1 Building a programming language

This chapter covers

- Why build a new programming language
- Kinds of programming languages
- Translators of programming languages
- Steps to develop translators

Have you ever wished for a programming language that's better than the one you're currently using? Like many programmers working on an especially difficult program, perhaps you believe that your current language ought to be easier to use. Does it lack constructs that would simplify the types of problems you often must solve? Too verbose? Prone to syntax and logic errors? Programs hard to test and maintain? If you're a programming instructor, is the language too difficult to teach or to learn?

You may be thinking, "Wouldn't it be cool if I could design an ideal language, develop a compiler for it, and then write, compile, and execute programs written in my new language?"

This book will show you how to do all that and more.

We'll write the compiler, interpreter, and other tools in Java, so you should be comfortable with that language. But this book assumes no prior knowledge of programming language design or compilers or interpreters.

1.1 What does it mean to build a programming language?

To build a programming language, you must first design the language's *syntax* (how do you correctly write statements and expressions in the language) and the language's *semantics* (what do those statements and expressions mean). This can be a brand new language or improvements to an existing language. Then you must implement your language to make it usable — you want to be able to write and execute programs written in the language. You

implement the language by developing one or more translators for it. In this book, the translators you'll learn to develop are

- A program converter which converts programs written in your language to equivalent programs written in an existing high-level language.
- An interpreter which executes programs written in your language. The interpreter will include a debugger that allows you to interact with a program while it's running.
- A compiler which translates your programs into low-level machine code that can execute at the machine's full speed.

Figure 1.1 illustrates these points.

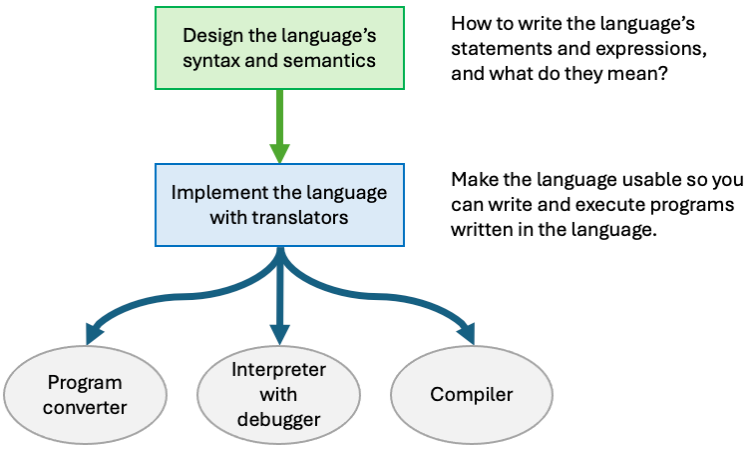

Figure 1.1: What it means to build a programming language.

Syntax and semantics are critical concepts to understand when developing translators. We'll discuss them in much greater detail in the following chapters, but here's a simple clarifying example. Consider this assignment statement in the Pascal programming language:

```
alpha := beta + gamma*delta
```

According to Pascal's syntax rules, this statement is syntactically correct. That's the correct way to write an assignment statement that has an arithmetic expression.

But what does the statement mean? This is where Pascal's semantic rules come in: `beta`, `gamma`, and `delta` are variables whose current values will be used to calculate the value of the expression to the right of the assignment operator `:=`. The operators `+` means to add and `*` means to multiply. Furthermore, `*` has precedence over `+`, and so the multiplication will take place before the addition. The value of variable `alpha` to the left of the assignment operator `:=` will be set to the calculated value of the expression.

In fewer words, syntax is about *how to write* the statement correctly, and semantics is about *what it means to execute* the statement to produce the correct results.

1.2 Why should you build a new programming language?

It is not a trivial task to build a new programming language, either a new one you invent from scratch or one you create by adding features to improve an existing language. What is the purpose of the language? What types of applications do you plan to write in it? What features should it have to support those applications? Would it become easier to write the kinds of programs that you write?

"Why isn't there a better language?" is a question asked by programmers who subsequently individually, or as a team, built new and better programming languages. They invented languages like Java, Python, SQL, Rust, Go, Kotlin, Swift, Lua, and many others. By following this book's careful step-by-step approach, *you can join the elite group of new language builders*.

Compilers are complicated software that require strong programming abilities to develop. This book will teach solid software architecture and organization methods, along with the necessary object-oriented programming techniques, advanced data structures and algorithms, design patterns, and more. Successfully developing a compiler will be convincing evidence that you have acquired these skills. You'll be able to apply the skills to any other large-scale software projects at work or school. Beyond a highly impressive technical achievement, successfully developing a compiler is an incredibly rewarding software engineering challenge.

Bottom line: You will become a better, more confident, programmer for any software project.

1.3 Categories of programming languages

We can take a big picture look at programming languages and group them into two broad categories: general-purpose languages and *domain-specific languages* (DSLs). Table 1.1 summarizes both categories.

Table 1.1 Programming Language Categories

Category	Examples	Purpose
General-purpose language	Java, Python, C++, C#	These languages have generic statements and expressions for writing programs in a wide variety of application domains. They often come with specialized packages and libraries to support specific types of applications.

Domain-specific language (DSL)	SQL for database applications, HTML + CSS for web applications, HCL for infrastructure configuration, VHDL to simulate the behavior of digital systems	These languages have features tailored for writing programs in a particular application domain. They have built-in statements and expressions to implement operations intrinsic to the domain, such as SQL's `SELECT` statement to search and join database tables.

Building a new general-purpose or a domain-specific language requires the same software engineering skills.

1.3.1 General-purpose languages

General-purpose languages such as Java, Python, and C++ have generic statements and expressions that enable writing a wide variety of programs. Programmers who use these languages to write certain types of applications can use the languages' libraries of classes that implement data structures and algorithms tailored for those applications. Two example libraries are a collection of mathematical functions and a framework for GUI-based programming.

You can design a new general-purpose programming language that's easier to learn or teach. It can have better control structures, such as a generalized `if` statement that combines Java's `if-then-else` and `switch` statements:

```
if
{
    i > j    : k = i - j;
    i < j    : k = j - i;
    x/y > 0  : k = i + j;
    else     : k = 0;
}
```

Or perhaps you want your language to have direct support for matrix operations, such as required for a least-squares solution to a linear regression problem:

```
matrix A[20, 2];  // data matrix with 20 rows and 2 columns
vector x[2];      // vector of 2 unknown regression coefficients
vector b[20];     // vector of 20 observed values

x = (A`t * A)`i * A`t * b;  // expression with matrix operators
```

where operation `*` multiplies two matrices, operation `` `t `` transposes a matrix, and `` `i `` inverts a matrix.

You can build a brand new general-purpose language from scratch, or you can add new features to an existing language.

1.3.2 Domain-specific languages

Domain-specific languages (DSLs) are especially important for companies that develop software for particular markets. A company's management wants its programmers to develop the company's software products in the most productive, reliable, and fastest way possible.

In today's competitive economy, programming language design is even more relevant. Companies rely on DSLs such as:

- workflow languages
- query and searching languages
- infrastructure languages
- configuration languages
- data pipeline messages

Even companies that rely on AI-generated code will want that code to be written in a language that can be most effectively generated and tested.

A DSL that supports a company's operations is a valuable asset.

If you design a new DSL, it should have features that support operations in that domain and thereby make it easier to write programs. For example, a DSL for database programming might have this construct to join two database tables to create a display table:

```
join tables employee, table salary
    display table "pay" by matching column id
        use headers "ID", "name", "payment"
        columns employee.id, employee.name, salary.hours*salary.rate
```

A DSL for web programming might have this construct to scrape an invoice number off of a webpage:

```
on page "Orders"
    label   invoice_label  = find "Invoice"
    textbox invoice_box    = locate right of invoice_label
    string  invoice_number = text from invoice_box
```

Most likely, you'll build a new DSL from scratch.

1.4 The journey from language design to program execution

Deciding what features to have in a programming language is just the first step of building the language. The language must enable you to write programs that will actually run. That can mean developing a compiler that will compile programs written in your new language to executable machine code.

Developing a compiler is certainly a daunting task for any programmer. But compilers are not the only way to get a program into a runnable state. We'll get to developing a compiler in this book not directly, but instead we'll take some practical intermediate steps along our journey. We'll first develop a *program converter* that converts programs written in one source language to another language, and then an *interpreter* that executes a source program and has an integrated interactive debugger. As shown in figure 1.2 below, at each step of the journey, you'll acquire skills that you'll apply at the next step.

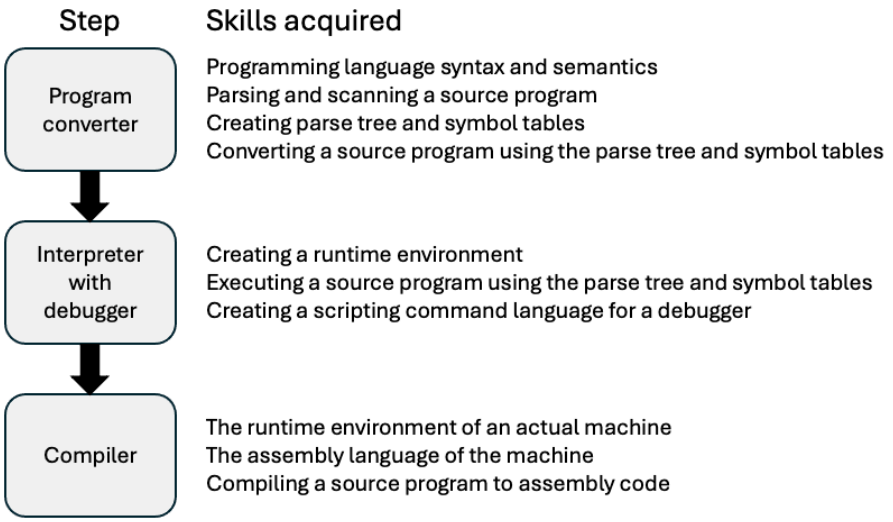

Figure 1.2: The steps and the skills acquired on the journey to developing a compiler.

1.5 It's all about translation!

Our first step is to develop a program converter, which translates a program written in one language to an equivalent program written in another language. This is the simplest translator and developing it will give you the skills for our next step, an *interpreter*. An interpreter translates a program into runtime actions. Developing an interpreter teaches you in great detail how to execute statements and expressions in a program. Then finally at our final step, you'll have acquired the skills to build a compiler.

The table below summarizes the outputs of each type of translator. Program converters and interpreters are extremely useful tools in their own right.

Table 1.2: Translators for Programming Languages

Type of translator	Translator output
Program converter	The output is the result of converting a source program (one written in the source language) to an equivalent program written in the target language, such as converting from a legacy or newly invented language to Java.
Interpreter	An interpreter directly executes a source program code under its full control. Its output is whatever the program writes out during its run time.
Compiler	A compiler translates a source program to low-level executable machine code. The output of compiling a source program is the equivalent program in machine code that can run at full machine speed.

For each translator, we'll use a subset of the Pascal language as our example source language. Although you're encouraged to design your own language, Pascal is a good base for several important reasons:

- Pascal was designed to be a simple, clean, and consistent language. It's an excellent source language for developing programming language translators. We'll be able to concentrate on translation issues without becoming entangled in a messy language syntax.
- Pascal is a real programming language. It was the teaching language of the 1980s. Nevertheless, it has many features that are in current programming languages.
- There are online Pascal compilers that you can use to compare their results to the results of the translators that you write. One example is at https://www.jdoodle.com/execute-pascal-online.

You do not need prior knowledge of Pascal to understand the examples in this book.

1.5.1 Program converter

A *program converter* translates a program written in the source language to the target language. This is the simplest translator of the three that we'll develop in this book. Developing it will give you the necessary skills to develop the next translator, an interpreter.

A program converter is itself an extremely useful tool. It is the fastest way to compile a program written in a newly invented source language, since you take advantage of the existing compiler of the target language.

If a company has programs that continue to be useful but are no longer maintainable because they're written in a legacy or proprietary language, you can use a program converter to translate them into a modern industry-standard language. Here's an example of a function written in legacy Pascal:

Listing 1.1 A function written in legacy Pascal

```
FUNCTION root(x : real) : real;
```

```
VAR
    r : real;
    diff, prev : real;

BEGIN
    r := 1;
    prev := 0;

    REPEAT
        r := (x/r + r)/2;
        diff := abs(prev - r);
        prev := r;
    UNTIL diff < 1.0e-8;

    root := r    #A
END;
```

#A Set the return value of the function `root`.

Here is the function translated to an equivalent modern Java function:

Listing 1.2 The equivalent function written in modern Java

```
static float root(float x)
{
    float r;
    float diff, prev;

    r = 1;
    prev = 0;

    do
    {
        r = (x/r + r)/2;
        diff = Math.abs(prev - r);
        prev = r;
    } while (diff >= 1.0e-8);

    return r;
}
```

You'll be able to translate an entire Pascal program and then run it as a Java program.

1.5.2 Interpreter

When you develop an interpreter for programs written in your new language, a major part of the interpreter is code that runs the programs under its full control. In other words, an interpreter directly translates a source program into the program's runtime actions.

Because an interpreter runs a source program, it must know how to execute each of the program's statements and expressions. Since it controls all of a source program's runtime actions, you can include a debugger that allows you to interact with the program as it's executing. With an interactive debugger, you can set breakpoints, single-step statement execution, examine and change the values of variables, and more.

Here's an example transcript of interactively debugging the previous Pascal `root()` function. The debugger refers to each statement by its line number:

Listing 3.3 An example transcript of interactively debugging the Pascal root () function

```
017 FUNCTION root(x : real) : real;
018     VAR
019         r : real;
020         diff, prev : real;
021
022     BEGIN
023         r := 1;
024         prev := 0;
025
026         REPEAT
027             r := (x/r + r)/2;
028             diff := abs(prev - r);
029             prev := r;
030         UNTIL diff < 1.0e-8;
031
032         root := r
033     END;
034
035 BEGIN
036     FOR n := 1 TO 10 DO writeln(n:4, root(n):10:4);
037 END.
```

Here's the start of the transcript. The debugger's prompts and output are in bold, and what you, the programmer, type in response is underlined. The remaining text is output from the source program itself. You can watch variable `r` each time its value is accessed and each time its value is changed.

Listing 1.3 Continued debugging of the root () function

```
At line 035
Command? break 27
Command? go
   1
At line 027
Command? step

At line 028
Command? step

At line 008
Command? go
```

```
At line 027
Command? go
    1.00000000
   2
At line 027
Command? show x
x :: 2.0
Command? show r
r :: 1.0
Command? watch r
Command? go
At line 027: r :: 1.0
At line 027: r :: 1.0
At line 027: r <= 1.5
At line 028: r :: 1.5
At line 029: r :: 1.5

At line 027
Command? unbreak 27
Command? go
At line 027: r :: 1.5                    #A
At line 027: r :: 1.5                    #A
At line 027: r <= 1.4166666666666665     #A
At line 028: r :: 1.4166666666666665     #A
At line 029: r :: 1.4166666666666665     #A
At line 027: r :: 1.4166666666666665     #A
At line 027: r :: 1.4166666666666665     #A
At line 027: r <= 1.4142156862745097     #A
At line 028: r :: 1.4142156862745097     #A
At line 029: r :: 1.4142156862745097     #A
At line 027: r :: 1.4142156862745097     #A
At line 027: r :: 1.4142156862745097     #A
At line 027: r <= 1.4142135623746899     #A
At line 028: r :: 1.4142135623746899     #A
At line 029: r :: 1.4142135623746899     #A
At line 027: r :: 1.4142135623746899     #A
At line 027: r :: 1.4142135623746899     #A
At line 027: r <= 1.414213562373095      #A
At line 028: r :: 1.414213562373095      #A
At line 029: r :: 1.414213562373095      #A
At line 032: r :: 1.414213562373095      #A
    1.41421356
```

#A Watch the value of `r` approach $\sqrt{2}$.

This is just a small demonstration, but it shows that an interpreter is extremely useful while developing and debugging a program that you're writing in your new language.

1.5.3 Compiler

The ultimate goal of this book is to write a compiler for your programming language. After you've acquired the skills to develop an interpreter, you'll be ready to concentrate on the compiler's code generator that will emit low-level machine code from a source program.

This book uses the *Java Virtual Machine* (JVM) as the target machine for the compiler. The JVM has its own machine code that it can execute. Machine code is commonly called *bytecode*. Therefore, it is very much like an actual machine, such as Intel- or Apple-based hardware. Using the JVM as our target machine is convenient because it runs on multiple platforms. Compiler skills you learn for the JVM can transfer to any actual hardware.

Our compiler won't generate the JVM's bytecode directly. You want to be able to read what code the compiler emits to see whether it's right or wrong. So instead, our compiler will emit the JVM's Jasmin *assembly code*, which is human readable. Then you'll use the Jasmin assembler that's available online to assemble the generated Jasmin code to executable bytecode.

Here's the Jasmin code that a compiler can generate from the Pascal `root()` function.

Listing 1.4 Jasmin code generated from compiling the Pascal root () function

```
.method private static root(D)D

.var 4 is diff D
.var 6 is prev D
.var 2 is r D
.var 8 is root D
.var 0 is x D
;
; 023 r:=1
;
     iconst_1
     i2d
     dstore_2
;
; 024 prev:=0
;
     iconst_0
     i2d
     dstore  6
;
; 026 REPEAT
;
L005:
;
; 027 r:=(x/r+r)/2
;
     dload_0
     dload_2
     ddiv
     dload_2
     dadd
     iconst_2
```

```
        i2d
        ddiv
        dstore_2
    ;
    ; 028 diff:=abs(prev-r)
    ;
        dload   6
        dload_2
        dsub
        invokestatic    newton5/abs(D)D
        dstore  4
    ;
    ; 029 prev:=r
    ;
        dload_2
        dstore  6
    ;
    ; UNTIL diff<1.0e-8
    ;
        dload   4
        ldc2_w  9.99999993922529E-9
        dcmpg
        iflt    L006
        goto    L005
    L006:
    ;
    ; 032 root:=r
    ;
        dload_2
        dstore  8

        dload   8
        dreturn

    .limit locals 10
    .limit stack 4
    .end method
```

You'll learn how to read and write Jasmin assembly language code.

1.5.4 Which translator to use

As shown in figure 1.2, this book will first show you how to develop a language converter, then an interpreter with a debugger, and finally a compiler, in that order to progressively acquire the necessary software engineering skills. Each translator is an extremely useful tool. The table below summarizes when it's most appropriate to use each one.

Table 1.3: When to use each translator

Type of translator	When it's appropriate to use
Program converter	A program converter is the quickest way to get a working compiler for a language, whether the programs are written in a legacy language or in your new language. You take advantage of an existing compiler to do the work.
Interpreter	An interpreter is in full control of executing a program. With an integrated debugger, you can thoroughly test and debug the program. Therefore, an interpreter is most useful while you're developing your program.
Compiler	A compiler will include a code generator that can produce the best low-level code specifically for the statements and expressions of your language. A compiled program can run orders of magnitude faster than an interpreted program. Therefore, first use an interpreter to develop and debug a program and then compile the program when you're ready to release it. Some languages are hybrid — for example, Java interprets its bytecode, but it also has a just-in-time (JIT) compiler that compiles bytecode to low-level machine code.

1.6 What you'll learn from this book

By the end of the book, not only will you have a precise understanding of how compilers and interpreters work, but you'll have successfully written them. You'll have learned how to

- Design a programming language's syntax and semantics.
- Generate a parser, scanner, parse tree, and symbol tables.
- Create a language-to-language program translator.
- Create an interpreter with an interactive debugger.
- Create a compiler that generates low-level machine code.
- Learn advanced software engineering techniques that you can apply to any large-scale software project.

"I wrote a compiler!"

I've been teaching undergraduate compiler classes at a major Silicon Valley university since 2008. At the start of each semester, my students are somewhat apprehensive. They've all certainly used compilers, such as for their Java, C++, or Python programming assignments. Compilers are mysterious, magical, and for sure complex software utility programs that they can only trust to work properly. Writing a compiler must be a daunting and frightening task to contemplate! But by the end of the semester, students can confidently claim that they've written not only a compiler, but also a program converter and an interpreter with a debugger. You can learn even the most complex material if you do it step by step, the approach we'll take in this book.

1.7 Summary

- You can build a new programming language if existing languages lack features that you need or are hard to use or teach.
    - It can be a brand new language that you invent, or new features added to an existing language.
- A programming language can be a general-purpose language or a domain-specific language (DSL).
    - A general-purpose language can be easier to learn, teach, and program for a wide variety of applications.
    - A DSL can be a valuable asset for a company.
- Program converters, interpreters, and compilers are examples of translators.
    - A converter translates source programs written in one high-level language to another.
    - An interpreter directly translates a source program to its runtime actions.
    - A compiler translates a source program to the low-level code of the target machine.
- This book teaches developing a program converter, an interpreter, and a compiler in that order so that you can use the skills you learn from developing one to develop the next one.
- Writing a translator such as a compiler is a major software engineering challenge. Successfully writing a compiler certifies that you are a top-notch programmer with the necessary skills to tackle any large-scale programming project.