WRITE YOUR OWN COMPILER

Nils M Holm

Contents

Preface

This text is the most minimal complete introduction to compiler-writing that I can imagine. It covers the entire process from analyzing source code to generating an executable file in about 100 pages of prose. The compiler discussed on the text is entirely self-contained and does not rely on any third-party software, except for an operating system.

The book covers lexical analysis, syntax analysis, and code generation by means of a minimal high-level programming language. The language is powerful enough to implement its own compiler in less than 1600 lines of readable code. The main part of the text is comprised of a tour through that code.

The language used in this book is a subset of T3X, a minimal procedural language that was first described in 1996 in the book "Lightweight Compiler Techniques". Although T3X already is quite minimal, T3X9, the dialect discussed here, is even smaller.

If you are familiar with Pascal, C, Java, or any other procedural language, T3X will be easy to pick up. If you are completely new to the field, there is a brief introduction to T3X9 in the appendix.

The T3X9 compiler runs on FreeBSD-386 and generates code for FreeBSD-386, so you will need that system on your computer if you want to experiment with the compiler. Of course it seems curious to install 2G bytes of software in order to run a 32K-byte executable, but then these are interesting times.

Like the compiler, this book is self-contained. It includes the full compiler, its runtime support code, and enough information to understand both the source language and the target platform.

Welcome to compiler writing, enjoy the tour!

Nils M Holm, April 2017

Rules of the Game

A compiler is a program that reads the source code of a program and outputs an executable form of the same program. The most important aspect of a compiler the generation of *correct* code, i.e. the executable program must perform exactly those actions which the source program describes.

Because a compiler is a program and translates *programs* from source to executable form, it may under some circumstances compile itself. A compiler that compiles itself is called a *self-hosting* compiler. The prerequisite for this to work is that the source language and the implementation language of the compiler are the same.

Generally, three languages are involved when talking about compilers:

- the source language
- the target language
- the implementation language

The source language S is the language which the compiler "understands", i.e. the form of the programs it *reads*. This is typically a *programming language*, such as C, Pascal, or Lisp.

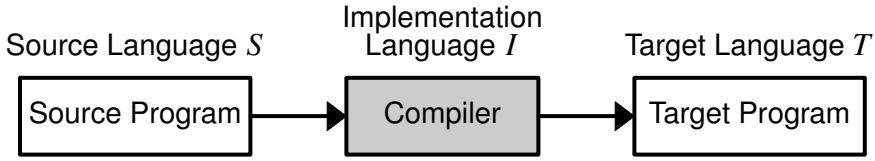

Figure 1: Compilation

The target language T is the machine language that the compiler outputs, for example: machine code for an x86 CPU or bytecode for the Java Virtual Machine (JVM). Executables are typically packaged in some executable file format, like JAR (Java ARchive), COFF (Common Object File Format), or ELF (Executable and Linkable Format).

While the term “compilation” is not necessarily limited to this scenario, we will use it to describe the transformation of a source program to an object program, as illustrated in figure 1.

Executable code is also called *object code*. Many compilers generate linkable object code instead of executable object code. In this case an additional program, a linker or linkage editor, is required to construct an executable program.

This is mostly done in order to support concepts such as separate compilation or runtime libraries. In separate compilation, chunks of a large program are compiled separately and then glued together by the linker.

Runtime libraries are a part of most compiler infrastructures. They provide pre-defined functions that can be used in a source program. Libraries are very common, but in some simple cases, the compiler may generate the pre-defined functions directly instead of referring to a library.

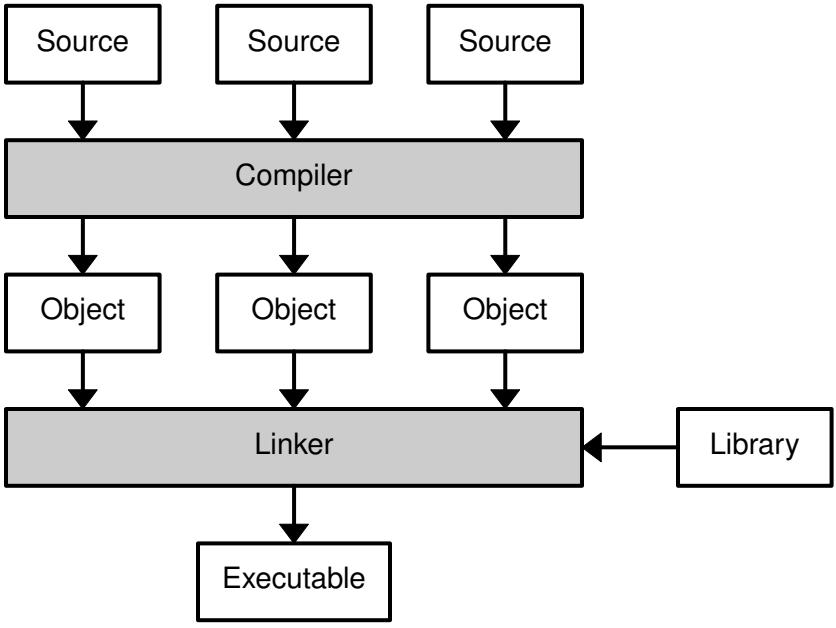

Figure 2: Separate Compilation with Linking

Figure 2 summarizes the process of separate compilation, libraries, and linking.

The compiler described in this text, however, uses the simplified model introduced initially. It reads a single source program and translates it directly to an executable file. All pre-defined functions are generated by the compiler, there is no linker and no support for runtime libraries.

The Language

The source language *and* implementation language used in this book is a subset of an obscure, little, procedural language called T3X. It is a tiny language that once had a tiny community, and it was even used to write some real-life software, like its own compiler and linker, an integrated development system, a database system, and a few simple games. It was also subject of a few college course, most probably because (1) it was reasonably well defined and documented and (2) due to the size of its community, nobody would do your homework assignments for you.

T3X looks like a mixture of C, Pascal, and BCPL. It has untyped data and typed operators, which simplifies the compiler *a lot*, but also leaves all the type checking to the programmer, which is a nightmare from the perspective of a modern software developer.

But this text is not about creating a product and make a shiny web page about it. This is about diving right into the depths of the matter and having some *fun*. And a fun language T3X is. It is interesting to see how little you need to be able to write quite comprehensible and expressive programs.

Syntax

Syntax is what a language looks like. T3X is *block-structured*, *procedural* language, which means that its programs describe *procedures* for manipulating data, i.e. “what to do with data”. It is called *block-structured*, because it is *structured* language that divides programs into blocks. A block is a chunk of source code that describes a part of a procedure.

A *structured* language uses certain constructs to describe the flow of control while a program executes, typically *selection* and *loops* (repetition).

Source code of procedural languages is mostly organized in the form a hierarchy consisting of a programs, procedures or functions, declarations, statements, and expressions. The most

abstract view is the program, the least abstract one the expression. See figure 3 for an illustration.

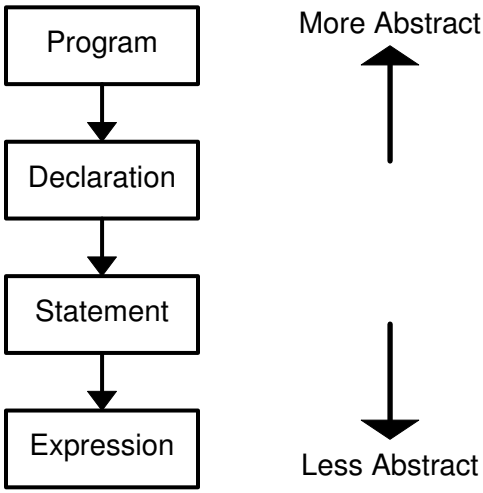

Figure 3: Elements of Block-Structured Languages

In procedural languages:

- programs contain declarations, statements, and expressions
- declarations contain statements and expressions
- statements contain expressions

If you are familiar with C or Pascal or Java, the T3X syntax will look quite familiar. Here is the infamous bubblesort algorithm in T3X:

```
! This is a comment
bubblesort(n, v) do var i, swapped, t;
    swapped := %1;
    while (swapped) do
        swapped := 0;
        for (i=0, n-1) do
            if (v[i] > v[i+1]) do
                t := v[i];
                v[i] := v[i+1];
                v[i+1] := t;
                swapped := %1;
```

```
            end
        end
    end
end
```

bubblesort(n, v) starts the declaration of the procedure *bubblesort* with the formal arguments n and v. The body of the procedure is a block statement (or *compound statement*) enclosed in the keywords **DO** and **END**. The compound statement declares the local variables i, *swapped*, and t.

Assignment is done by the **:=** operator (and equality is expressed with **=**). The statement **FOR (i=0, n-1)** counts from 0 to $n-2$. The i^{th} element of a vector (or array) v is accessed using **v[i]**. Elements are numbered $v[0] \cdots v[n-1]$.

The lexeme **%1** denotes the number -1. You could also write **-1**, but there is a subtle difference: the former is a value, and the latter is an operator applied to a value, which will not work in contexts where a constant is expected.

Furthermore, **/** and **\/** denotes logical (short-circuit) AND and OR, and **X->Y:Z** means "if x then y else z", just like **x?y:z** in C.

IF with an **ELSE** is called **IE** (If/Else).

You will pick up the rest of the T3X syntax as we walk through the compiler source code. If you are interested, there is a brief introduction to T3X in the appendix.

Semantics

Semantics is how the syntax is interpreted. Note that "interpreted" does not imply the use of interpreting software here. Interpretation can be done at various levels, and in the case of the T3X compiler presented here, the code will eventually be interpreted by a 386 (or x86) CPU.

Interpretation in this case is a question of meaning. What does a statement like

```
WHILE (swapped) DO ... END
```

mean? To you, it is probably obvious that it means: "while the value of *swapped* is a 'true' value, repeat everything between **DO** and **END**".

But now we need to know what a "true" value is and what "repetition" means. This is what the semantics of a language describe.

For example, the expressions **v[i]** and **s::i** both denote the i^{th} element of a vector. However, the first variant describes the i^{th} machine word in a vector of machine words, and the second one describes the i^{th} *byte* in a *byte vector*.

In this book, semantics will be specified in two ways: by diagrams describing program flow and by short machine code sequences that resemble the meaning of a language construct.

For instance, the meaning of the **[]** operator in **v[i]** would be specified as

```
shl     $0x2,%eax
pop     %ebx
add     %ebx,%eax
mov     (%eax),%eax
```

assuming that $i = \%eax$ and the address of v is on top of the stack.

(Note that AT&T notation is used here, so **mov a,b** means "move a to b". See the 386 assembly summary in the appendix for further details.)

The exact semantics of the T3X language will be explained informally during the tour through the compiler source code.

The Target Architecture

The target language of the compiler discussed here will be code for the 386 processor family. The code will be packaged in an ELF-format file, and it will use the FreeBSD application binary interface (ABI).

386 machine code can be interpreted by a variety of modern processors, even the latest members of the x86 family. Unlike x86-64 code (64-bit x86 code), it can also be interpreted by older, 32-bit processors.

The ELF format is very popular in modern Unix-based operating systems. For instance, it is used by all modern BSD and Linux systems.

The compiler outputs machine programs that use the FreeBSD ABI to communicate with the operating system, which means that the code generated by the compiler will run on FreeBSD systems.

If you do not have a FreeBSD system, you can either install one in a virtual machine, or change the ABI-specific parts of the runtime support functions of the compiler to use a different ABI. This will be explained later in the text (page 107).

The CPU- and ABI-specific parts of the compiler are all contained in a single procedure in the source code, so porting it to a different operating system or a different 32-bit CPU could be an interesting project once you have finished the tour.

Execution Model

In the ideal case, the execution model of a compiler is exactly the target machine for which code is generated. In fact, CPUs and compilers are designed in such a way that the mapping from source code to machine instructions is as straight-forward as possible.

However, such an ideal mapping is also non-trivial, so in practice, there is often an additional layer between the source language and the target language. We call this additional layer the *execution model* of the compiler.

Statement Parser

The *statement* is the fundamental building block of a procedural program. While an *expression* has a value, a statement *does something*. For instance, a **RETURN** statement ends interpretation of a function and returns a value, an assignment changes the value of a variable or vector element, and a loop repeats a part of a program.

Many statements operate on other statements. For instance, the **IF** statement runs another statement conditionally or **WHILE** repeats another statement. So statements, like expressions, are inherently recursive structures.

```
halt_stmt → HALT constval ;
```

Figure 34: HALT Statement Syntax Rule

One of the simplest statements is the **HALT** statement, which just has a cvalue as its argument and, like all statements, is terminated by a semicolon. See figure 34 for its syntax rule. It is implemented by **halt_stmt()**, below.

```
halt_stmt() do
    T := scan();
    gen(CG_HALT, constval());
    xsemi();
end
```

The **RETURN** statement has two forms: with and without a value to return. When no value is specified, zero is returned. The value is returned in the accumulator. The syntax rules of **RETURN** are shown in figure 35.

```
return_stmt → RETURN ;
            | RETURN expr ;
```

Figure 35: RETURN Statement Syntax Rules

The **return_stmt()** function, which implements the **RETURN** statement, also makes sure that the statement appears in a function context, because it is not allowed in the main program.

If the function containing the statement has any local variables, **return_stmt()** generates code to deallocate the space used by them before returning.

```
return_stmt() do
    T := scan();
    if (Fun = 0)
        aw("can't return from main body", 0);
    ie (T = SEMI)
        gen(CG_CLEAR, 0);
    else
        expr(1);
    if (Lp \= 0) do
        gen(CG_DEALLOC, -Lp);
    end
    gen(CG_EXIT, 0);
    xsemi();
end
```

The syntax rules for the **IF** and **IE/ELSE** statements can be found in figure 36. Note that no terminating semicolon is included in either rule, because it will be supplied by the *stmt* part of each rule.

```
if_stmt → IF ( expr ) stmt
        | IE ( expr ) stmt ELSE stmt
```

Figure 36: IF Statement Syntax Rules

The **IF** statement **IF (a) b;** is the statement equivalent to the expression **a/\b**, i.e. it executes b only if a is true. Hence the control flow diagram of **IF**, as shown in figure 37, looks similar to the one of the short-circuit AND operator (figure 30). A similar observation can be made regarding the conditional operator (figure 33) and the **IE**/**ELSE** statement (also figure 37).

When compiling **IE**/**ELSE**, the **if_stmt()** function uses the same jump-around-jump method as the **expr()** function implementing the conditional operator.

When **if_stmt()** is called with $alt = 1$, it expects an "alternative" statement, i.e. it compiles **IE**/**ELSE**, otherwise it compiles **IF**.

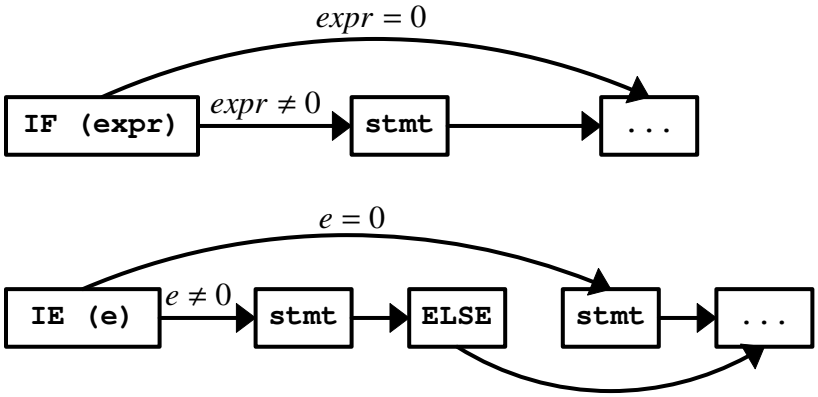

Figure 37: IF/IE Statement Control Flow

```
if_stmt(alt) do
    T := scan();
    xlparen();
    expr(1);
    gen(CG_JMPFALSE, 0);
    xrparen();
    stmt();
    ie (alt) do
        gen(CG_JUMPFWD, 0);
        swap();
        gen(CG_RESOLV, 0);
        expect(KELSE, "ELSE");
        T := scan();
        stmt();
    end
    else if (T = KELSE) do
        aw("ELSE without IE", 0);
    end
    gen(CG_RESOLV, 0);
end
```

The **WHILE** loop, as depicted in figure 38, has structure similar to the **IE** statement. It also uses a jump around jump, but the second jump, after the statement, goes back to the beginning of the loop.

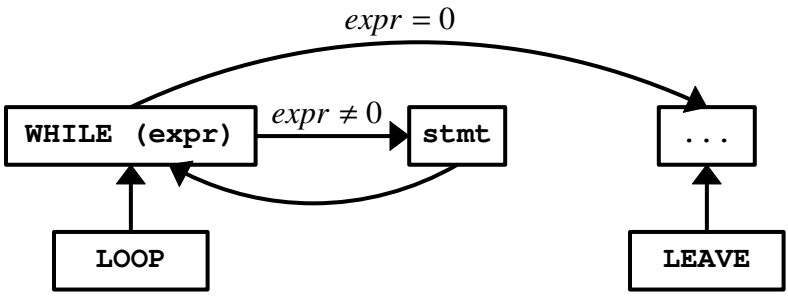

Figure 38: WHILE Statement Control Flow

In a **WHILE** context, the **LOOP** statement jumps to the point where *expr* is tested. This is a backward jump that can be generated immediately. The **LEAVE** statement, though, generates a forward jump, which has to be backpatched at the end of the loop.

While_stmt(), which implements the **WHILE** statement, stores the address for re-entering the loop in **Loop0** and the first **Leaves** slot used by it in *olv*. It restores **Loop0** and **Lvp** at the end, so loops can be nested.

while_stmt → WHILE (expr) stmt

Figure 39: WHILE Statement Syntax Rule

The syntax rule for **WHILE** can be found in figure 39.

```
while_stmt() do var olp, olv;
    T := scan();
    olp := Loop0;
    olv := Lvp;
    gen(CG_MARK, 0);
    Loop0 := tos();
    xlparen();
    expr(1);
    xrparen();
    gen(CG_JMPFALSE, 0);
    stmt();
    swap();
    gen(CG_JUMPBACK, 0);
    gen(CG_RESOLV, 0);
```

```
    while (Lvp > olv) do
        push(Leaves[Lvp-1]);
        gen(CG_RESOLV, 0);
        Lvp := Lvp-1;
    end
    Loop0 := olp;
end
```

Figure 40 shows the syntax rules for the **FOR** statement.

```
for_stmt → FOR ( expr , expr , constval ) stmt
         | FOR ( expr , expr ) stmt
```

Figure 40: FOR Statement Syntax Rules

The compiled structure of the **FOR (v=x1, x2, c) stmt** construct is a bit more complicated, as can be seen in figure 41. The initialization part on the left is run once and then the exit condition is tested. It depends on the sign of the third parameter of **FOR**. When the exit condition does not hold, the statement is run, and *then* the increment part of the statement is executed.

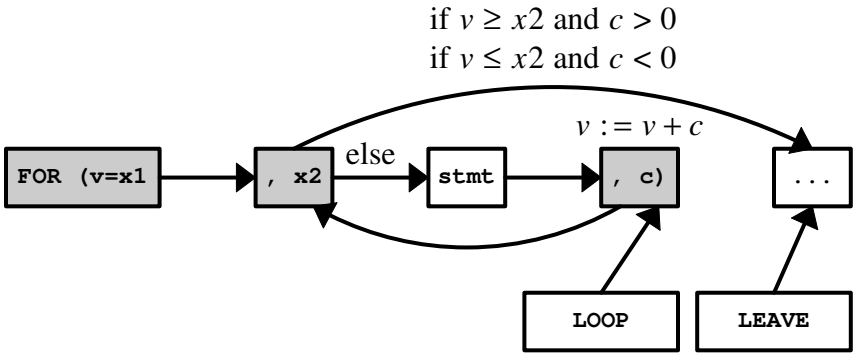

Figure 41: FOR Statement Control Flow

Therefore, the head of the **FOR** construct (in gray) has been spread around the statement in figure 41. After running the statement, the variable v is incremented and then the loop is repeated by jumping back to the test part.

List of Figures

Index

Program Symbols

Definitions