PRACTICAL COMPILER CONSTRUCTION

Nils M Holm

Contents

Preface (First Edition)

A lot of time has passed since the publication of my previous compiler textbook, *Lightweight Compiler Techniques* (LCT), in 1996. I had sold some spiral-bound copies of the book on my own back then and later published a nice soft-cover edition on Lulu.com in 2006. To my surprise the paperback sold more copies than I had sold when the book was first published, and a few people even asked me if I planned to finish the final chapter on code generation and optimization. However, I never found the motivation to do so. Until now.

Instead of polishing and updating LCT, though, I decided to create an entirely new work that is only loosely based on it. As you probably know, sometimes it is easier to toss all your code to the bin and start from scratch. So while this book is basically an updated, extended, improved, greater, better, funnier edition of LCT, it differs from it in several substantial points:

Instead of the T3X language, which never caught on anyway, it uses the wide-spread C programming language (ANSI C, C89) as both the implementation language and the source language of the compiler being implemented in the course of the text.

It includes much more theory than LCT, thereby shifting the perspective to a more abstract view and making the work more general and less tightly bound to the source language. However, all theory is laid out in simple prose and immediately translated to practical examples, so no prior knowledge in the field is required.

The main part of the book contains the complete code of the compiler, including its code generator, down to the level of the emitted assembly language instructions. It also describes the interface to the operating system and the runtime library—at least those parts that are needed to implement the compiler itself, which is quite a bit!

Then there is a self-contained part dealing with code synthesis and optimization. The approaches discussed here are mostly taken from LCT, but explained in greater detail and illustrated with lots of diagrams and tables. This part covers topics like abstract program rewriting, register allocation, machine-dependent peephole optimization and common subexpression elimination.

The final part of the book describes the bootstrapping process that is inherent in the creation of a compiler from the scratch, i.e. it explains where to start and how to proceed when creating a new compiler for a new or an existing programming language.

The intended audience of this book is “programmers who are interested in writing a compiler”. Familiarity with a programming language is required, working knowledge of the C language is helpful, but a C primer is included in the appendix for the brave of heart. This book is intended as a practical “first contact” to the world of compiler-writing. It covers theory where necessary or helpful, explains the terminology of compilers, and then goes straight to the gory details.

The book is not a classic teaching book. It does not take a break and requires you to pace yourself. Be prepared to read it at least twice: once concentrating on the prose and another time concentrating on the code. The goal of the book is to bring you to a point where you can write your own real-world compiler with all the bells and whistles.

The code spread out in the book is a compiler for a subset of the C programming language as described in the second edition of the book *The C Programming Language* (TCPL2) by Brian W. Kernighan and Dennis M. Ritchie. The compiler targets the plain 386 processor without any extensions. It requires the GNU 386 assembler to compile its output. Its runtime environment has been designed for the FreeBSD[1] operating system, but I suspect that it will also compile on Linux—or should be easy to port, at least.

The complete machine-readable source code to all versions of the compiler can be found at **`http://www.t3x.org`**.

[1] See **`http://www.freebsd.org`**.

All the code in this book is in the public domain, so you can do whatever you please with it: compile it, use it, extend it, give it to your friends, sell it, whatever makes you happy.

Nils M Holm
March 2012

Preface

SubC and Practical Compiler Construction were an unexpected success. It is probably still the best-selling one of my books, and people have done interesting things with it, which is probably the greatest praise that the work of a human being can attract. Since the publication of the first edition, the SubC compiler has been ported to Linux, OpenBSD, Mac OS (Darwin), Windows, and DOS. It also supports the ARMv6, 8086, and x86-64 CPUs now, and some of the optimizations explained in this book have been added. The code of the new compiler, however, has evolved substantially and is now quite a bit more complex than the code described initially.

This text still explains the original SubC compiler as it first appeared in the first edition of the book. However, there are some differences. All bugs that have been discovered since the publication of the first edition have been fixed. The dependency on the host system's standard C library has been removed. The system call interface has been made more Unix-like in order to increase the compatibility with existing code.

Many lessons have been learned during the further development of SubC. One is that the C language is really complex, and writing a compiler for even a tiny subset of C is a major undertaking with many small details waiting to be overlooked. This is why the second edition still discusses the original SubC compiler instead of one of its descendants. Another lesson is that emulating **`signal()`** is probably not worth the pain that it inflicts, so it has been removed from this edition. (You can still have it by linking against the host system's C library.)

This book has been available in PDF format only since 2016, because the print master no longer adhered to evolving standards. This was mostly caused by typesetting the book in TeX, which is simply to complicated for my little brain (i.e. this is not a fault of TeX, but a limitation of my understanding of it). So the obvious remedy was to change the typesetting to TROFF, which has never let me down in all the decades of writing. I started this endeavor in

2017, but then lost interest along the way. However, people kept asking about the print edition and so I eventually decided to finish the work on the second edition. It was a much greater effort than originally expected, but I think it was worth it. The second edition is a better book than the first one and, as of now, the book is back in print!

Nils M Holm
May 2022

Compilers

Traditionally, most of us probably understand a *compiler* as a stand-alone program that reads source code written in a formal language and generates an executable program for a specific computer system. Indeed this is how most compilers work, even today. In the past it has been common to refer to interactive programming language implementations as “interpreters”. Even the distinction between “interpreted” and “compiled” languages has been made. These boundaries are highly artificial, though.

Many languages that have been referred to as interpreted languages in the past have gotten compilers long ago. BASIC and LISP, for instance, both of which were first implemented as “interpreters”, soon evolved into implementations that generated native machine code. Most modern LISP systems compile code to machine code behind the scenes, even in interactive mode. In the age of just-in-time compilation, interactive program execution is no longer an evidence for what we commonly call “interpretation”.

What is “program *interpretation*” anyway? It is nothing but the process of performing certain tasks based on the statements written down in a formal program. Does it really matter whether this interpretation is performed by a human being, by another program on the same or a different machine, or by a program that happens to be cast in silicon (a.k.a. a CPU)?

I think it is time to define the terms “compilation” and “interpretation” in a more precise way. In this book, *compilation* will refer to the process of translating a formal language S to a formal language O by a program C. “S” is called the *source language* of the process, “O” is called the *object language*, and “C” the compiler. Interpretation is simply a synonym for program execution, be it by a hardware machine, a virtual machine, or a human being.

Given this definition, even the first BASIC interpreters were in fact compilers *and* interpreters at the same time. The compiler part translated the input language BASIC into some internal form that was suitable for efficient interpretation. The interpreter part gave meaning to the internal form by “running”—i.e. executing—it. The stages of the process were tightly coupled in this case, so the compiler and interpreter were inseparable parts of the language implementation but, nevertheless, both stages were present.

Another rather useless classification is that of compiled and interpreted languages. Even primordial BASIC is compiled to a token stream and not stored in memory as literal text, and even the output of the most clever and aggressive optimizing compiler is interpreted by a CPU in the end. And then any language that can be interpreted can be compiled as well. If we want to make this differentiation at all, then “interpreted” (by software) versus “compiled” (to native code) is a property of the *implementation* and not one of the language.

Even a code beautifier—a program that improves the formatting of an input program—is a compiler. It just happens to have the same source and object language. The same is true for a code obfuscator, a program that scrambles source code while preserving its meaning.

Compilers come in various flavors, from simple translators that turn their input into a different representation, like early BASIC tokenizers, to full-blown stand-alone optimizing compilers with extensive runtime libraries. However, they all employ the same basic principles.

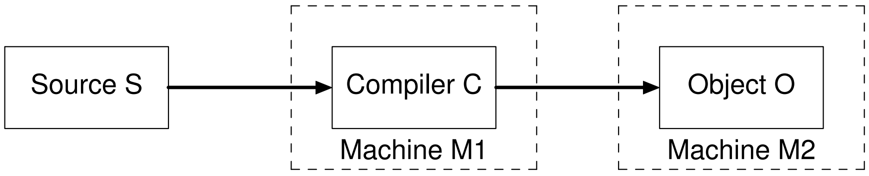


Figure 1: Compiler model

A compiler is a program C written in language L running on machine M_1 that translates source language S to object language O, which may be executable on a machine M_2. S, O, and L may be all different, all the same, or any variation in between. Both M_1 and M_2 may be actual hardware or virtual machines. Here is a non-conclusive list of variations:

- When $S \neq O$, then the compiler is an “ordinary” compiler.
- When $S = L$, then the compiler is *self-hosting*. [1]
- When $M_1 \neq M_2$, the compiler is a *cross-compiler*. [2]
- When $S = O$, the compiler is a code beautifier, obfuscator, etc.
- When O is not a formal language, C is a code analyzer. [3]

In this book we will concentrate on the first and most common case. We will have a thorough look at a compiler for a subset of the ANSI C programming language (C89) for the 386 architecture. We will explore the stages of compilation from source code analysis to native code generation. We will also cover the runtime library and the interface to the operating system (FreeBSD).

The compiler has been designed to be simple and portable. Understanding its principles does not require any prior knowledge other than mastery of C or a comparable programming language. For those who are not familiar with the language, a terse C primer can be found in the appendix (pg. 421). The compiler itself is written in pure ANSI C, so it should compile fine on any 32-bit system providing a preexisting C compiler. Porting it to other popular free Unix clones (e.g. Linux) should be easy. Once bootstrapped, the compiler is capable of compiling itself.

How Does a Compiler Work?

All but the most simple compilers are divided into several *stages* or *phases*, and each of these phases belongs to one of the two principal parts of the compiler called its *front-end* and its *back-end*. The front-end is located on the input side of the compiler. It analyzes the source program and makes sure that it is free of formal errors, such as misspelled keywords, wrong operator symbols, unbalanced parentheses or scope delimiters, etc. This part also collects information about symbols used in the program

[1] It can compile itself.
[2] A compiler that generates code for a CPU and/or platform other than its host CPU and/or platform.
[3] A generator of code metrics, code quality reports, etc.

and associates symbol names with values, addresses, or other entities.

The back-end is on the output side of the compiler. It generates an object program that has the same meaning as the source program read by the front-end. We say that the compiler “preserves the *semantics* of the program being compiled”. The semantics of a program is what it “does” while the *syntax* of a program is what it “looks like”. A compiler typically reads a program having one syntax and generates a program with a different syntax, but the two programs will always have the same semantics. [1]

Preservation of semantics is the single most important design goal of a compiler. The semantics of a source language is mostly specified semi-formally and sometimes formally. Some formal semantics are even suitable for generating a compiler from them, but this is not (yet) the normal case these days. Languages with formal semantics include, for example, Scheme or Standard ML (SML). Some languages with informal or semi-formal semantics are Common Lisp, Java, C++, and C.

The front-end of a compiler is typically portable and deals with the input language in an abstract way, while the back-end is normally coupled to a specific architecture. When a compiler is intended to be portable among multiple architectures, the front-end and back-end are loosely coupled, for example by passing information from one to the other using a formal protocol or a “glue language”. Compilers for single architectures allow for a tighter coupling of the stages and therefore typically generate faster and/or more compact code.

Most phases of compilation belong either to the front-end or the back-end, but some may belong to both of them. The optimization phase, which attempts to simplify the program, is often spread out over both parts, because there are some optimizations that are best applied to an abstract representation of a program, while others can only be performed in the back-end, because they require some knowledge about a specific target. In compiler-

[1] With the sole exception of code analyzers, which are intended to generate human-readable meta-information instead of executable programs.

speak, a *target* is the architecture on which the generated code will run. The SubC compiler discussed in this book will target the 386 processor.

Phases of Compilation

All the phases of compilation are outlined in detail in figure 2 (shaded boxes denote processes, clear boxes represent data). The first phase on the input side of the compiler is the *lexical analysis*. This step transforms the source program into a stream of small units called *tokens*. Each token represents a small textual region of the source program, like a *numeric literal*, a *keyword*, an *operator*, etc. This phase also detects and reports input characters that are not part of the source language and sequences of text that cannot be tokenized. [1]

To illustrate the principle of tokenization, here comes a small sample program, in its "natural" form on the left and with blanks between the individual tokens on the right:

```
void printrev(char *s) {         void printrev ( char * s ) {
    if (*s) {                        if ( * s ) {
        printrev(s+1);                   printrev ( s + 1 ) ;
        putchar(*s);                     putchar ( * s ) ;
    }                                }
}                                }
```

The lexical analysis phase is also referred to as the *scanner*. In fact scanning does a bit more than the above. It does not only split up the source program into manageable parts, but it also categorizes the individual parts. For instance, the scanner output of the above program may look like this (where each symbol denotes a unique small integer):

```
VOID IDENT LPAREN CHAR STAR IDENT RPAREN LBRACE
IF LPAREN STAR IDENT RPAREN LBRACE
IDENT LPAREN IDENT PLUS INTLIT RPAREN SEMI
IDENT LPAREN STAR IDENT RPAREN SEMI
RBRACE RBRACE
```

[1] If such sequences exist in the source language; see pg. 82.

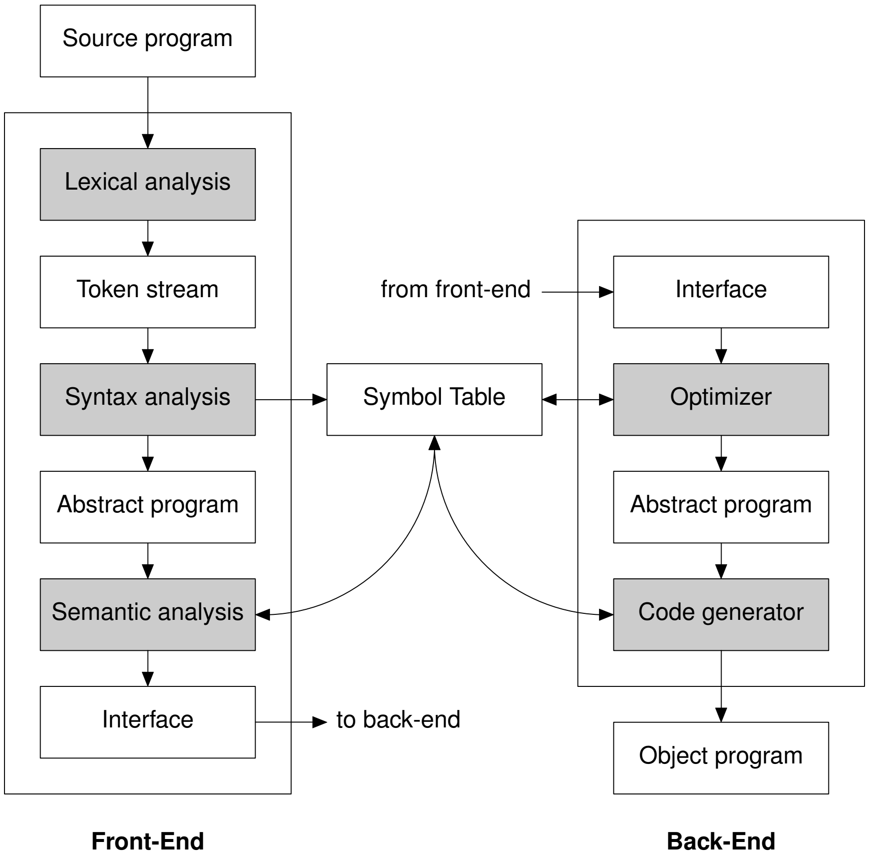


Figure 2: Phases of compilation

Of course this representation loses some information, because all the symbols (like **`s`** or **`putchar`**) are simply referred to as **`IDENT`** (identifiers), and the value of the **`INTLIT`** (integer literal) is lost. This is why the scanner adds *attributes* to some of the tokens, resulting in:

```
VOID IDENT(revstr) LPAREN CHAR STAR IDENT(s) RPAREN LBRACE
IF LPAREN STAR IDENT(s) RPAREN LBRACE
IDENT(revstr) LPAREN IDENT(s) PLUS INTLIT(1) RPAREN SEMI
IDENT(putchar) LPAREN STAR IDENT(s) RPAREN SEMI
RBRACE RBRACE
```

This would be the output of the lexical analysis stage. Some very simple compilers may perform in-place textual comparisons instead of scanning their input. To find out whether the current input token is the **if** keyword, for instance, they would do something like

```
if (!strncmp(src, "if", 2) && !isalnum(src[2])) {
    /* got an IF */
}
```

This works fine for small languages with few operators and keywords, but does not scale up well, because quite a few comparisons may have to be performed to reach a procedure that can handle a specific token. Even the SubC language, which is still a subset of C89, knows 77 different token types, so tens of comparisons will have to be made on average before a token can eventually be processed. In larger languages such as C++ or COBOL, the number of comparisons may easily range in the hundreds and involve longer strings due to longer keywords. This is where tokenization really pays off, because it allows to compare tokens using the == operator:

```
if (IF == Token) {
    /* got an IF */
}
```

The next step is to make sure that the sequence of tokens delivered by the scanner forms a valid *sentence* of the source language. Like natural languages, formal languages have grammars that define what a correct sentence looks like. The grammar describes the *syntax* of a language, and the *syntax analyzer* compares the token stream to the rules of a grammar. The syntax analyzer is also called the *parser* and the analysis itself is called “parsing”.

The parser is the heart of the compiler: it pulls the source program in through the scanner and emits an object program through the back-end. The syntax of the source program controls the entire process, which is why this approach is called *syntax-directed translation*.

Most of the error reporting is also done by the parser, because most errors that can be caught by a compiler are syntax errors, i.e.

input sequences that do not match any rule in the grammar of the source language.

When parsing succeeds, i.e. no formal errors are found, the parser transforms the token stream into an even more abstract form. The *abstract program* constructed at this stage may have various forms, but the most common and most practical one is probably the *abstract syntax tree* (*AST*).

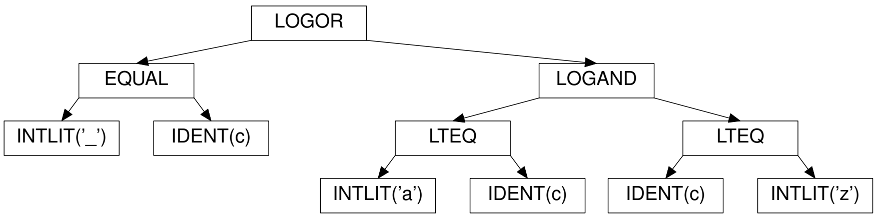


Figure 3: Abstract syntax tree (AST)

ASTs tend to be rather large, which is why we use a smaller example to illustrate them. The C expression

```
'_' == c || 'a' <= c && c <= 'z'
```

would generate the AST displayed in figure 3. (EQUAL denotes the == operator, LTEQ is <=, LOGAND is `&&`, and LOGOR is `||`). The AST reflects the structure of the input program according to the rules specified in the source language's grammar (we will discuss grammars in great detail at a later point). For example, the AST reflects the fact that the `&&` operator has a higher *precedence* than the `||` operator. [1]

The output of the parser is suitable for more complex operations, because it allows to refer to subsentences by their root nodes. For instance, the subexpression

```
'_' == c
```

of the above expression may be referred to simply by pointing to the EQUAL node in the corresponding AST.

[1] When these two operators occur in the same sentence without any explicit grouping (by parentheses), `&&` has to be evaluated before `||`.

The parser also builds the *symbol table*. Symbol tables are used to store the names of identifiers and their associated meta data, like addresses, values, types, *arity,* [1] array sizes, etc. Most compilers for procedural languages employ two symbol tables: one for global symbols and one for local symbols and procedure parameters.

The symbol table is heavily used by all subsequent phases of compilation. The next stage, *semantic analysis*, uses it to look up all kinds of properties of identifiers. It performs tasks such as *type checking* and context checking. The former has to be done on the semantic level, because the correctness of a statement like

```
x = 0;
```

cannot be decided without referring to meta information in the symbol table. When **`x`** is an array or a function, for instance, the above sentence is semantically incorrect. Its syntax is correct, but it makes no sense, because C cannot assign a value to an array or a function. This kind of error cannot be detected on a purely syntactic level, because an indeterminate number of tokens may appear between the definition of **`x`** and a reference to it.

The reporting of undefined identifiers is also done at this stage, because the semantics of a statement cannot be verified in a static language like C without knowing the properties of an identifier. (In a dynamic language such tests can sometimes be postponed to the run time of the program.)

Context checking finds out whether a statement appears in a proper context. For instance, a **`break`** statement is only valid inside of a loop or a **`switch`** statement and would be flagged otherwise. This kind of analysis could also be performed at the syntax level, but this often leads to cumbersome formal specifications, so in practice it is mostly performed at the semantic level.

The *interface* between front-end and back-end can be implemented in a variety of ways. In the most simple case the parser calls the procedures of the code generator directly. In this

[1] The number of arguments of a procedure.

case the back-end does not even need access to the symbol table. When the back-end performs more complex operations than just emitting predefined fragments of code, though, the interface must provide procedures for accessing the symbol table from within the back-end and transferring the abstract program to the back-end stages. The most sophisticated approach would be to generate code in a portable abstract *glue language* and pass that program to the back-end. The abstract program would contain all information needed for optimizing the code and synthesizing native machine instructions. No access to the symbol table would be necessary in this case.

No matter in which way the abstract program and the related meta information is transported to the back-end, the next step would be the *optimization* phase. In figure 2, optimization has been placed in the back-end exclusively, but there are some optimizations that can be performed in the front-end as well. When both is possible, an optimization should be performed in the front-end. Only machine-dependent code should be moved to the back-end. A typical platform-neutral optimization is *constant expression folding*, which collapses subtrees of constant factors into single factors. It is illustrated in figure 4.

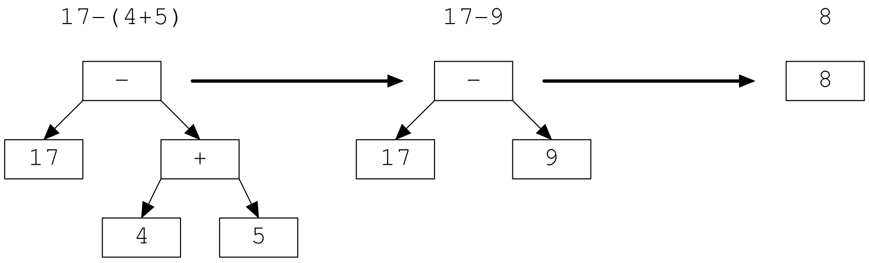


Figure 4: Constant folding

Constant folding is particularly important, because many operations that appear to be simple at the language level may expand to constant expressions when generating an abstract program. For instance, the expression

```
a[0] = 0;
```

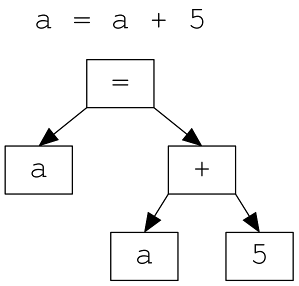
(where **`a`** is an array) may expand to a constant expression adding zero to the address of the array, which can be optimized away. However, adding an arbitrary (non-zero) constant to the address of an array would require to know what the code loading the address will look like, so it must be done in the back-end. Even constant folding may require some knowledge about the target architecture, for example for detecting numeric overflow. In the case of C, portable constant folding must be limited to the range $-32768 \cdots 32767$, because this is the minimal numeric range of the **`int`** type.

When in doubt, it is always better *not* to perform an optimization. It is better to generate inefficient code than to break the preservation of semantics.

The abstract program generated by the optimizer may be an AST or a more specialized representation, like three-address code, which emulates a virtual architecture. The optimizer may also generate trees of actual machine code instructions for the target platform, thereby integrating the stages of back-end optimization and code synthesis.

The final stage, the *code generator synthesizes* instructions for the target machine and emits them to the output. This is why it is also called the *emitter*. Machine instructions are typically generated from entire subtrees rather than individual nodes of the abstract program because, up to some limit, considering larger subtrees will lead to more efficient code.

For instance, a naive synthesizer may examine each single node of the program



and generate code that works as follows:

- load a into register $r1$;
- load 5 into register $r2$;
- add $r2$ to $r1$;
- store $r1$ in a.

A synthesizer that examines the entire expression before emitting any code could recognize that a constant is being added to a variable and synthesize the single instruction

- add 5 to a.

This process is called "synthesis" because it synthesizes more complex instructions from simpler ones contained in the abstract program. The front-end stages of a compiler decompose complex instructions formulated in the source language into simple instructions of an abstract form. The back-end stages then reconstruct complex instructions from simpler ones, trying to make best use of the features of the target architecture.

The overall task of a compiler may be thought of as *mapping* a source program P_S of a source language S to the most efficient program P_O of an object language O that has the same meaning as P_S.

A Simplified Model

The SubC compiler, which will be discussed in the main part of this book, will use a simplified model of compilation as depicted in figure 5.

Like the full model, this simplified model uses a scanner producing a token stream that is fed to the parser. However, the semantic analysis phase is tightly coupled to the parser in this model, so the compiler performs semantic analysis basically "inside of" the parser. No abstract program is generated and the symbol table is also set up in the syntactic analysis phase.

The interface to the back end is a simple procedure call interface, where each procedure in the code generator emits a parameterized code fragment. The parameters are passed to the code generator via procedure calls. Sometimes these parameters identify slots in the symbol table, allowing the back-end direct access to the attributes of symbols. Because the code generator is only loosely coupled to the rest of the compiler, retargeting the compiler, i.e. porting it to a new target platform, involves just the creation of a new set of procedures that emit the desired object code.

The SubC compiler does not have an optimizer and it often emits inefficient and sometimes even redundant code. Emphasis was put on correctness rather than cleverness. This is the price that we pay for a simple and easy-to-comprehend compiler. However, SubC is not *that* bad after all. The entire compiler including its full runtime library bootstraps itself in just about two seconds on a mainstream 750MHz[1] processor, and it is quite sufficient for a lot of every-day programming tasks.

[1] No, this text is not that ancient; the author is just content with old hardware. He has upgraded from 600Mhz to 750MHz since the first edition in 2012, though.

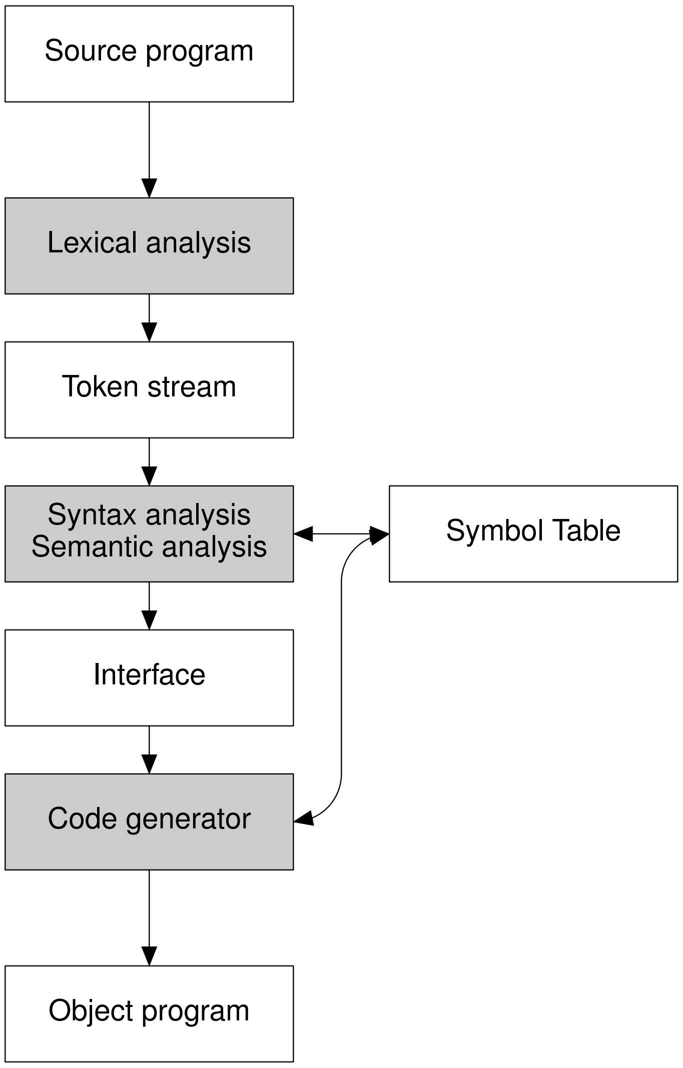


Figure 5: A simplified compiler model

The Peculiarities of C

Viewed superficially, C is a simple, rather low level, block-structured, procedural language. However, it is also quirky as hell. It has quite some hidden complexity in places where you would least expect it. It is not a language that an aspiring compiler-writer would want to implement in their first project. Not without the help of a decent textbook, that is.

For instance, C has 15 levels of operator precedence, quite a weird declaration syntax, and hairy *pointer semantics*. Quick, what does

```
int *(*foo[17])();
```

declare? And what do the following expressions deliver?

```
a[0]                        **a++
*a                          *(*a)++
*a[0]                       (**a)++
(*a)[0]                     ++**a
*a++                        *++*a
(*a)++                      **++a
*++a
```

Even the SubC compiler discussed in this book will have to get all of the above expressions right! [1]

Most of the complexity of the C compiler is hidden in the expression parser, the part of the parser that analyzes the smallest parts of the language, the formulae used in its statements. It not only does have to support all of the above constructs, it also has to do proper pointer addition and subtraction, even in the += and −= operations, etc. There is quite a bit of work hidden in this superficially simple language.

On the other hand, the C standard ("The C Programming Language, 2nd Ed." (TCPL2) by K&R) gives quite a bit of leeway to the implementor. This is a good thing for a compiler-writer. It

[1] But not the declaration.

allows experienced implementors to choose the most efficient approach and the beginner to use the most simple solution. This is also the primary reason why C is known as a fast and efficient language: it is *underspecified*, i.e., it allows the implementor to interpret the specification quite freely in order to squeeze out a few more optimizations. However, this is often done at the cost of clarity to the user (of the C language). For instance, the statement

```
f(putchar('a'), putchar('b'), putchar('c'));
```

may print any permutation of “a”, “b”, and “c”, because the standard leaves the *order of argument evaluation unspecified*. “Unspecified” means that a compiler can evaluate the arguments to a function in any order it likes. It can even use different orders at different optimization levels or a different order for exactly the same statement when compiled twice in a row. It could throw a dice. “Unspecified” means: *just do not rely on it*.

It gets even worse when adding in the increment operators. The program fragment

```
x = 0; f(x++, x++);
```

may pass $(0,0)$ to f or $(0,1)$, or $(1,0)$. This is because the standard only says that the post-increment has to take place before the end of the statement. [1] So the compiler may take the first x, increment it, and then take the second x [giving $(0,1)$]. Or it may evaluate the second argument first [$(1,0)$]. Or it may decide to increment x by two after calling the function, resulting in $(0,0)$.

To the compiler writer, of course, such leeway is a great advantage, because it allows them to pick any variant that is convenient to them—and therefore to us! In this text, we will always choose the most simple and obvious version in order to keep the compiler source code manageable.

Studying a compiler for a language often changes the way in which we see a language, and mostly for the better. Even if you already have some programming experience in C, this textbook may reveal some of the more subtle details of the language to you.

[1] More specifically: before the next “sequence-point”, which would be the `;` in this case.

For example:

Why are statements like `i = i++;` not a good idea?

Is `*x[1]` the indirection of an array element or the array element of an indirection?

Why will the declaration

```
extern char *foo[];
```

bring you into trouble when the identifier was defined in a different file as

```
char **foo;
```

while there is absolutely no difference between the meanings of the following two declarations?

```
int main(int argc, char **argv);
int main(int argc, char *argv[]);
```

Answers

In case you want to verify your answers to the questions at the beginning of this chapter:

`int *(*foo[17])();`	An array (size 17) of pointers to functions returning pointer to int
`a[0]`	return first element of `a`
`*a`	return object pointed to by `a`
`*a[0]`	object pointed to by `a[0]`
`(*a)[0]`	return first element of `*a`
`*a++`	return `*a`, then increment `a`
`(*a)++`	return `*a`, then increment `*a`
`*++a`	increment `a`, then return `*a`
`**a++`	return `**a`, then increment `a`
`*(*a)++`	return `**a`, then increment `*a`
`(**a)++`	return `**a`, then increment `**a`
`++**a`	increment `**a`, then return `**a`
`*++*a`	increment `*a`, then return `**a`
`**++a`	increment `a`, then return `**a`