

Finally, it should be emphasized that this issue of lower efficiency of interpreted implementations is rarely a reason to dismiss the use of an interpreter. The interpreting overhead in time and space may well be irrelevant, particularly in larger computer systems, and the benefits offered may well overwhelm any efficiency issues.

5. TERMINOLOGICAL CONSIDERATIONS

It used to be easier to explain the differences between an interpreter and a compiler. With the growing popularity of virtual machines, the picture gets a bit fuzzy.

A virtual machine is a program that simulates a computer. This program can run on different actual computer platforms. For example, the Java virtual machine (JVM) can run on a Microsoft Windows-based PC, an Apple Macintosh, a Linux system, and many others.

The virtual machine has its own virtual machine language, and the machine language instructions are interpreted by the actual computer that's running the virtual machine. So if you write a translator

that translates a C source program into virtual machine language that is interpreted, is the translator a compiler or an interpreter?

Rather than splitting hairs, let's agree that if a translator translates a source program into machine language, whether for an actual computer or for a virtual machine, the translator is a *compiler*. A translator that executes the source program without first generating machine language is an *interpreter*.

There are many implementations of high-level programming languages where the *compiler* generates code for a virtual machine and then a separate program, the *interpreter*, reads the virtual machine code and emulates the execution of the virtual machine, instruction by instruction. At first sight this may seem a strange thing to do —why not generate target machine code directly? But this approach has several significant advantages, including the following:

- The design of the code generated by the compiler is not constrained by the architecture of the target machine. This can simplify the design of the compiler because it does not have to deal with the quirks and restrictions of the real hardware. For example, it may be convenient to use a stack-based architecture for the virtual machine, not directly supported by the target hardware.
- Portability is enhanced. If the interpreter is written in a portable language, the interpreter and the virtual machine code can be shipped and easily run on machines with different architectures or operating systems. This ties in well with today's prevalence of heterogeneous networked environments.
- The virtual machine code can be designed to be particularly compact. There are application areas where this may be very important.
- Runtime debugging and monitoring features can be incorporated in the virtual machine interpreter allowing improved safety and security in program execution.
- The virtual machine code can run in a sandbox, preventing it from performing illegal operations. For example, the virtual machine can operate on typed data, and runtime type checking can provide helpful debugging information.

The obvious disadvantage of this approach concerns the question of efficiency. Interpreted code is likely to be slower than native execution. But for most applications this turns out not to be of real significance. The advantages usually easily outweigh this disadvantage and so many modern programming languages are implemented in this way.

The nature of the virtual machine poses interesting questions. The design of the virtual machine should not be too close to that of the hardware because the advantages of compiler simplification essentially disappear. But if the virtual machine's design is made very close or identical to the language that is being implemented, the compiler is made very simple, but the interpreter has to deal with the detail of decoding and analyzing this potentially complex code. The functions of the compiler are being shifted into the interpreter. However, several languages have been implemented successfully in this way,

where the interpreter does all the work, removing the necessity for the separate compiler. Some implementations of the BASIC language have been implemented in this way. Because of the need for repeated analysis of the source language statements in the interpreter, this is rarely a practical approach for a production system.

6. FUNCTION OF INTERPRETERS

Interpreters are magical. On the surface they look deceptively simple: text goes in and something comes out. They are programs that take other programs as their input and produce something. Simple, right? But the more you think about it, the more fascinating it becomes. Seemingly random characters —letters, numbers and special characters— are fed into the interpreter and suddenly become *meaningful*. The interpreter gives them meaning! It makes sense out of nonsense. And the computer, a machine that's built on understanding ones and zeroes, now understands and acts upon this weird language we feed into it —thanks to an interpreter that translates this language while reading it.

It's difficult to make generic statements about interpreters since the variety is so high and none are alike. What can be said is that the one fundamental attribute they all share is that they take source code and evaluate it without producing some visible, intermediate result that can later be executed. That's in contrast to compilers, which take source code and produce output in another language that the underlying system can understand.

Here are some well-known examples of interpreters:

- *A Basic interpreter*: Basic has expressions and assignment commands like other high-level languages. But its control structures are low-level: a program is just a sequence of commands linked by conditional and unconditional jumps. A Basic interpreter fetches, analyzes, and executes one command at a time.
- *A Lisp interpreter*: Lisp is a very unusual language in that it assumes a common data structure (trees) for both code and data. Indeed, a Lisp program can manufacture new code at run-time! The Lisp program structure lends itself to interpretation.
- *The UNIX command language interpreter (shell)*: A UNIX user instructs the operating system by entering textual commands. The shell program reads each command, analyzes it to extract a command-name together with some arguments, and executes the command by means of a system call. The user can see the results of a command before entering the next one. The commands constitute a command language, and the shell is an interpreter for that command language.
- *An SQL interpreter*: SQL is a database query language. The user extracts information from the database by entering an SQL query, which is analyzed and executed immediately. This is done by an SQL interpreter within the database management system.

Some interpreters are really tiny, and do not even bother with a parsing step. They just interpret the input right away. Look at one of the many *Brainfuck* interpreters out there to see what I mean.

On the other end of the spectrum are much more elaborate types of interpreters. Highly optimized and using advanced parsing and evaluation techniques. Some of them don't just evaluate their input, but compile it into an internal representation called bytecode and then evaluate this. Even more advanced are JIT interpreters that compile the input just-in-time into native machine code that gets then executed.

But then, in between those two categories, there are interpreters that parse the source code, build an abstract syntax tree (AST) out of it and then evaluate this tree. This type of interpreter is sometimes called "tree-walking" interpreter, because it "walks" the AST and interprets it.

In order to work with source code, it must be turned into a more accessible form. As easy as plain text is to work with in an editor, it becomes cumbersome pretty fast when trying to interpret it in a programming language as another programming language.

So, the representation of the source code must be changed two times before it's evaluated (Fig. 2):

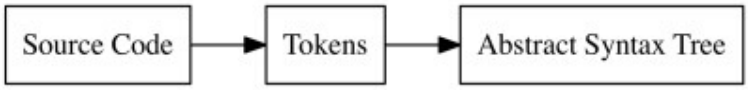


Figure 2. Changes of the source code before evaluation

The first transformation, from source code to tokens, is called "lexical analysis", or "lexing" for short. It's done by a lexical analyzer or *lexer* (also called *tokenizer* or *scanner* —some use one word or the other to denote subtle differences in behavior), which reads the characters of the source program and groups them together into a stream of lexical tokens. Each lexical token is a small, easily categorizable data structure representing a basic syntactic component of the programming language being processed. There are tokens such as numbers, identifiers, punctuation, operators, strings, reserved words and so on. Comments can be ignored unless the language defines special syntactic components encoded in comments that may be needed later. White space (spaces, tabs, newlines, etc.) may be ignored except, again, where they have some syntactic significance (where spacing indicates block structure, where white space may occur in character strings and so on). The syntax of these basic lexical tokens is usually simple, and the expectation is that the syntax can be specified formally in terms of a Chomsky type 3 grammar (i.e. in terms of regular expressions). This considerably simplifies the coding of the lexical analyzer.

Tokens are then fed to the syntax analyzer or *parser*. The interface could be such that the lexical analyzer tokenizes the entire input file and then passes the whole list of tokens to the syntax analyzer. Alternatively, the tokens could be passed on to the syntax analyzer one at a time, when demanded by the syntax analyzer. The parser then does the second transformation and turns the tokens into an "Abstract Syntax Tree".

Everyone who has ever programmed has probably heard about parsers, mostly by encountering a "parser error". Or maybe even said something like "we need to parse this", "the parser blows up with this input". The word "parser" is as common as "interpreter" and "programming language". Everyone knows that parsers *exist*. They have to, right? Because who else would be responsible for "parser errors"?

But what is a parser exactly? How does it do its job? This is what Wikipedia has to say:

> A parser is a software component that takes input data (frequently text) and builds a data structure —often some kind of parse tree, abstract syntax tree or other hierarchical structure— giving a structural representation of the input, checking for correct syntax in the process. […] The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters.

For a Wikipedia article about a computer science topic this excerpt is remarkably easy to understand. Even the lexer can be recognized in there!

A parser turns its input into a data structure that represents the input. That sounds pretty abstract, so let's illustrate this with an example. Here is a little bit of JavaScript:

```
> var input = '{"name": "Luke", "age": 23}';
> var output = JSON.parse(input);
> output
{ name: "Luke", age: 23 }
> output.name
"Luke"
> output.age
23
```

The input is just some text, a string. It's passed to a parser hidden behind the `JSON.parse` function and receive an output value. This output is the data structure that represents the input: a JavaScript object with two fields named `name` and `age`, their values also corresponding to the input. A JSON parser takes text as input and builds a data structure that represents the input. That's exactly what the parser of a programming language does. The difference is that in the case of a JSON parser you can see the data structure when looking at the input. Whereas if you look at this

```
if ((5 + 2 * 3) == 91) { return computeStuff(input1, input2); }
```

it's not immediately obvious how this could be represented with a data structure. Users of programming languages seldom get to see or interact with the parsed source code, with its internal representation. Lisp programmers are the exception to the rule —in Lisp the data structures used to represent the source code are the ones used by a Lisp user. The parsed source code is easily accessible as data in the program. "Code is data, data is code" is something heard a lot from Lisp programmers.

In most interpreters and compilers the data structure used for the internal representation of the source code is called a "syntax tree" or an "abstract syntax tree" (AST for short). The "abstract" is based on the fact that certain details visible in the source code are omitted in the AST. Semicolons, newlines, whitespace, comments, braces, bracket and parentheses —depending on the language and the parser these details are not represented in the AST, but merely guide the parser when constructing it.

A fact to note is that there is not one true, universal AST format that's used by every parser. Their implementations are all pretty similar, the concept is the same, but they differ in details. The concrete implementation depends on the programming language being parsed.

So, this is what parsers do. They take source code as input and produce a data structure which represents this code. While building up the data structure, they unavoidably analyze the input, checking that it conforms to the expected structure. Thus the process of parsing is also called *syntactic analysis*.

But the analysis phase is not quite complete even after the syntax tree has been constructed. Traditional Chomsky type 2 grammars used to build the syntax analyzer cannot deal directly with contextual issues such as type checking, declaration and scopes of names, choice of overloaded operators and so on. This is the role of the *semantic analysis* phase. Traversing the tree, inserting and checking type information is done here. Typed languages may require that all or almost all the nodes in the tree be labelled with a data type. Complexity is increased when the language being compiled allows user-defined types. Rules for type compatibility have to be applied here too. For example, does the language allow an integer value to be assigned to a real (floating point) variable?

The evaluation process of an interpreter is where code becomes meaningful. Without evaluation an expression like `1 + 2` is just a series of characters, tokens, or a tree structure that represents this expression. It doesn't mean anything. Evaluated, of course, `1 + 2` becomes `3`, `5 > 1` becomes `true`, `5 < 1` becomes `false`, and `puts("Hello World!")` becomes the friendly message we all know.

The evaluation process defines how the programming language being interpreted works.

```
let num = 5;
if (num) {
    return a;
} else {
    return b;
}
```

Whether this returns `a` or `b` depends on the decision of the interpreter's evaluation process whether the integer `5` is *truthy* or not. In some languages it's truthy, in others we'd need to use an expression that produces a *boolean* like `5 != 0`.

Consider this:

```
let one = fn() {
    printLine("one");
    return 1;
};

let two = fn() {
    printLine("two");
    return 2;
};

add(one(), two());
```

Does this first output `one` and then `two` or the other way around? It depends on the specification of the language and ultimately on the implementation of its interpreter and in which order it evaluates the arguments in a call expression.

Evaluation is also where interpreter implementations (regardless of which language they're interpreting) diverge the most. There are a lot of different strategies to choose from when evaluating source code. It's also worth noting again that the line between interpreters and compilers is a blurry one. The notion of an interpreter as something that doesn't leave executable artifacts behind (in contrast to a compiler, which does just that) gets fuzzy real fast when looking at the implementations of real-world and highly-optimized programming languages.

The so-called *environment model of evaluation* has two basic parts:

1. To *evaluate* a combination (a compound expression), evaluate the subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.
2. To *apply* a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment.

These two rules describe the essence of the evaluation process, a basic cycle in which expressions to be evaluated in environments are reduced to procedures to be applied to arguments, which in turn are reduced to new expressions to be evaluated in new environments, and so on, until we get down to symbols, whose values are looked up in the environment, and to primitive procedures, which are applied directly. This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, `eval` and `apply`. The implementation of the evaluator will depend upon procedures that define the syntax of the expressions to be evaluated.

The *environment model of evaluation* is similar to how a human evaluates a mathematical expression: We insert the values of variables in the expression and evaluate it bit by bit, starting with the innermost parentheses and moving out until we have the result of the expression. We can then repeat the process with other values for the variables.

There are some differences, however. Where a human being will copy the text of the formula with variables replaced by values, and then write a sequence of more and more reduced copies of a formula until it is reduced to a single value, an interpreter will keep the formula (or, rather, the abstract syntax tree of an expression) unchanged and use a symbol table to keep track of the values of variables. Instead of reducing a formula, the interpreter is a function that takes an abstract syntax tree and a symbol table as arguments and returns the value of the expression represented by the abstract syntax tree. The function can call itself recursively on parts of the abstract syntax tree to find the values of subexpressions, and when it evaluates a variable, it can look its value up in the symbol table.

This process can be extended to also handle statements and declarations, but the basic idea is the same: A function takes the abstract syntax tree of the program and, possibly, some extra information about the context (such as a symbol table or the input to the program) and returns the output of the program. Some input and output may be done as side effects by the interpreter.

With that said, the most obvious and classical choice of what to do with the AST is to just traverse it (often in *post-order*), visit each node and do what the node signifies: print a string, add two numbers, execute a function's body —all on the fly. Interpreters working this way are called "tree-walking interpreters" and are the archetype of interpreters. Sometimes their evaluation step is preceded by small optimizations that rewrite the AST (e.g. remove unused variable bindings) or convert it into another *intermediate representation* (IR) that's more suitable for recursive and repeated evaluation.

Other interpreters also traverse the AST, but instead of interpreting the AST itself they first convert it to bytecode. Bytecode is another IR of the AST and a really dense one at that. The exact format and of which op-codes (the instructions that make up the bytecode) it's composed of varies and depends on the guest and host programming languages. In general though, the op-codes are pretty similar to the mnemonics of most assembly languages; it's a safe bet to say that most bytecode definitions contain op-codes for `push` and `pop` to do stack operations. But bytecode is not native machine code, nor is it assembly language. It can't and won't be executed by the operating system and the CPU of the machine the interpreter is running on. Instead it's interpreted by a virtual machine, that's part of the interpreter. Just like VMware and VirtualBox emulate real machines and CPUs, these virtual machines emulate a machine that understands this particular bytecode format. This approach can yield great performance benefits.

A variation of this strategy doesn't involve an AST at all. Instead of building an AST the parser emits bytecode directly. Now, are we still talking about interpreters or compilers? Isn't emitting bytecode that gets then interpreted (or should we say "executed"?) a form of compilation? The line becomes blurry. And to make it even more fuzzy, consider this: some implementations of programming languages parse the source code, build an AST and convert this AST to bytecode. But instead of executing the operations specified by the bytecode directly in a virtual machine, the virtual machine then compiles the bytecode

to native machine code, right before its executed —just in time. That's called a JIT (for "just in time") interpreter/compiler.

Others skip the compilation to bytecode. They recursively traverse the AST but before executing a particular branch of it the node is compiled to native machine code. And then executed. Again, "just in time".

A slight variation of this is a mixed mode of interpretation where the interpreter recursively evaluates the AST and only after evaluating a particular branch of the AST multiple times does it compile the branch to machine code.

Amazing, isn't it? So many different ways to go about the evaluation, so many twists and variations.

The choice of which strategy to choose largely depends on performance and portability needs, the programming language that's being interpreted and how far you're willing to go. A tree-walking interpreter that recursively evaluates an AST is probably the slowest of all approaches, but easy to build, extend, reason about and as portable as the language it's implemented in.

An interpreter that compiles to bytecode and uses a virtual machine to evaluate said bytecode is going to be a lot faster. But more complicated and harder to build, too. Throw JIT compilation to machine code into the mix and now you also need to support multiple machine architectures if you want the interpreter to work on both ARM and x86 CPUs.

All of these approaches can be found in real-world programming languages. And most of the time the chosen approach changed with the lifetime of the language. Ruby is a great example here. Up to and including version 1.8 the interpreter was a tree-walking interpreter, executing the AST while traversing it. But with version 1.9 came the switch to a virtual machine architecture. Now the Ruby interpreter parses source code, builds an AST and then compiles this AST into bytecode, which gets then executed in a virtual machine. The increase in performance was huge.

The WebKit JavaScript engine JavaScriptCore and its interpreter named "Squirrelfish" also used AST walking and direct execution as its approach. Then in 2008 came the switch to a virtual machine and bytecode interpretation. Nowadays the engine has four (!) different stages of JIT compilation, which kick in at different times in the lifetime of the interpreted program – depending on which part of the program needs the best performance.

Another example is Lua. The main implementation of the Lua programming language started out as an interpreter that compiles to bytecode and executes the bytecode in a register-based virtual machine. 12 years after its first release another implementation of the language was born: LuaJIT. The clear goal of Mike Pall, the creator of LuaJIT, was to create the fastest Lua implementation possible. And he did. By JIT compiling a dense bytecode format to highly-optimized machine code for different architectures the LuaJIT implementation beats the original Lua in every benchmark. And not just by a tiny bit, no; it's sometimes 50 times faster.

7. DESIGN OF INTERPRETERS

The conceptual design of a program is a high-level view of its software architecture. The conceptual design includes the primary components of the program, how they're organized, and how they interact with each other. It does not necessarily say how these components will be implemented. Rather, it allows you to examine and understand the components first without worrying about how you're eventually going to develop them.

Both compilers and interpreters are programming language translators. A compiler translates a source program into machine language, and an interpreter translates the program into actions. Such a translator, as seen at the highest level, consists of a *front end* and a *back end*. Following the principle of software reuse, a C compiler and a C interpreter can share the same front end, but they'll each have a different back end.

Figure 3 shows the conceptual design of compilers and interpreters. If everything is designed properly, only the front end needs to know which language the source programs are written in, and only the back end needs to distinguish between a compiler and an interpreter.

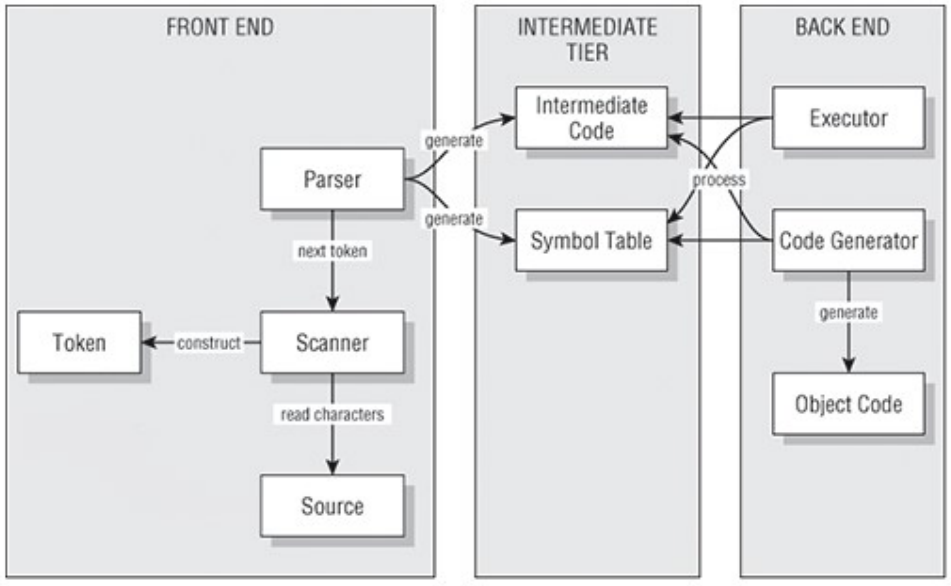


Figure 3. Conceptual design of compilers and interpreters

In this figure, an arrow represents a command issued by one component to another. The parser tells the scanner to get the next token. Then the scanner reads characters from the source and constructs a new token.

The front end of a translator reads the source program and performs the initial translation stage. Its primary components are the *parser*, the *scanner*, the *token*, and the *source*.

The parser controls the translation process in the front end. It repeatedly asks the scanner for the next token, and it analyzes the sequences of tokens to determine what high-level language elements it is

translating, such as arithmetic expressions, assignment statements, or procedure declarations. The parser verifies that what it sees is syntactically correct as written in the source program; in other words, the parser detects and flags any syntax errors. What the parser does is called *parsing*, and the parser *parses* the source program to translate it.

The scanner reads the characters of the source program sequentially and constructs *tokens*, which are the low-level elements of the source language. For example, C tokens include *reserved words* such as `if`, `switch`, `case`, `for`, `do` and `while`, *identifiers* that are names of variables and functions, and *special symbols* such as `= + - *` and `/`. What the scanner does is called *scanning*, and the scanner *scans* the source program to break it apart into tokens.

A compiler ultimately translates a source program into machine language object code, so the primary component of its back end is a *code generator*. An interpreter executes the program, so the primary component of its back end is an *executor*.

If the compiler and the interpreter are to share the same front end, their different back ends need a common intermediate interface with the front end. Recall that the front end performs the initial translation stage. The front end generates intermediate code and a *symbol table* in the *intermediate tier* that serve as the common interface.

Intermediate code is a predigested form of the source program that the back end can process efficiently. The intermediate code is a data structure (often an in-memory tree) that represents the statements of the source program. This annotated syntax tree still bears very much the traces of the source language and the programming paradigm it belongs to: higher-level constructs like for-loops, method calls, list comprehensions, logic variables, and parallel select statements are all still directly represented by nodes and subtrees. It is worth noting that the methods used to obtain the annotated syntax tree are largely language- and paradigm-independent.

The symbol table contains information about the symbols (such as the identifiers) contained in the source program. A compiler's back end processes the intermediate code and the symbol table to generate the machine language version of the source program. An interpreter's back end processes the intermediate code and the symbol table to execute the program. In processing the intermediate code, the choice is between little preprocessing followed by execution on an interpreter, and much preprocessing, in the form of machine code generation, followed by execution on hardware. It is worth to note that the specifics of these transformations and the run-time features they require are for a large part language- and paradigm-dependent —although the techniques by which they are applied will often be similar.

To further software reuse, the intermediate code and the symbol table structures can be designed to be language independent. In other words, the same structures can be used for different source languages. Therefore, the back end will also be language independent; when it processes these structures, it doesn't need to know or care what the source language was.

8. INTERMEDIATE REPRESENTATIONS

Compilers and interpreters are typically organized as a series of passes. As the compiler or interpreter derives knowledge about the code it processes, it must convey that information from one pass to another. Thus, the compiler or interpreter needs a representation for all of the facts that it derives about the program. We call this representation an *intermediate representation*, or IR. A compiler or interpreter may have a single IR, or it may have a series of IRs that it uses as it transforms the code from source language into its target language. During translation, the IR form of the input program is the definitive form of the program. The compiler or interpreter does not refer back to the source text; instead, it looks to the IR form of the code. The properties of a compiler's or an interpreter's IR or IRs have a direct effect on what it can do to the code.

Almost every phase of the compiler or interpreter manipulates the program in its IR form. Thus, the properties of the IR, such as the mechanisms for reading and writing specific fields, for finding specific facts or annotations, and for navigating around a program in IR form, have a direct impact on the ease of writing the individual passes and on the cost of executing those passes.

Most passes in the compiler or interpreter consume IR; the scanner is an exception. Most passes produce IR; passes in the compiler's code generator or in the interpreter's executor can be exceptions. Many modern compilers and interpreters use multiple IRs. In a pass-structured compiler or interpreter, the IR serves as the primary and definitive representation of the code.

An IR must be expressive enough to record all of the useful facts that the compiler or interpreter might need to transmit between passes. Source code is insufficient for this purpose; the compiler or interpreter derives many facts that have no representation in source code. To record all of the detail that the compiler or interpreter must encode, most writers of compilers or interpreters augment the IR with tables and sets that record additional information. These tables may be considered part of the IR.

Selecting an appropriate IR for a project requires an understanding of the source language, the target machine, and the properties of the applications that the compiler or interpreter will process. For example, a source-to-source translator might use an IR that closely resembles the source code, while a compiler that produces assembly code for a microcontroller might obtain better results with an assembly-code-like IR. Similarly, a compiler for C might need annotations about pointer values that are irrelevant in a compiler for Perl, and a Java compiler keeps records about the class hierarchy that have no counterpart in a C compiler.

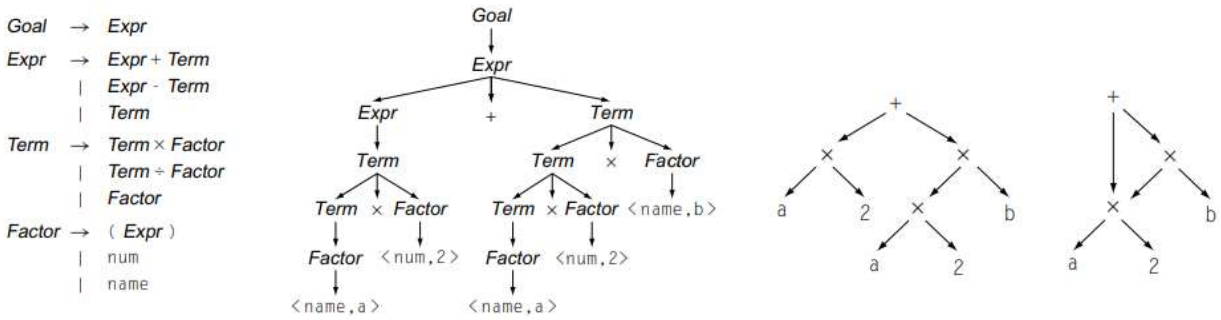
Implementing an IR forces the writer of a compiler or interpreter to focus on practical issues. The compiler or interpreter needs inexpensive ways to perform the operations that it does frequently. It needs concise ways to express the full range of constructs that might arise during compilation or interpretation. The writer of a compiler or interpreter also needs mechanisms that let humans examine the IR program easily and directly. Self-interest should ensure that writers of compilers or interpreters pay heed to this

last point. Finally, compilers or interpreters that use an IR almost always make multiple passes over the IR for a program. In the case of a compiler, the ability to gather information in one pass and use it in another improves the quality of code that it can generate.

Compilers and interpreters have used many kinds of IR. Broadly speaking, IRs fall into three structural categories:

- *Linear IRs* resemble pseudo-code for some abstract machine. The algorithms iterate over simple, linear sequences of operations. There are advantages for the use of a linear IR. For example, it should be easy to generate target code (not necessarily optimized) from linear code by translating statement by statement. A wide range of linear IRs are used today, but it is also possible to adopt existing high-level languages as the IR. For example, C has been used in many compilers as an IR. The generation of C from the syntax tree should be easy, and this C can be compiled by an existing C compiler to produce an executable program, without having to write a conventional code generator at all. Generating the IR from the abstract syntax tree using just a simple prefix or postfix traversal will also yield a form of linear IR. Such IRs were sometimes used in the early days of compilers, particularly for stack-based target hardware. These representations have advantages of simplicity but they are not so popular now because they do not work quite so well with the powerful optimization algorithms used in today's compilers. Although stack-based IRs, such as the Java Virtual Machine, are used, being interpreted or translated into a target machine code, non-stack based linear IRs based on instructions containing an operator, up to two arguments and a result have become more popular.
- *Graphical IRs* encode the compiler's knowledge in a graph. The algorithms are expressed in terms of graphical objects: nodes, lists, or trees. Graph-based IRs are popular too, again with a variety of designs. Some of these IR designs are allied to particular programming languages or language types, some are good for particular types of optimizations and a few try to be general-purpose, appropriate for many source languages. Although it is convenient to have a human-readable linear IR, using more powerful data structures may offer advantages. The parse tree or *syntax tree* data structure produced by the parser can be regarded as an IR in itself. Figure 4 shows a classic expression grammar (a) alongside a *syntax tree* (b) for `a × 2 + a × 2 × b`. The parse tree is large relative to the source text because it represents the complete derivation, with a node for each grammar symbol. The corresponding AST (*abstract syntax tree*, Fig. 4c) is easy to construct and can be used directly for the generation of target code. A tree traversal based on post-order can be used for code generation, with code being generated for the children of a node before dealing with the operation specified in the node. This appears to be a very attractive approach to code generation, but there is a clear disadvantage in that it is extremely difficult to generate high-quality code from this representation. While the AST is more concise than a syntax tree, it faithfully retains the structure of the original source code. For example, the previous AST contains two distinct copies of the expression `a × 2`.

A DAG (*directed acyclic graph*, Fig. 4d) is a contraction of the AST that avoids this duplication. In a DAG, nodes can have multiple parents, and identical subtrees are reused. Such sharing makes the DAG more compact than the corresponding AST. In real systems, DAGs are used for two reasons. If memory constraints limit the size of programs that the compiler can handle, using a DAG can help by reducing the memory footprint. Other systems use DAGs to expose redundancies. Here, the benefit lies in better compiled code. These latter systems tend to use the DAG as a derivative IR —building the DAG, transforming the definitive IR to reflect the redundancies, and discarding the DAG.



(a) Expression Grammar (b) Syntax Tree (c) AST (d) DAG

Figure 4. Syntax Tree, AST and DAG for a × 2 + a × 2 × b using the classic Expression Grammar

- *Hybrid IRs* combine elements of both graphical and linear IRs, in an attempt to capture their strengths and avoid their weaknesses. A common hybrid representation uses a low-level linear IR to represent blocks of straight-line code and a graph to represent the flow of control among those blocks. Many further IRs have been developed in recent years, some being designed for general-purpose optimization and others being focused on specific types of optimization.

9. A TAXONOMY OF INTERPRETERS

An interpreter simulates an idealized computer in software. Such "computers" have a processor, code memory, data memory, and (usually) a stack. The processor pulls instructions from the code memory, decodes them, and executes them, performing the actions prescribed for those instructions by the semantics of the language. An instruction can read or write to the data memory or onto the stack. Function calls save return addresses so they can return to the instruction following the function call. Note that unlike compilation, interpretation requires the presence of the input data needed for the program. There are three things to consider when building an interpreter: how to store data, how and when to track symbols, and how to execute instructions.

Interpreters come in two varieties: iterative and recursive. An iterative interpreter works on a linearized version of the AST and requires more preprocessing than a recursive interpreter, which works directly on the AST.

Iterative interpretation is suitable when the source language's instructions are all primitive. The instructions of the source program are fetched, analyzed, and executed, one after another. Iterative interpretation is suitable for real and abstract machine codes, for some very simple programming languages, and for command languages.

Recursive interpretation is necessary if the source language has composite instructions. (In this context, 'instructions' could be statements, expressions, and/or declarations.) Interpretation of an instruction may trigger interpretation of its component instructions. An interpreter for a high-level programming language or query language must be recursive. However, recursive interpretation is slower and more complex than iterative interpretation, so we usually prefer to compile high-level languages, or at least translate them to lower-level intermediate languages that are suitable for iterative interpretation.

9.1. Iterative interpreters

The structure of an *iterative interpreter* is much closer to that of a CPU than that of a recursive interpreter. It consists of a flat loop over a case statement which contains a code segment for each node type; the code segment for a given node type implements the semantics of that node type, as described in the language definition manual. It requires a fully annotated and threaded AST, and maintains an *active-node pointer*, which points to the node to be interpreted, the *active node*. The iterative interpreter runs the code segment for the node pointed at by the active-node pointer; at the end, this code sets the active-node pointer to another node, its successor, thus leading the interpreter to that node, the code of which is then run, etc. The active-node pointer is comparable to the instruction pointer in a CPU, except that it is set explicitly rather than incremented implicitly.

The data structures inside an iterative interpreter resemble much more those inside a compiled program than those inside a recursive interpreter. There will be an array holding the global data of the source program, if the source language allows these. If the source language is stack-oriented, the iterative interpreter will maintain a stack, on which local variables are allocated. Variables and other entities have addresses, which are offsets in these memory arrays. Stacking and scope information, if applicable, is placed on the stack. The symbol table is not used, except perhaps to give better error messages. The stack can be conveniently implemented as an extensible array.

Conventional interpreters are iterative: they work in a fetch-analyze-execute cycle. This is captured by the following *iterative interpretation scheme*:

```
initialize
do {
    fetch the next instruction
    analyze this instruction
    execute this instruction
} while (still running);
```

First, an instruction is fetched from storage, or in some cases entered directly by the user. Second, the instruction is analyzed into its component parts. Third, the instruction is executed. The whole cycle is then repeated. Typically the source language has several forms of instruction, so execution of an instruction decomposes into several cases, one case for each form of instruction.

Iterative interpreters are usually somewhat easier to construct than recursive interpreters; they are much faster but yield less extensive run-time diagnostics. Iterative interpreters are much easier to construct than compilers and in general allow far superior run-time diagnostics. Executing a program using an interpreter is, however, much slower than running the compiled version of that program on a real machine. Using an iterative interpreter can be expected to be between 100 and 1000 times slower than running a compiled program, but an interpreter optimized for speed can reduce the loss to perhaps a factor of 30 or even less, compared to a program compiled with an optimizing compiler. Advantages of interpretation unrelated to speed are increased portability and increased security, although these properties may also be achieved in compiled programs. An iterative interpreter is the best means to run programs for which extensive diagnostics are desired or for which no suitable compiler is available.

In the following subsections we apply this scheme to the interpretation of machine code, and the interpretation of simple command and programming languages.

9.1.1. Iterative interpretation of machine code

An interpreter of machine code is often called an *emulator*. It is worth recalling here that a real machine *M* is functionally equivalent to an emulator of *M*'s machine code. The only difference is that a real machine uses electronic (and perhaps parallel) hardware to fetch, analyze, and execute instructions, and is therefore much faster than an emulator.

A machine-code instruction is essentially a record, consisting of an operation field (usually called the *op-code*) and some operand fields. Instruction analysis (or *decoding*) is simply unpacking these fields. Instruction execution is controlled by the op-code.

To implement an emulator, we employ the following simple techniques:

- Represent the machine's storage by an array. If storage is partitioned, for example into separate stores for code and data, then represent each store by a separate array.
- Represent the machine's registers by variables. This applies equally to visible and hidden registers. One register, the *code pointer* or *program counter*, will contain the address of the next instruction to be executed. Another, the *status register*, will be used to control program termination.
- Fetch each instruction from the (code) store.
- Analyze each instruction by isolating its op-code and operand field(s).
- Execute each instruction by means of a switch-statement, with one case for each possible value of the op-code. In each case, emulate the instruction by updating storage and/or registers.

When we write an interpreter of machine code, it makes no difference whether we are interpreting a real machine code or an abstract machine code. For an abstract machine code, the interpreter will be the only implementation. For a real machine code, a hardware interpreter (*processor*) will be available as well as a software interpreter (*emulator*). Of these, the processor will be much the faster. But an emulator is much more flexible than a processor: it can be adapted cheaply for a variety of purposes. An emulator can be used for experimentation before the processor is ever constructed. An emulator can also easily be extended for diagnostic purposes. So, even when a processor is available, an emulator for the same machine code complements it nicely.

An *interpretive compiler* is a combination of compiler and interpreter, giving some of the advantages of each. The key idea is to translate the source program into an *intermediate language*, designed to the following requirements:

- it is intermediate in level between the source language and ordinary machine code;
- its instructions have simple formats, and therefore can be analyzed easily and quickly;
- translation from the source language into the intermediate language is easy and fast.

Thus an interpretive compiler combines fast compilation with tolerable running speed.

Example 3. Interpretive compilation

Oracle's Java Development Kit (JDK) is an implementation of an interpretive compiler for Java. At its heart is the Java Virtual Machine (JVM), a powerful abstract machine.

JVM-code is an intermediate language oriented to Java. It provides powerful instructions that correspond directly to Java operations such as object creation, method call, and array indexing. Thus translation from Java into JVM-code is easy and fast. Although powerful, JVM-code instructions have simple formats like machine-code instructions, with operation fields and operand fields, and so are easy to analyze. Thus JVM-code interpretation is relatively fast: 'only' about ten times slower than machine code.

JDK consists of a Java-into-JVM-code translator and a JVM-code interpreter, both of which run on some machine M.

Interpretive compilers are very useful language processors. In the early stages of program development, the programmer might well spend more time compiling than running the program, since he or she is repeatedly discovering and correcting simple syntactic, contextual, and logical errors. At that stage fast compilation is more important than fast running, so an interpretive compiler is ideal. (Later, and especially when the program is put into production use, the program will be run many times but rarely recompiled). At that stage fast running will assume paramount importance, so a compiler that generates efficient machine code will be required. In Java, this problem is typically addressed by a so-called *just-in-time compiler*.

9.1.2. Iterative interpretation of simple command and programming languages

Command languages (like some graphics languages, network protocols, text-processing languages, job control languages, and shell scripting languages, such as the UNIX *shell* language) are relatively simple languages. In normal usage, the user enters a sequence of commands, and expects an immediate response to each command. Each command will be executed just once. These factors suggest interpretation of each command as soon as it is entered. In fact, command languages are specifically designed to be interpreted.

Iterative interpretation is also possible for certain programming languages, provided that a source program is just a sequence of primitive commands. The programming language must not include any composite commands, i.e., commands that contain subcommands.

Iterative interpretation is carried out by a *syntax-directed interpreter* which mimics what we do when we trace source code manually. As we step through the code, we parse, validate, and execute instructions. Everything happens in the parser because a syntax-directed interpreter doesn't create an AST or translate the source code to bytecodes or machine code. The interpreter directly feeds off of syntax to execute statements.

Example 4. Interpreter for Mini-Basic

Consider a simple programming language with the following syntax (expressed in EBNF):

```
Program            ::= Command*
Command            ::= Variable = Expression |
                       read Variable |
                       write Variable |
                       go Label |
                       if Expression Relational-Op Expression go Label |
                       stop
Expression         ::= primary-Expression |
                       Expression Arithmetic-Op primary-Expression
primary-Expression ::= Numeral | Variable | ( Expression )
Arithmetic-Op      ::= + | - | * | /
Relational-Op      ::= = | <> | < | <= | >= | >
Variable           ::= a | b | c | ... | z
Label              ::= Digit Digit*
```

In the iterative interpretation scheme, the 'instructions' are taken to be the commands of the programming language. Analysis of a command consists of syntactic and perhaps contextual analysis. This makes analysis far slower and more complex than decoding a machine-code instruction. Execution is controlled by the form of command, as determined by syntactic analysis.

A Mini-Basic program is just a sequence of commands. The commands are implicitly labeled `0`, `1`, `2`, etc., and these labels may be referenced in `go` and `if` commands. The program may use up to twenty-six variables, which are predeclared.

The semantics of Mini-Basic programs should be intuitively clear. All values are real numbers. The program shown in Figure 5 reads a number (into variable `a`), computes its square root accurate to two decimal places (in variable `b`), and writes the square root.

It is easy to imagine a *Mini-Basic abstract machine*. The Mini-Basic program is loaded into a code store, with successive commands at addresses `0`, `1`, `2`, etc. The code pointer, CP, contains the address of the command due to be executed next.

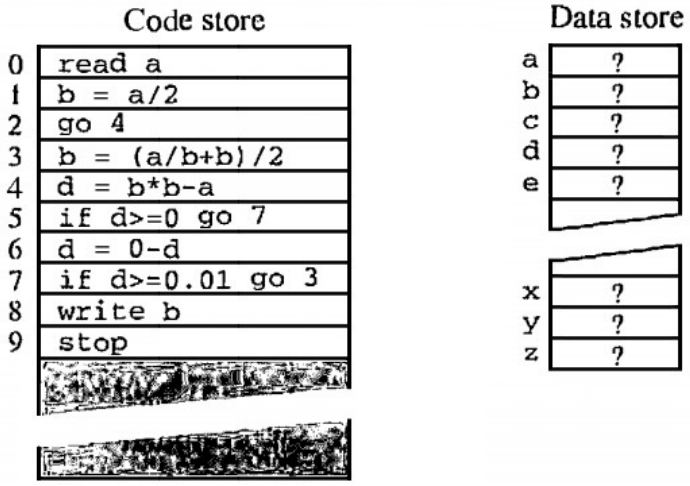


Figure 5. Mini-Basic abstract machine: code store and data store

The program's data are held in a data store of 26 cells, one cell for each variable. Figure 5 illustrates the code store and data store. We must decide how to represent Mini-Basic commands in the code store. The choices, and their consequences, are as follows:

(a) *Source text*: Each command must be scanned and parsed at run-time (i.e., every time the command is fetched from the code store).

(b) *Token sequence*: Each command must be scanned at load-time, and parsed at runtime.

(c) *AST*: All commands must be scanned and parsed at load-time.

Choice (a), illustrated in Figure 5, would slow the interpreter drastically. Choice (c) is better but would slow the loader somewhat. Choice (b) is a reasonable compromise.

Although a sequence of tokens is a convenient way of representing a command in storage, it would be inconvenient for execution. Analysis of a scanned command should parse it and translate it to an internal form suitable for execution. In Mini-Basic all variables are predeclared, so there is no need to check for undeclared variables; and there is only one data type, so there is no need for type checking.

For particular forms of commands and expressions, the functions/methods `eval` and `apply` (the so-called *interpreting methods*) must be implemented. Access to the state of the Mini-Basic abstract machine must be allowed to both interpreting methods.

Since an interpreter is easier to implement than a compiler, many programming languages rely on an interpreter for their first implementation. The original Basic was one of the few 'high-level' programming languages for which interpretation was normal. A typical Basic language processor allowed programs to be entered, edited, and executed incrementally. Such a language processor could run on a microcomputer with very limited storage, hence its popularity in the early days of microcomputers. But this was possible only because the language was very primitive indeed. Its control structures were more typical of a low-level language, making it unattractive for serious programmers. More recently, 'structured' dialects of Basic have become more popular, and compilation has become an alternative to interpretation.

9.2. Recursive interpreters

Modern programming languages are higher-level than a simple programming language. In particular, commands may be composite: they may contain subcommands, subsubcommands, and so on.

It is possible to interpret higher-level programming languages. However, the iterative interpretation scheme is inadequate for such languages. Analysis of each command in the source program entails analysis of its subcommands, recursively. Likewise, execution of each command entails execution of its subcommands, recursively. Thus we are driven inexorably to a two-stage process, whereby the entire source program is analyzed before interpretation proper can begin. This gives rise to the *recursive interpretation scheme*:

```
fetch and analyze the program
execute the program
```

where both analysis and execution are recursive.

We must decide how the program will be represented at each stage. If it is supplied in source form, 'fetch and analyze the program' must perform syntactic and contextual analysis of the program. A decorated AST is therefore a suitable representation for the result of the analysis stage. Therefore 'execute the program' will operate on the program's decorated AST.

Recursive interpretation is carried out by a *tree-based interpreter* which builds a complete scope tree before executing a program. That means it can support both statically typed languages like Java and dynamically typed languages like Python. (It can resolve all symbols statically before execution.). A tree-based interpreter is like a compiler front-end with an interpreter grafted onto the back-end instead of a code generator.

A *recursive interpreter* has an interpreting routine for each node type in the AST. Such an interpreting routine calls other similar routines, depending on its children; it essentially does what it says in the language definition manual. This architecture is possible because the meaning of a language construct is defined as a function of the meanings of its components. For example, the meaning of an if-statement is defined by the meanings of the condition, the then-part, and the else-part it contains, plus a short paragraph in the manual that ties them together. This structure is reflected faithfully in a recursive interpreter, which first interprets the condition and then, depending on the outcome, interprets the then-part or the else-part; since the then and else-parts can again contain if-statements, the interpreter routine for if-statements will be recursive, as will many other interpreter routines. The interpretation of the entire program starts by calling the main interpretation routine with the top node of the AST as a parameter.

Another important feature is the *status indicator*; it is used to direct the flow of control. Its primary component is the mode of operation of the interpreter. This is an enumeration value; its normal value is something like *NormalMode*, indicating sequential flow of control, but other values are available, to indicate jumps, exceptions, function returns, and possibly other forms of flow of control. Its second component is a value in the wider sense of the word, to supply more information about the non-sequential flow of control. This may be a value for the mode *ReturnMode*, an exception name plus possible values for the mode *ExceptionMode*, and a label for *JumpMode*. The status indicator should also contain the file name and the line number of the text in which the status indicator was created, and possibly other debugging information.

Each interpreting routine checks the status indicator after each call to another routine, to see how to carry on. If the status indicator is *NormalMode*, the routine carries on normally. Otherwise, it checks to see if the mode is one that it should handle; if it is, it does so, but if it is not, the routine returns immediately, to let one of the parent routines handle the mode.

Variables, named constants, and other named entities are handled by entering them into the symbol table, in the way they are described in the manual.

The typical recursive interpreter analyzes the entire source program before execution commences. Thus it forgoes one of the usual advantages of interpretation, that is immediacy. This explains why recursive interpretation is not generally used for higher-level programming languages. For such languages, a better alternative is compilation of source programs to a simple intermediate language, followed by iterative interpretation of the intermediate language. Two notable exceptions to the general rule are Lisp and Prolog.

Recursive interpretation is less common than iterative interpretation. However, this form of interpretation has long been associated with Lisp. A Lisp program is not just represented by a tree: it is a tree! Several features of the language —dynamic binding, dynamic typing, and the possibility of

manufacturing extra program code at run-time— make interpretation of Lisp much more suitable than compilation. Lisp has always had a devoted band of followers, but not all are prepared to tolerate slow execution. A more recent successful dialect, Scheme, has discarded Lisp's problematic features in order to make compilation feasible.

It is noteworthy that two popular programming languages, Basic and Lisp, both suitable for interpretation but otherwise utterly different, have evolved along somewhat parallel lines, spawning structured dialects suitable for compilation! Another example of a high-level language suitable for interpretation is Prolog. This language has a very simple syntax, a program being a flat collection of clauses, and it has no scope rules and few type rules to worry about. Interpretation is almost forced by the ability of a program to modify itself by adding and deleting clauses at run-time.

A recursive interpreter can be written relatively quickly, and is useful for rapid prototyping; it is not the architecture of choice for a heavy-duty interpreter. A secondary but important advantage is that it can help the language designer to debug the design of the language and its description. Disadvantages are the speed of execution, which may be a factor of 1000 or more lower than what could be achieved with a compiler, and the lack of static context checking: code that is not executed will not be tested. Speed can be improved by doing judicious memoizing: if it is known, for example, from the identification rules of the language that an identifier in a given expression will always be identified with the same type (which is true in almost all languages) then the type of an identifier can be memoized in its node in the syntax tree. If needed, full static context checking can be achieved by doing full attribute evaluation before starting the interpretation; the results can also generally be used to speed up the interpretation.

Recursive interpreters that operate on high-level programs with little to no preprocessing before execution are best suited to implementing DSLs rather than general-purpose programming languages. They are the fastest path to getting a language up and running, they are not too hard to build and are very flexible. (We can add new instructions without much trouble.). A DSL is just that: a (typically small) language designed to make users particularly productive in a specific domain, i.e. a language designed for a narrow class of problems. Examples are Mathematica, shell scripts, wikis, UML, XSLT, makefiles, formal grammars, database query languages, text-formatting languages, scene description languages for ray-tracers, languages for setting up economic simulations, and even data file formats like comma-separated values and XML. The opposite of a DSL is a general-purpose programming language like C, Java, or Python. In the common usage, DSLs also typically have the connotation of being smaller because of their focus. This isn't always the case, though. SQL, for example, is a lot bigger than most general-purpose programming languages. The target language for a compiler for a DSL may be traditional machine code, but it can also be another high-level language for which compilers already exist, a sequence of control signals for a machine, or formatted text and graphics in some printer-control language (e.g., PostScript), and DSLs are often interpreted instead of compiled. Even so, all DSL

compilers and interpreters will have front-ends for reading and analyzing the program text that are similar to those used in compilers and interpreters for general-purpose languages.

The only drawback of recursive interpreters is run-time efficiency: they are not particularly efficient at run-time. If we really care about efficiency for a particular DSL or need to implement a more general programming language, those interpreters aren't appropriate. To squeeze memory resources down and to accelerate execution speed, we need to process the input source code more. In order to do that, another category of interpreters that is much more efficient is needed. Unfortunately, the efficiency comes at the cost of a more complicated implementation. In essence, we need a tool that translates the high-level source code down into low-level instructions called *bytecode instructions* (instructions whose operation code fits in an 8-bit byte). Then, we can execute those bytecodes on an efficient interpreter called a *virtual machine* (VM). These machines are to be distinguished from the virtual machines that run one operating system inside another. Most of the currently popular languages are VM-based (Java, JavaScript, C#, Python, Ruby 1.9).

If we're translating source code to low-level bytecodes, you might ask why we don't compile all the way down to machine code and skip the interpreter altogether. Raw machine code would be even more efficient. Bytecode interpreters have a number of useful characteristics including portability. (Machine code is specific to a CPU, whereas bytecodes run on any computer with a compatible interpreter.) But, the biggest reason we avoid generating machine code is that it's pretty hard.

CPU instructions have to be very simple so that we can implement them easily and efficiently in hardware. The result is often an instruction set that is quirky, irregular, and far removed from high-level source code. Bytecode interpreters, on the other hand, are specifically designed to be easy to *target* (generate code for). At the same time, instructions have to be low-level enough that we can interpret them quickly.

10. ON LEARNING ABOUT COMPILERS AND INTERPRETERS

Why Learn About Compilers and Interpreters? Few people will ever be required to write a compiler or an interpreter for a general-purpose language like C, Java o Python. So why do most computer science institutions offer courses on this topic and often make these mandatory? Some typical reasons are:

(a) It is considered a topic that you should know in order to be "well-cultured" in computer science.
(b) A good craftsman should know his tools, and compilers and interpreters are important tools for programmers and computer scientists.
(c) The techniques used for constructing a compiler or interpreter are useful for other purposes as well.
(d) There is a good chance that a programmer or computer scientist will need to write a compiler or an interpreter for a domain-specific language.

The first of these reasons is somewhat dubious, though something can be said for "knowing your roots", even in such a hastily changing field as computer science.

Reason "b" is more convincing: Compilers and interpreters are complex pieces of code and an awareness of how they work can very helpful. Understanding how a high-level language program is translated into a form that can be executed by the hardware gives a good insight into how a program will behave when it runs, where the performance bottlenecks will be, and into the costs of executing individual high-level language statements, i.e. it will allow programmers to tune their high-level programs for better efficiency. Furthermore, the error reports that compilers and interpreters provide are often easier to understand when one knows about and understands the different phases of compilation and interpretation, such as knowing the difference between lexical errors, syntax errors, type errors, and so on.

The third reason is also quite valid. The algorithms used in a compiler or interpreter are relevant to many other application areas such as aspects of text decoding and analysis and the development of command-driven interfaces. In particular, the techniques used for reading (lexing and parsing) the text of a program and converting this into a form (abstract syntax) that is easily manipulated by a computer, can be used to read and manipulate any kind of structured text such as XML documents, address lists, etc. For example, transforming data from one syntactic form into another can be approached by considering the grammar of the structure of the source data and using traditional parsing techniques to read this data and then output it in the form of the new grammar. Furthermore, it may be appropriate to develop a simple language to act as a user interface to a program. The simplicity and elegance of some parsing algorithms and basing the parsing on a formally specified grammar helps produce uncomplicated and reliable software tools.

Building a compiler or an interpreter gives a practical application area for many fundamental data structures and algorithms, it allows the construction of a large-scale and inherently modular piece of software, ideally suited for construction by a team of programmers. It gives an insight into programming languages, helping to show why some programming languages are the way they are, making it easier to learn new languages. Indeed, one of the best ways of learning a programming language is to write a compiler or an interpreter for that language, preferably writing it in its own language. Writing a compiler also gives some insight into the design of target machines, both real and virtual. It is worth noting that the methods needed to make the front-end of a compiler or interpreter are more widely applicable than the methods needed to make the back-end, but the latter is more important for understanding how a program is executed on a machine.

The more language applications you build, the more patterns you'll see. The truth is that the architecture of most language applications is freakishly similar. A broken record will play in your head every time you start a new language application: "First build a syntax recognizer that creates a data

structure in memory. Then sniff the data structure, collecting information or altering the structure. Finally, build a report or code generator that feeds off the data structure." You even start seeing patterns within the tasks themselves. Tasks share lots of common algorithms and data structures.

Once you get these language implementation design patterns and the general architecture into your head, you can build pretty much whatever you want. Building a new language doesn't require a great deal of theoretical computer science. You might be skeptical because every book you've picked up on language development has focused on compilers. Yes, building a compiler for a general-purpose programming language requires a strong computer science background. But, most of us don't build compilers. Some things that we build all the time are: configuration file readers, data readers, model-driven code generators, source-to-source translators, source analyzers, and interpreters.

When we talk about language applications, we're not just talking about *implementing* languages with a compiler or interpreter. We're talking about any program that processes, analyzes, or translates an input file. Implementing a language means building an application that executes or performs tasks according to sentences in that language. That's just one of the things we can do for a given language definition. For example, from the definition of C, we can build a C compiler, a translator from C to Java, or a tool that instruments C code to isolate memory leaks. Similarly, think about all the tools built into the Eclipse development environment for Java. Beyond the compiler, Eclipse can refactor, reformat, search, syntax highlight, and so on.

In order to build an interpreter for a general-purpose programming language, you will need garbage collection, executing code from more than one source file (linking), classes, libraries, and debuggers. The best way to figure out how all that works is to dig into the source for an existing interpreter. Interpreter source code is available for just about every common language, both dynamically typed and statically typed. You can also read the literature on the Smalltalk and Self interpreters as well as garbage collection and dynamic method dispatch. A good paper to read on register-based bytecode machines is *The Implementation of Lua 5.0*. One of the things you'll discover quickly when reading source code or building your own interpreter is that making an interpreter go really fast is not that easy. Every CPU clock cycle and memory access counts. That's why the core of most fast interpreters is some hand-tuned assembly code. Knowledge of computer architecture, such as cache memory and CPU pipelines, is essential. In January 2009, Dan Bornstein, the guy who designed the Dalvik VM (a register-based VM that runs Java programs on Google's Android mobile platform, though with a different bytecode) described the amazing gymnastics he and the VM team went through to squeeze every last drop of efficiency out of the phone's ARM CPU, and flash memory. (Flash memory is much slower than dynamic memory but doesn't get amnesia when you lose power.) Aside from being slow, there isn't much flash memory on the phone device. The Dalvik VM tries to share data structures and compress them whenever possible. On average, the Dalvik VM uses a bit more code memory than the

Java VM's stack-based instruction set for a given method. But, as Bornstein explains, the trade-off is totally worth it. The Dalvik VM executes many fewer instructions to achieve the same result. (It can reuse registers to avoid unnecessary operand stack pushes and pops.) Because of the overhead to execute each instruction, fewer instructions means much less overhead and higher performance. You can learn more about the Dalvik VM implementation details by looking at the Dalvik VM source code. (You can also learn some of the underlying principles by doing a web search for *threaded interpreter*.)

Reason "d" is becoming more and more important as domain-specific languages (DSLs) are gaining in popularity. The need for DSLs occurs frequently and the knowledge of compiler and interpreter design can facilitate their rapid implementation.

In brief, writing a compiler or interpreter is an excellent educational project and enhances skills in programming language understanding and design, data structure and algorithm design and a wide range of programming techniques. Studying compilers and interpreters makes you a better programmer.

Bibliography

Abelson, H., Sussman, G. J. y Sussman, J. (1996). *Structure and interpretation of computer programs* (2ª Ed.). MIT Press.

Ball, T. (2017). *Writing an interpreter in Go*. Edición del autor.

Cooper, K. D. y Torczon, L. (2012). *Engineering a compiler* (2ª Ed.). Elsevier/Morgan Kaufmann.

Friedman, D. P. y Wand, M. (2008). *Essentials of programming languages* (3ª Ed.). MIT Press.

Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C. y Langendoen, K. (2012). *Modern compiler design* (2ª Ed.). Springer.

Mak, R. (2009). *Writing compilers and interpreters: A modern software engineering approach using Java* (3ra Ed.). Wiley.

Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.

Parr, T. (2010). *Language implementation patterns: Create your own domain-specific and general programming languages*. Pragmatic Bookshelf.

Watson, D. (2017). *A practical approach to compiler construction*. Springer.

Watt, D. A. y Brown, D. F. (2000). *Programming language processors in Java: Compilers and interpreters*. Pearson Education Ltd.