PRACTICAL COMPILER CONSTRUCTION

Nils M Holm

Nils M Holm

PRACTICAL COMPILER CONSTRUCTION

Nils M Holm, 2012
Print and distribution:
Lulu Press, Inc.; Raleigh, NC; USA

Preface

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 there is no reason to be afraid!

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!

There is a complete 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

also covers 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 the compiler can be found at `http://www.t3x.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, Mar. 2012

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

Contents

In Memoriam
Dennis M. Ritchie
1941–2011

Part I

Introduction

Chapter 1

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 source 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 with 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.

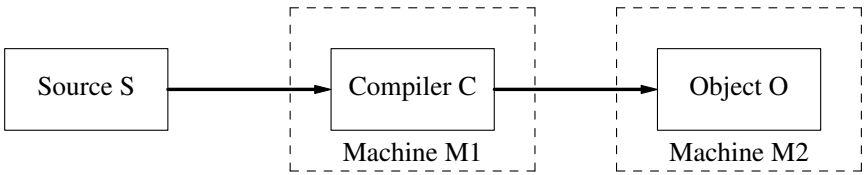


Figure 1.1: Compiler model

A compiler is a program C written in language L running on machine M_1 that translates source language S into 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 glance 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. 343ff). 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.

1.1 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, 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 are what it "does" while the *syntax* of a program is what it "looks like". A compiler typically

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

reads a program having one syntax and generates a program with a different syntax, but the two programs will always have the same semantics.[4]

Preservation of semantics is the single most important design goal of a compiler. The semantics of a source language are mostly specified semi-formally and sometimes formally. Some formal semantics are even suitable for generating a compiler out of 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-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.

1.2 Phases of Compilation

All the phases of compilation are outlined in detail in figure 1.2 (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

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

text that cannot be tokenized[5]

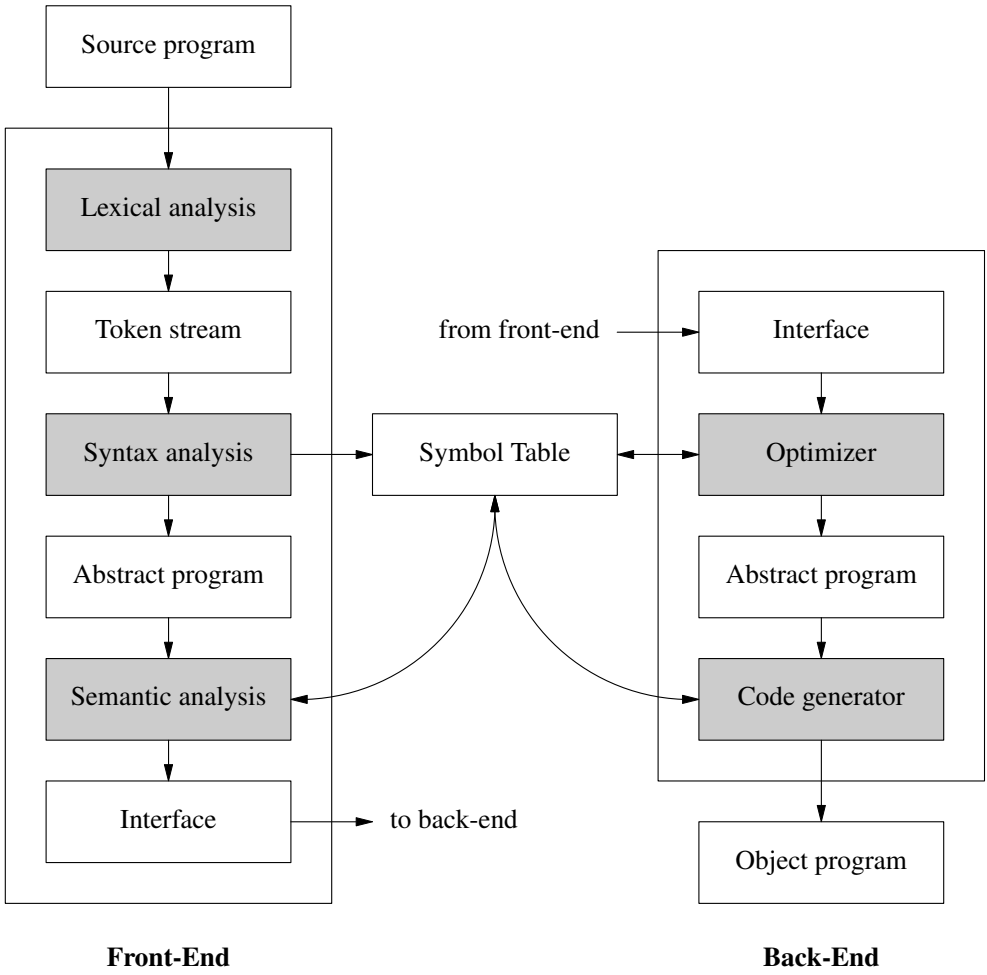


Figure 1.2: Phases of compilation

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

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

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

[5] If such sequences exist in the source language; see pg. 59.

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 the source program up 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
```

Of course this representation loses some information, because all the symbols (like `s` or `putchar`) are simply referred to as `IDENT`s (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 made 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 the 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 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)*.

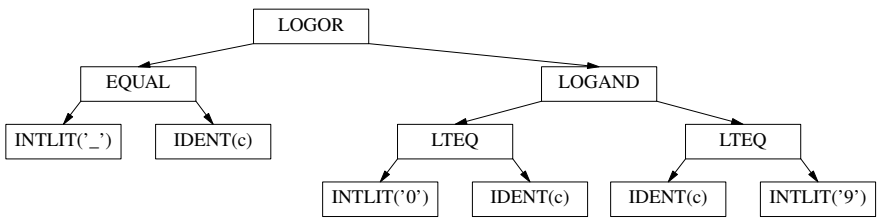


Figure 1.3: An abstract syntax tree (AST)

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 1.3. (EQUAL is 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.[6]

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.

The parser also builds the *symbol tables*. Symbol tables are used to store the names of identifiers and their associated meta data, like addresses, values, types, *arities*[7], 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 without knowing the properties of an identifier in a static language.

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.

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

[7] The number of arguments of a procedure.

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 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 1.1, 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 1.4.

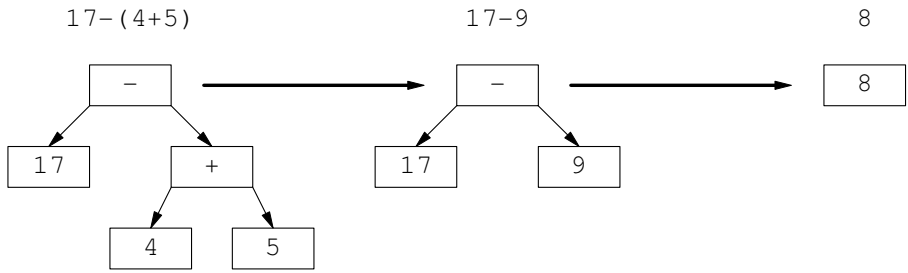


Figure 1.4: Constant folding

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

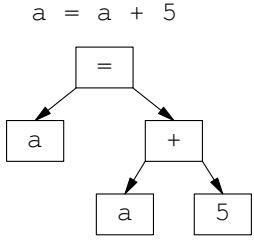
an arbitrary (non-zero) constant to the address of an array would require to know what the code loading the address will look like, so it must be done in the back-end. Even constant folding may require some knowledge about the target architecture, for example for detecting numeric overflow. In the case of C, portable constant folding must be limited to the range $-32768..32767$, 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 naïve 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* an input program P_{in} of a source language S to the most efficient program P_{out} of an object language O that has the same meaning as P_{in}.

1.3 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 1.5.

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 set up in the same phase. No interface to the meta information in the symbol table is needed because the later phases do not make any use of these data.

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. 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 600MHz[8] processor, and it is quite sufficient for a lot of every-day programming tasks.

[8] No, this text is not that ancient; the author is just content with old hardware.

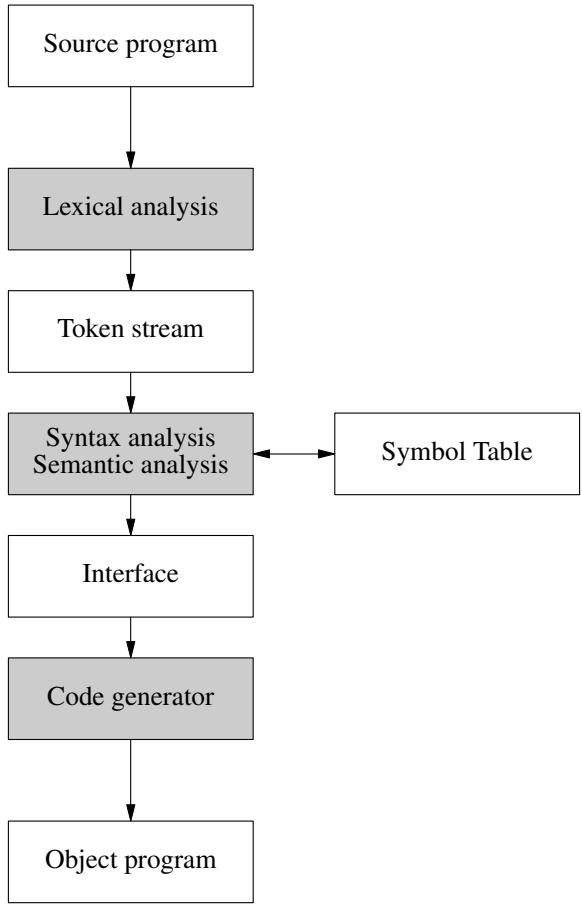


Figure 1.5: A simplified model

Chapter 2

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 not 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[0]
(*a)[0]
*a++
(*a)++
***a
```

```
**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 statements. It does not only have to support all of the above constructs, it also has to do proper pointer addition and

[1] But not the declaration.

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 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*—it allows the implementor to interpret the specs 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.[2] 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

[2] More specifically: before the next "sequence-point", which would be the ; in this case.

programming experience in C, this textbook may reveal some of the more subtle details of the language to you. 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 thew 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]`	first element of a
`*a`	object pointed to by a
`*a[0]`	object pointed to by `a[0]`
`(*a)[0]`	first element of `*a`
`*a++`	return `*a`, then increment a
`(*a)++`	increment `*a` (object pointed to by a)
`*++a`	object pointed to by `++a`
`**a++`	object pointed to by `*a` (then increment a)
`*(*a)++`	increment `*a`, then return object pointed to by result
`(**a)++`	return `**a`, then increment `**a`
`++**a`	increment `**a`, then return `**a`
`*++*a`	increment `*a`, then return `**a`
`**++a`	increment a, then return `**a`

Chapter 3

Rules of the Game

3.1 The Source Language

The SubC language is an almost *strict subset* of the C programming language as defined in TCPL2 (a.k.a. "C89" or "ANSI C"). This means that the vast majority of the programs that SubC accepts will also be accepted by a C89 compiler (and, by extension, by most modern C compilers).

The SubC language contains only two extensions to the ANSI standard, one unavoidable and one due to the laziness of its author. The unavoidable extension is the addition of the `__argc` keyword, which delivers the number of arguments passed to the current function. It is required to implement variable-argument procedures on top of the left-to-right calling conventions of the SubC compiler. Details will be explained in a later part (pg. 260ff).

The other extension allows function declarations to mix parameters with and without explicit type information in the same parameter list, e.g.:

```
int f(char *a, b, c);
```

which would be equal to the ANSI C declaration

```
int f(char *a, int b, int c);
```

Such declarations will not be accepted by any C compiler other than SubC. They have been included because the author uses them occasionally in prototyping.

Except for these two extensions SubC differs from C89 by omission only. The following is a (hopefully!) comprehensive list of restrictions that apply to the SubC language as opposed to C89.

- The following keywords are *not* recognized: `auto`, `const`, `double`, `float`, `goto`, `long`, `register`, `short`, `signed`, `struct`, `typedef`, `union`, `unsigned`, `volatile`.
- There are only two data types: the signed `int` and the unsigned `char`; there are also `void` pointers, and there is limited support for `int(*)()` (pointers to functions of type `int`).
- No more than two levels of indirection are supported, and arrays are limited to one dimension, i.e. valid declarators are limited to `x`, `x[]`, `*x`, `*x[]`, `**x` (and `(*x)()`).
- K&R-style function declarations (with parameter declarations between the parameter list and function body) are not accepted.
- There are no "register", "volatile", or "const" variables. No register allocation takes place, so all variables are implicitly "volatile".
- There is no `typedef`.
- There are no unsigned integers and no long integers.
- There are no `struct`s or `union`s.
- Only `int`s, `char`s and arrays of `int` and `char` can be initialized in their declarations; pointers can be initialized with `0` (but not with `NULL`).
- Local arrays cannot have initializers.
- There are no local `extern`s or `enum`s.
- Local declarations are limited to the beginnings of function bodies (they do not work in other compound statements).
- There are no `static` prototypes.
- Arguments of prototypes must be named.
- There is no `goto`.
- There are no parameterized macros.
- The `#error`, `#if`, `#line`, and `#pragma` preprocessor commands are not recognized.
- The preprocessor does not recognize the "`#`" and `##` operators.
- There may not be any blanks between the "`#`" that introduces a preprocessor command and the subsequent command (e.g.: `# define` would not be recognized as a valid command).
- The `sizeof` operator is limited to types and single identifiers; the operator requires parentheses.
- The address of an array must be specified as `&array[0]` instead of `&array` (but just `array` also works).
- Subscripting an integer with a pointer (e.g. `1["foo"]`) is not supported.
- Function pointers are limited to one single type, `int(*)()`, and they

have no argument types.
- There is no `assert()` due to the lack of parameterized macros.
- The `atexit()` mechanism is limited to one function (this may even be covered by TCPL2).
- Environments of `setjmp()` have to be defined as `int[_JMP_BUFSIZ];` instead of `jmp_buf` due to the lack of `typedef`.
- `FILE` is an alias of `int` due to the lack of `typedef`.
- The `signal()` function returns `int` due to the lack of a more sophisticated type system; the return value must be casted to `int(*)()`.
- Most of the time-related functions are missing due to the lack of `structs`; in particular: `asctime()`, `gmtime()`, `localtime()`, `mktime()`, and `strftime()`.
- The `clock()` function is missing, because `CLOCKS_PER_SEC` varies among systems.
- The `ctime()` function ignores the time zone.

3.2 The Object Language

The SubC compiler generates assembly language for the 386 processor in "AT&T" syntax, which is predominant on Unix systems. For those familiar with the "Intel" syntax, here are the major differences:

- The operands are specified with the *source operand first*, so the instruction `addl %eax,foo` would add the value of `%eax` to `foo`.
- The size of an operand is explicitly specified in the name of an instruction. For instance, `movl` moves a long word (32 bits), `movw` moves a 16-bit word, and `movb` moves a byte.
- Immediate operands are prefixed with a "`$`" sign, e.g. `subl $5,%eax`.
- Parentheses are used for indirection and an optional displacement can be specified in front of the parentheses. For example, `movl (%eax),%eax` loads the value pointed to by `%eax` and `incl 12(%ebp)` increments the value pointed to by `%ebp+12`.
- A star is used for indirect jumps and calls: `call *%eax` calls the routine whose address is stored in `%eax`.

A brief 386 assembly language primer can be found in the appendix (pg. 357ff).

In order to create an executable program, the output of the compiler has to be assembled and linked against the SubC runtime library. The assembler and linker are not discussed in this book and they are not part

of the source code presented here. The SubC compiler controller invokes the system assembler and system linker in order to generate an executable program.

3.3 The Runtime Library

The traditional C runtime environment consists of two parts: the C runtime startup module `crt0` and a collection of library functions widely known as `libc`. SubC uses this very approach, it just renames `libc` to `libscc`, because it also uses a few routines of the C library of the system compiler.

The startup module `crt0` is the only part of the compiler that is written in assembly language. It comprises an interface to the C library of the system compiler which in turn provides an interface to the system calls of the operating system. The primary tasks of the runtime startup module are the initialization of internal library structures and the translation between SubC's and C's calling conventions. The dependency on the system's C library could be broken by implementing the system calls directly in `crt0`, but the author has decided to aim for portability instead.

The standard C library (as defined in TCPL2) for SubC is itself written in SubC. Except for references to system calls it consists of portable C89 code. Like the compiler, it differs from the full implementation only by omission. It also suffers from slow performance due to the lack of parameterized macros (e.g. `getc`, `putc`), but instead of tweaking for performance, correctness was considered to be more important due to the widespread use of these functions. Individual design decisions will be discussed in the sections dealing with the C library (pg. 233ff).

Part II

The Tour

Chapter 4

Definitions and Data Structures

4.1 Definitions

The compiler source code begins with some global definitions in the `defs.h` file. This file describes the data structures used in the compiler, specifies some static limits, paths to components, etc. The file is included by all modules of the code, so we also include the required library headers here.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Definitions
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

`PREFIX` is a character that will be prepended to *all* symbols generated by the compiler, e.g. `main` will be exported as `Cmain`. This is done for two reasons:

(1) It distinguishes the names of identifiers defined in a source program from identifiers generated by the compiler (jump labels, etc);
(2) It distinguishes the names of identifiers compiled by SubC from those in the C library of the underlying platform, e.g. the `_exit()` function of the SubC library will be exported as `C_exit()`, so no conflict can arise

at link time when `C_exit()` calls the `_exit()` function of the system library.

The resulting symbol name is called an *external name*, because this is what the linker will see in the case of public symbols.

Likewise, the `LPREFIX` constant specifies the prefix for labels generated by the compiler. All compiler-created labels will consist of this prefix with a small integer attached, e.g. `L123`.

```
#define PREFIX      'C'
#define LPREFIX     'L'
```

The following definitions specify various paths to components of both the SubC compiler and the C compiler of the host platform. `ASCMD` is the command that will be used to assemble the output of the compiler, where the first `%s` is replaced by the name of the output file and the second one with the name of the input file. The `LDCMD` command is used to invoke the system linker. Again the first `%s` is replaced by the output file name. The second one is substituted with the path to the SubC base directory (`SCCDIR`). The `SCCLIBC` and `SYSLIBC` strings will be appended to the linker invocation with spaces in between, eventually giving:

`ld -o` *output-file* *SCCDIR*`/lib/crt0.o ...` *SCCLIBC* *SYSLIBC*

```
#ifndef SCCDIR
 #define SCCDIR     "."
#endif

#define ASCMD       "as -o %s %s"
#define LDCMD       "ld -o %s %s/lib/crt0.o"
#define SCCLIBC     "%s/lib/libscc.a"
#define SYSLIBC     "/usr/lib/libc.a"
```

`INTSIZE`, `PTRSIZE`, and `CHARSIZE` specify the sizes in bytes of an `int`, a generic pointer, and a `char`, respectively. `CHAROFF` is the offset of a `char` in an `int`, but this constant it not actually used by the 386 back-end. It may be necessary when porting SubC to non-little-endian architectures, though.

The SubC compiler requires that `INTSIZE`=`PTRSIZE`, because it does not distinguish between pointers and integers internally, but stores everything in registers that provide enough space for either type.

```
#define INTSIZE     4
#define PTRSIZE     INTSIZE
#define CHARSIZE    1
#define CHAROFF     0
```

TEXTLEN and NAMELEN are rather arbitrary values specifying the maximum length of an individual token and the number of relevant characters in an identifier. TEXTLEN is a hard limit; tokens that exceed this length will cause the compiler to bail out instantly. Identifiers may be longer than NAMELEN, but only NAMELEN characters will be saved in the symbol table (including the PREFIX part!).

```
#define TEXTLEN     512
#define NAMELEN     16
```

MAXFILES is the maximum number of files that can be passed to the SubC compiler controller on the command line for compilation *and* linking, i.e. `scc *.c` may not pass more than MAXFILES files to the compiler. However, no such limit exists for `scc -c *.c`, because in this case the compiler does not have to memorize the file names for linking.

```
#define MAXFILES    32
```

The following definitions specify various limits regarding language constructs. MAXIFDEF is the maximum number of *nested* #ifdefs and #ifndefs. This is a total limit that spans any included headers.

MAXNMAC is the maximum number of nested *macro* expansions. Each time a macro is expanded (possibly while already expanding a macro), one level of nesting is added. In other words, this is the number of possible *forward references* allowed in macro expansion.[1] With MAXNMAC=2 and

```
#define FOO BAR
#define BAR BAZ
#define BAZ 1
```

(defined in this order), the expansion of BAR and BAZ would succeed, but the expansion of FOO would fail, because it would require three levels of nested expansion.

[1] More precisely, MAXNMAC is one less than the number of possible forward references, so MAXNMAC = 2 will allow for 1 forward reference.

`MAXCASE` is the maximum number of `case`s per `switch`, and `MAXBREAK` is the maximum number of nested `break` and `continue` contexts (i.e. loops and `switch` statements).

`MAXLOCINIT` is the maximum number of *initializers* per local context, and `MAXFNARGS` is the maximum number of *formal arguments* per function.

These values are rather arbitrary and can be increased safely without suffering any side effects except for larger memory footprint.

```
#define MAXIFDEF    16
#define MAXNMAC     32
#define MAXCASE     256
#define MAXBREAK    16
#define MAXLOCINIT  32
#define MAXFNARGS   32
```

`NSYMBOLS` specifies the maximum number of global and local symbols that can be placed in the *symbol table* at the same time. Local symbols will be removed at the end of their contexts, but global symbols will accumulate. Function declarations (prototypes) also take up space in the global part of the symbol table.

`POOLSIZE` is the space reserved for *identifier* names and *function signatures* (see section B.4, pg. 352).

```
#define NSYMBOLS    1024
#define POOLSIZE    8192
```

The SubC compiler separates *type information* into two parts, a "meta type" and a "primitive type". The following are *meta types*: *arrays*, *functions*, *constants*, *variables*, and *macros*. Variables, arrays, and functions have additional primitive types, which are defined after the meta types.

```
#define TVARIABLE   1
#define TARRAY      2
#define TFUNCTION   3
#define TCONSTANT   4
#define TMACRO      5
```

These are the *primitive types* supported by the SubC compiler:

```
#define PCHAR   1       /* char      */
#define PINT    2       /* int       */
#define CHARPTR 3       /* char *    */
#define INTPTR  4       /* int *     */
#define CHARPP  5       /* char **   */
#define INTPP   6       /* int **    */
#define PVOID   7       /* void      */
#define VOIDPTR 8       /* void *    */
#define VOIDPP  9       /* void **   */
#define FUNPTR  10      /* int (*)() */
```

The following are the storage classes supported by the compiler. `CPUBLIC` is the default for all global identifiers. Symbols of this class will be exported and made visible to other modules.

`CEXTERN` is used for identifiers declared with explicit `extern` keywords as well as for implicit function declarations and function prototypes.

`CSTATIC` is used for identifiers declared with a `static` keyword at the *top level* (in the global context). `CLSTATC` is used for static identifiers in local contexts. Neither of them will be exported.

`CAUTO` is the storage class of automatically allocated (non-static) local identifiers.

```
#define CPUBLIC 1
#define CEXTERN 2
#define CSTATIC 3
#define CLSTATC 4
#define CAUTO   5
```

The `LV` (lvalue) structure is a central data structure of the SubC compiler. Because the language has no `struct`s, though, it is defined as a set of symbolic array subscripts plus the number of elements in the structure, so an `LV` structure is defined with

```
int lv[LV];
```

and its members are accessed using `lv[LVSYM]` and `lv[LVPRIM]`. The `LV` structure will be explained in detail in the section on expression parsing (9.2, pg. 87ff).

```
#define LVSYM   0
#define LVPRIM  1
#define LV      2
```

The following are *debug options* that can be activated on the command line. `D_LSYM` lists local symbol tables on `stdout`, `D_GSYM` lists the global symbol table, and `D_STAT` prints some usage data at the end of the compilation process.

```
#define D_LSYM  1
#define D_GSYM  2
#define D_STAT  4
```

These are the *tokens* generated by the scanner. Symbols in the last block, containing tokens with `P_` prefixes, represent preprocessor commands. The ordering of the first block is in fact important, because it reflects the *precedence* of the corresponding operators.

```
enum {
    SLASH, STAR, MOD, PLUS, MINUS, LSHIFT, RSHIFT,
    GREATER, GTEQ, LESS, LTEQ, EQUAL, NOTEQ, AMPER,
    CARET, PIPE, LOGAND, LOGOR,

    __ARGC, ASAND, ASXOR, ASLSHIFT, ASMINUS, ASMOD, ASOR,
    ASPLUS, ASRSHIFT, ASDIV, ASMUL, ASSIGN, BREAK, CASE,
    CHAR, COLON, COMMA, CONTINUE, DECR, DEFAULT, DO,
    ELLIPSIS, ELSE, ENUM, EXTERN, FOR, IDENT, IF, INCR,
    INT, INTLIT, LBRACE, LBRACK, LPAREN, NOT, QMARK,
    RBRACE, RBRACK, RETURN, RPAREN, SEMI, SIZEOF, STATIC,
    STRLIT, SWITCH, TILDE, VOID, WHILE, XEOF, XMARK,

    P_INCLUDE, P_DEFINE, P_ENDIF, P_ELSE, P_ELSENOT,
    P_IFDEF, P_IFNDEF, P_UNDEF
};
```

4.2 Global Data

The `data.h` file is also included by all modules of the compiler. It defines (or declares) all global data objects. Only the main module `#defines _extern`

before including this file, so it is the only module that reads this file with `_extern` set to nothing, resulting in variable definitions instead of `extern` declarations.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Global variables
 */

#ifndef _extern
 #define _extern extern
#endif
```

`Infile` is the current input file from which source code is being read; it changes when including header files. `Outfile` is the file to which object code is written. When it is `NULL`, no code will be emitted.

```
_extern FILE    *Infile;
_extern FILE    *Outfile;
```

`Token` is the *token* most recently returned by the scanner, `Text` is its textual form, i.e. a string containing the characters comprising the token, and `Value` is its value, if any. For integers `Value` is set to the value of the integer, for characters it is the *ASCII* code of the character, for strings and identifiers, it is their length. For all other tokens the value is undefined.

```
_extern int     Token;
_extern char    Text[TEXTLEN+1];
_extern int     Value;
```

The following variables deal with error handling. `Line` is the current input line number (which is reset to 1 when including headers), `Errors` is the error counter, and `Syntoken` is the next *scanner synchronization* token. When the compiler detects an error that causes it to lose synchronization, it will set `Syntoken` to a token that is very likely to follow at some point in the input stream (like a semicolon). It will not report any further errors until this token is found. This method reduces the number of redundant and/or bogus error messages generated by the parser.

```
_extern int     Line;
_extern int     Errors;
_extern int     Syntoken;
```

`Putback` is a character that has been "put back" into the input stream. This is basically identical to the `ungetc()` mechanism, but works on a more explicit level. A `Putback` of -1 indicates that no character has been put back since the last read.

`Rejected`, `Rejval`, and `Rejtext` are to Tokens what `Putback` is to characters. They are used to put an already-scanned token back to the token stream. This allows the parser to *look ahead* at the next Token in the stream without consuming it. There is only one point in the whole SubC grammar where this is actually necessary (see pg. 102). A `Rejected` value of -1 means that no token is currently in the *reject queue*.

```
_extern int     Putback;
_extern int     Rejected;
_extern int     Rejval;
_extern char    Rejtext[TEXTLEN+1];
```

`File` is the current input file name. It changes to the name of a header file when including it. `Basefile` is the name of the principal file being compiled; it does *not* change when including a header.

```
_extern char    *File;
_extern char    *Basefile;
```

`Macp[]` and `Macc[]` implement a stack of structures for implementing macro expansion. `Mp` is the stack pointer which points at the first unused slot; when it is zero, the stack is empty.

Each element of `Macp[]` holds a pointer to a macro text. When `Mp` is non-zero, the scanner reads `Macp[Mp-1]` instead of the current input file. `Macc[]` holds the first character that follows the name of a macro in the input stream. It is used to clear the "put back" queue when macro expansion starts. It will be used to restore the queue when macro expansion ends.

The `Expandmac` flag can be set to zero to turn off macro expansion entirely. This is necessary when scanning macro names in `#define`, for example, because otherwise

```
#define FOO 1
#define FOO 1
```

would expand to

```
#define FOO 1
#define 1   1
```

before the preprocessor would see the command.

```
_extern char    *Macp[MAXNMAC];
_extern int     Macc[MAXNMAC];
_extern int     Mp;
_extern int     Expandmac;
```

`Ifdefstk[]` and `Isp` implement the `#ifdef` stack. It keeps track of the current state of the scanner. When the top of the `#ifdef` stack indicates an unsatisfied `#ifdef`, the scanner will not deliver any tokens before a matching `#else` or `#endif` is found.

`Inclev` is the number of nested `#include`s currently in effect.

```
_extern int     Ifdefstk[MAXIFDEF], Isp;
_extern int     Inclev;
```

When `Textseg` is non-zero, the emitter emits to the *text* (code) segment, otherwise it emits to the *data segment*.

```
_extern int     Textseg;
```

The following set of arrays implements the *symbol table*:

`Names`	Pointers to symbol names in the name pool.
`Prims`	Primitive types (`int`, `char`, `void*`, etc).
`Types`	Meta types (array, function, macro, etc).
`Stcls`	Storage classes (public, extern, static, auto, etc).
`Sizes`	Sizes of arrays, arities of functions.
`Vals`	Values of constants (`enum`s), addresses of locals.
`Mtext`	Macro texts and function signatures.

```
_extern char    *Names[NSYMBOLS];
_extern char    Prims [NSYMBOLS];
_extern char    Types [NSYMBOLS];
_extern char    Stcls [NSYMBOLS];
_extern int     Sizes [NSYMBOLS];
_extern int     Vals  [NSYMBOLS];
_extern char    *Mtext[NSYMBOLS];
```

The SubC compiler uses a shared symbol table for global and local identifiers. Global identifiers are inserted at the bottom of the table and `Globs` points to the first free global slot. Local symbols are inserted at the top of the table and `Locs` points to the first local symbol in the table (or to the end of table when no local symbols are present). When `Globs>=Locs`, the symbol table is full.

```
_extern int     Globs;
_extern int     Locs;
```

`Thisfn` points to the symbol table entry of the function currently being defined.

```
_extern int     Thisfn;
```

`Nlist` is the *name list* of the compiler. All identifier names are stored here. Like in the symbol table itself, global names are inserted at the bottom and local names at the top, so `Nbot` points to the free space between the segments, and `Ntop` points to the name of the most recently inserted local symbol.

```
_extern char    Nlist[POOLSIZE];
_extern int     Nbot;
_extern int     Ntop;
```

`Breakstk[]` and `Bsp` implement the *break stack* and `Contstk[]` and `Csp` the *continue stack*. They keep track of the current `break` and `continue` contexts, respectively. The values stored in these stacks are the labels to which `break` and `continue` statements in the various contexts will branch. An empty stack signals the compiler that a branch statement occurs out of context.

`Retlab` is the label to which a `return` statement will branch.

```
_extern int     Breakstk[MAXBREAK], Bsp;
_extern int     Contstk[MAXBREAK], Csp;
_extern int     Retlab;
```

`LIaddr[]` and `LIval[]` are used to *initialize* local variables. They are filled while parsing local declarations. At the beginning of a function body the code generator will emit code to move each `LIval` (value) to the corresponding `LIaddr` (address). `Nli` is the number of local initializers found.

```
_extern int     LIaddr[MAXLOCINIT];
_extern int     LIval[MAXLOCINIT];
_extern int     Nli;
```

`Files[]` and `Nf` (number of files) are used to store the names of source files for later invocation of the linker:

```
_extern char    *Files[MAXFILES];
_extern int     Nf;
```

These are option flags and values that can be passed to the compiler controller on the command line.

Variable	Option	Description
`O_verbose`	`-v`	Verbose operation (`-vv` = more).
`O_componly`	`-c`	Compile only, do not link.
`O_asmonly`	`-S`	Generate assembly output only.
`O_testonly`	`-t`	Test only, generate no output.
`O_outfile`	`-o file`	The name of the output file.
`O_debug`	`-d opt`	Debug flags.

```
_extern int     O_verbose;
_extern int     O_componly;
_extern int     O_asmonly;
_extern int     O_testonly;
_extern char    *O_outfile;
_extern int     O_debug;
```

4.3 Function Prototypes

The `decl.h` file lists the prototypes of all functions that are used outside of their modules. It is included for the sake of completeness; feel free to skip ahead.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Function declarations
 */

int     addglob(char *name, int prim, int ttype, int scls,
```

```
                int size, int val, char *mval, int init);
int     addloc(char *name, int prim, int type, int scls,
               int size, int val, int init);
void    cerror(char *s, int c);
int     chrpos(char *s, int c);
void    clear(void);
void    clrlocs(void);
void    colon(void);
void    compound(int lbr);
int     constexpr(void);
void    copyname(char *name, char *s);
int     declarator(int arg, int scls, char *name, int *pprim,
                   int *psize, int *pval, int *pinit);
void    dumpsyms(char *title, char *sub, int from, int to);
int     eofcheck(void);
void    error(char *s, char *a);
int     expr(int *lv);
void    fatal(char *s);
int     findglob(char *s);
int     findloc(char *s);
int     findmac(char *s);
int     frozen(int depth);
void    gen(char *s);
int     genadd(int p1, int p2);
void    genaddr(int y);
void    genand(void);
void    genargc(void);
void    genasop(int op, int p1, int p2);
int     genbinop(int op, int p1, int p2);
void    genbool(void);
void    genbrfalse(int dest);
void    genbrtrue(int dest);
void    genbss(char *name, int len);
void    gencall(int y);
void    gencalr(void);
void    gencmp(char *inst);
void    gendata(void);
void    gendefb(int v);
void    gendefl(int id);
void    gendefp(int v);
void    gendefs(char *s, int len);
void    gendefw(int v);
void    gendiv(int swap);
```

```
void    genentry(void);
void    genexit(void);
void    geninc(int *lv, int inc, int pre);
void    genind(int p);
void    genior(void);
void    genjump(int dest);
void    genlab(int id);
void    genldlab(int id);
void    genlit(int v);
void    genln(char *s);
void    genlocinit(void);
void    genlognot(void);
void    genmod(int swap);
void    genmul(void);
void    genname(char *name);
void    genneg(void);
void    gennot(void);
void    genpostlude(void);
void    genprelude(void);
void    genpublic(char *name);
void    genpush(void);
void    genpushlit(int n);
void    genraw(char *s);
void    genscale(void);
void    genscale2(void);
void    genshl(int swap);
void    genshr(int swap);
void    genstack(int n);
void    genstore(int op, int *lv, int *lv2);
int     gensub(int p1, int p2, int swap);
void    genswitch(int *vals, int *labs, int nc, int dflt);
void    gentext(void);
void    genxor(void);
char    *globname(char *s);
char    *gsym(char *s);
void    ident(void);
int     inttype(int p);
int     label(void);
char    *labname(int id);
void    lbrace(void);
void    lgen(char *s, char *inst, int n);
void    lgen2(char *s, int v1, int v2);
void    load(void);
```

```
void    lparen(void);
void    match(int t, char *what);
char    *newfilename(char *name, int sfx);
int     next(void);
void    ngen(char *s, char *inst, int n);
void    ngen2(char *s, char *inst, int n, int a);
int     objsize(int prim, int type, int size);
void    playmac(char *s);
int     pointerto(int prim);
void    preproc(void);
int     primtype(int t);
void    putback(int t);
void    rbrack(void);
void    reject(void);
int     rexpr(void);
void    rparen(void);
void    rvalue(int *lv);
int     scan(void);
int     scanraw(void);
void    semi(void);
void    sgen(char *s, char *inst, char *s2);
int     skip(void);
int     synch(int syn);
void    top(void);
int     typematch(int p1, int p2);
```

Chapter 5

Utility Functions

The file `misc.c` contains some very general functions as well as some *parsing* and semantic analysis routines that do not belong to any specific part of the parser.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Miscellanea
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

The `chrpos()` function returns the offset of the first occurrence of c in s or -1 when c does not occur in s.

```
int chrpos(char *s, int c) {
    char    *p;

    p = strchr(s, c);
    return p? p-s: -1;
}
```

The `copyname()` function copies the name of an identifier from s to *name*. It copies no more than `NAMELEN` characters and attaches a delimiting NUL character to *name*.

```
void copyname(char *name, char *s) {
    strncpy(name, s, NAMELEN);
```

```
    name[NAMELEN] = 0;
}
```

The `newfilename()` function creates a new file name that is equal to the *file* parameter passed to it except for the suffix, which is changed to *sfx*. All suffixes are single characters. This function is used to create output file names, e.g. `"foo.c"` → `"foo.o"`. It returns the new name.

```
char *newfilename(char *file, int sfx) {
    char    *ofile;

    ofile = strdup(file);
    ofile[strlen(ofile)-1] = sfx;
    return ofile;
}
```

The `match()` function matches the token *t*: when the current token is equal to *t*, it consumes it by reading another token. When a token cannot be matched, the function prints an error message explaining what it expected. The *what* argument is a textual description of what was expected. Some examples follow below.

```
void match(int t, char *what) {
    if (Token == t) {
        Token = scan();
    }
    else {
        error("%s expected", what);
    }
}
```

The following functions use `match()` to make sure that the next object in the token stream is a specific token. They are just abbreviations of the corresponding `match()` calls for some particularly frequently-used tokens.

```
void lparen(void) {
    match(LPAREN, "'('");
}

void rparen(void) {
```

```
    match(RPAREN, "')'");
}

void lbrace(void) {
    match(LBRACE, "'{'");
}

void rbrack(void) {
    match(RBRACK, "']'");
}

void semi(void) {
    match(SEMI, "';'");
}

void colon(void) {
    match(COLON, "':'");
}

void ident(void) {
    match(IDENT, "identifier");
}
```

The `eofcheck()` function is used to check for a premature *end of file* (*EOF*) inside of blocks delimited by curly braces. A missing closing brace is an error that often causes the parser to lose its synchronization for good, so the EOF is reached during error recovery. So the parser calls this function in brace-delimited blocks to make sure it can bail out when the EOF is reached. `eofcheck()` returns 1 when the EOF was reached and otherwise 0.

```
int eofcheck(void) {
    if (XEOF == Token) {
        error("missing '}'", NULL);
        return 1;
    }
    return 0;
}
```

The `inttype()` function returns 1 when p is an integer type (`int` or `char`) and otherwise 0.

```
int inttype(int p) {
    return PINT == p || PCHAR == p;
}
```

Chapter 6

Error Handling

The file `error.c` contains routines for reporting errors and resynchronizing the parser.

```
/*
 *  NMH's Simple C Compiler, 2011
 *  Error handling
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

In case of an error the `cleanup()` function removes the assembly source code and object code files generated by the compiler and assembler so far. Note that this is a necessity rather than a convenience, because leaving an old object file in the file system after aborting compilation may result in an obsolete module being linked into a program, which has the potential to cause a lot of confusion.

```
static void cleanup(void) {
    if (!O_testonly && NULL != Basefile) {
        remove(newfilename(Basefile, 's'));
        remove(newfilename(Basefile, 'o'));
    }
}
```

The `error()` function is the principal *error reporting* function of the SubC compiler, so it is called in quite a few places. It prints an error message of the form

`error:` *File*`:` *Line*`:` *message*

where the message is passed to `error()` in the *s* argument. The argument may contain one single format specifier of the form `%s`, which will be replaced by the argument *a* (when *s* does not contain a format specifier, *a* should be `NULL`).

When catching the first error, the function removes any partial or obsolete output files. No errors are reported while waiting for a synchronization token. When too many errors have been reported, a fatal error will be signaled. In this case the error counter has to be reset first in order to avoid indefinite recursion.

```
void error(char *s, char *a) {
    if (Syntoken) return;
    if (!Errors) cleanup();
    fprintf(stderr, "error: %s: %d: ", File, Line);
    fprintf(stderr, s, a);
    fprintf(stderr, "\n");
    if (++Errors > 10) {
        Errors = 0;
        fatal("too many errors");
    }
}
```

The `fatal()` function reports an error and then aborts compilation instantly. It is only called when there is no hope for recovery, e.g. when the symbol table is full or one of the internal stacks overflowed.

```
void fatal(char *s) {
    error(s, NULL);
    error("fatal error, stop", NULL);
    exit(EXIT_FAILURE);
}
```

The `cerror()` function is used to report input characters that are not in the input alphabet of the compiler (e.g. `'$'` or `BEL` (ASCII code 7)). It reports unprintable characters using their hexa-decimal ASCII codes and printable characters using both their ASCII code and their printable form.

```
void cerror(char *s, int c) {
    char    buf[32];
```

```
    if (isprint(c))
        sprintf(buf, "'%c' (\\x%x)", c, c);
    else
        sprintf(buf, "\\x%x", c);
    error(s, buf);
}
```

The `synch()` function is the other half of the resynchronization mechanism of the parser, the first half being the one that suppresses error messages in `error()` when the `Syntoken` variable is set to a token value.

The `synch()` function skips input tokens until it finds one that matches the *syn* parameter. It then sets `Syntoken` to *syn*. At this point the current token in the stream is the one at which resynchronization will take place. However, the parser may call `match()` unsuccessfully a few times until it "catches up" by also reaching the point where it eventually matches `Syntoken`. This is why error messages will be suppressed until `Syntoken` is matched.

```
int synch(int syn) {
    int t;

    t = scan();
    while (t != syn) {
        if (EOF == t)
            fatal("found EOF in error recovery");
        t = next();
    }
    Syntoken = syn;
    return t;
}
```

Chapter 7

Lexical Analysis

The `scan.h` file contains the *scanner*. Even here, there is quite a bit of hidden complexity. While the set of tokens of the C language has a rather moderate size, many of its tokens come in many forms (like decimal, octal, and hexa-decimal integers) or contain components that require more elaborate scanning (like the escape sequences in string and character literals).

The scanner also invokes the preprocessor, which is not a separate program, but built into the SubC compiler.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Lexical analysis (scanner)
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

We start with single characters. The `next()` function extracts the next character from the input stream. It first tries the *put-back queue*. When it contains a character, it clears the queue and returns that character.

When the queue is empty, it tries the *macro expansion* stack. When macro expansion is in progress, it returns the next character from the current expansion buffer except when it is a NUL character. A NUL character indicates the end of the buffer. When reaching the end of a buffer, `next()` closes the buffer (by decrementing `Mp`) and then returns the put-back (`Macc[]`) character of that buffer. This character is the first input character following the name of the macro that has just been expanded.

Only when both of these sources are empty, `next()` fetches the next character from the current input file. The function also increases the *line counter* (`Line`) when it finds a newline character.

```
int next(void) {
    int c;

    if (Putback) {
        c = Putback;
        Putback = 0;
        return c;
    }
    if (Mp) {
        if ('\0' == *Macp[Mp-1]) {
            Macp[Mp-1] = NULL;
            return Macc[--Mp];
        }
        else {
            return *Macp[Mp-1]++;
        }
    }
    c = fgetc(Infile);
    if ('\n' == c) Line++;
    return c;
}
```

Putting a character in the put-back queue is easy.

```
void putback(int c) {
    Putback = c;
}
```

The `hexchar()` function returns the value of the hexa-decimal literal that follows in the input stream. It is used to decode `\x` *escape sequences*.

```
static int hexchar(void) {
    int     c, h, n = 0, f = 0;

    while (isxdigit(c = next())) {
        h = chrpos("0123456789abcdef", tolower(c));
        n = n * 16 + h;
```

```
        f = 1;
    }
    putback(c);
    if (!f)
        error("missing digits after '\\x'", NULL);
    if (n > 255)
        error("value out of range after '\\x'", NULL);
    return n;
}
```

The `scanch()` function extracts the next character of a string or character literal. When the character is a backslash ("\"), it decodes the escape sequence introduced by it. It returns the ASCII code of the resulting character, except for the sequence `\"`. For this sequence it returns `'"'|256`, so that the `scanstr()` function will not interpret it as the end of its input. However, when the string scanner puts this value into an array of `char`, the 9'th bit (set by OR'ing in 256) will be stripped, resulting in `'"'`.

This trick is indeed portable, because C converts between the `int` and `char` types silently.

```
static int scanch(void) {
    int c, c2;

    c = next();
    if ('\\' == c) {
        switch (c = next()) {
        case 'a': return '\a';
        case 'b': return '\b';
        case 'f': return '\f';
        case 'n': return '\n';
        case 'r': return '\r';
        case 't': return '\t';
        case 'v': return '\v';
        case '\\': return '\\';
        case '"': return '"' | 256;
        case '\'': return '\'';
        case '0': case '1': case '2':
        case '3': case '4': case '5':
        case '6': case '7':
            for (c2 = 0; isdigit(c) && c < '8'; c = next())
                c2 = c2 * 8 + (c - '0');
            putback(c);
```

```
            return c2;
        case 'x':
            return hexchar();
        default:
            cerror("unknown escape sequence: %s", c);
            return ' ';
        }
    }
    else {
        return c;
    }
}
```

The `scanint()` function scans decimal, octal, and hexa-decimal integer literals and returns their values. It determines the radix of a literal by examining its prefix ("`0`" for octal or `0x` for hexa-decimal). After scanning the literal it puts back the first non-numeric character so that it will be scanned again when extracting the next token. This is necessary, because two tokens may appear in direct succession in the input stream, like in `1-x`; in this case the "`-`" would be swallowed by `scanint()` if it was not put back to the stream for later digestion.

Note that `scanint()` does not recognize signs—this will be done by the parser. In C, the number -123 is in fact a "`-`" operator followed by the literal `123`. Try it! Just place a blank between the leading "`-`" and the digits of `123` and feed it to your favorite C compiler!

```
static int scanint(int c) {
    int val, radix, k, i = 0;

    val = 0;
    radix = 10;
    if ('0' == c) {
        Text[i++] = '0';
        if ((c = next()) == 'x') {
            radix = 16;
            Text[i++] = c;
            c = next();
        }
        else {
            radix = 8;
        }
    }
```

```
    while ((k = chrpos("0123456789abcdef", tolower(c))) >= 0)
    {
        Text[i++] = c;
        if (k >= radix)
            cerror("invalid digit in integer literal: %s",
                    c);
        val = val * radix + k;
        c = next();
    }
    putback(c);
    Text[i] = 0;
    return val;
}
```

The `scanstr()` function scans a string literal. It does not even try to recover when finding a string literal that is longer than `TEXTLEN` characters, because such a long string normally indicates a missing `'"'` character—a situation with virtually no hope for recovery.

```
static int scanstr(char *buf) {
    int i, c;

    buf[0] = '"';
    for (i=1; i<TEXTLEN-2; i++) {
        if ((c = scanch()) == '"') {
            buf[i++] = '"';
            buf[i] = 0;
            return Value = i;
        }
        buf[i] = c;
    }
    fatal("string literal too long");
    return 0;
}
```

The `scanident()` function scans an identifier. It reports identifiers with an absurd length of more than `TEXTLEN` characters. Like `scanint()` it puts back the first character not matching its domain.

We also could report identifiers that are longer than `NAMELEN`-1 at this point, but we won't. What would life be without all the little surprises?

```
static int scanident(int c, char *buf, int lim) {
```

```
    int i = 0;

    while (isalpha(c) || isdigit(c) || '_' == c) {
        if (lim-1 == i) {
            error("identifier too long", NULL);
            i++;
        }
        else if (i < lim-1) {
            buf[i++] = c;
        }
        c = next();
    }
    putback(c);
    buf[i] = 0;
    return i;
}
```

The `skip()` function skips over white space and *comments* in the input program. Note that `/*...*/` comments may *not* be nested in C89. The function detects the end of a comment by using a single-element queue (p) to check the current and previous input character. As soon as it finds $p =$`'*'` and $c =$`'/'`, it assumes that the end of the current comment has been reached. `skip()` returns the first non-white and non-comment character found in the input.

The `skip()` function also invokes the *preprocessor* when it finds a "`#`" character after a newline. Preprocessor commands are parsed and processed completely *before* `skip()` returns, so the scanner actually never "sees" the preprocessor commands. This is necessary, because C is actually two languages in one and has been designed to be processed by the preprocessor first and then by the compiler.

SubC does not employ a separate preprocessor, though. When the scanner finds a preprocessor command, it suspends compilation, runs the built-in preprocessor on a single command, and then resumes scanning. A preprocessor command is any command that is introduced by a newline (or the beginning of the file) followed by any number of blanks and a "`#`" character. Preprocessor commands are delimited by the end of the line and may occur *anywhere* in a program. Even the following program is valid C:

```
int
#define SIZE 25
    x[SIZE];
```

It illustrates nicely why compilation has to be suspended while the preprocessor is run.

```
int skip(void) {
    int c, p, nl;

    c = next();
    nl = 0;
    for (;;) {
        if (EOF == c) {
            strcpy(Text, "<EOF>");
            return EOF;
        }
        while (' ' == c || '\t' == c || '\n' == c ||
            '\r' == c || '\f' == c
        ) {
            if ('\n' == c) nl = 1;
            c = next();
        }
        if (nl && c == '#') {
            preproc();
            c = next();
            continue;
        }
        nl = 0;
        if (c != '/')
            break;
        if ((c = next()) != '*') {
            putback(c);
            c = '/';
            break;
        }
        p = 0;
        while ((c = next()) != EOF) {
            if ('/' == c && '*' == p) {
                c = next();
                break;
            }
            p = c;
        }
    }
    return c;
}
```

The `keyword()` function checks whether its parameter s is a SubC keyword. It reduces the number of necessary `strcmp()` operations by comparing s only to keywords that have the same first character as s. In the case of preprocessor commands, it even takes the second character into consideration.

The function returns the token representing the keyword s or zero when s is not a keyword.

```
static int keyword(char *s) {
    switch (*s) {
    case '#':
        switch (s[1]) {
        case 'd':
            if (!strcmp(s, "#define")) return P_DEFINE;
            break;
        case 'e':
            if (!strcmp(s, "#else")) return P_ELSE;
            if (!strcmp(s, "#endif")) return P_ENDIF;
            break;
        case 'i':
            if (!strcmp(s, "#ifdef")) return P_IFDEF;
            if (!strcmp(s, "#ifndef")) return P_IFNDEF;
            if (!strcmp(s, "#include")) return P_INCLUDE;
            break;
        case 'u':
            if (!strcmp(s, "#undef")) return P_UNDEF;
            break;
        }
        break;
    case 'b':
        if (!strcmp(s, "break")) return BREAK;
        break;
    case 'c':
        if (!strcmp(s, "case")) return CASE;
        if (!strcmp(s, "char")) return CHAR;
        if (!strcmp(s, "continue")) return CONTINUE;
        break;
    case 'd':
        if (!strcmp(s, "default")) return DEFAULT;
        if (!strcmp(s, "do")) return DO;
        break;
    case 'e':
        if (!strcmp(s, "else")) return ELSE;
```

```
        if (!strcmp(s, "enum")) return ENUM;
        if (!strcmp(s, "extern")) return EXTERN;
        break;
    case 'f':
        if (!strcmp(s, "for")) return FOR;
        break;
    case 'i':
        if (!strcmp(s, "if")) return IF;
        if (!strcmp(s, "int")) return INT;
        break;
    case 'r':
        if (!strcmp(s, "return")) return RETURN;
        break;
    case 's':
        if (!strcmp(s, "sizeof")) return SIZEOF;
        if (!strcmp(s, "static")) return STATIC;
        if (!strcmp(s, "switch")) return SWITCH;
        break;
    case 'v':
        if (!strcmp(s, "void")) return VOID;
        break;
    case 'w':
        if (!strcmp(s, "while")) return WHILE;
        break;
    case '_':
        if (!strcmp(s, "__argc")) return __ARGC;
        break;
    }
    return 0;
}
```

The `macro()` function checks whether *name* exists and refers to a macro. When it refers to a macro, it starts expanding ("playing") that macro and returns 1. Otherwise it just returns 0.

```
static int macro(char *name) {
    int y;

    y = findmac(name);
    if (!y || Types[y] != TMACRO)
        return 0;
    playmac(Mtext[y]);
```

```
    return 1;
}
```

7.1 A Brief Theory of Scanning

There are two kinds of *tokens*: tokens representing individual objects like `INCR` (the `++` operator) or `CONTINUE` (the `continue` keyword) and tokens representing classes of objects (like `INTLIT` or `STRLIT`, which represent the classes of all integer literals and string literals, respectively).

The task of the *scanner* is to find the next token in the input stream, isolate it, and return a small integer representing that token. The small integer is also commonly referred to as a "token". When the scanner extracts a "class token", it also stores the *attributes* of that token for later use. Such attributes include, for example, the value of an integer literal, the characters of a string literal, the name of an identifier, etc.

The first step performed by the scanner is to skip over "noise characters"—characters that do not have any meaning to the parser. This class of characters includes blanks, line separators, characters of comments, and—in the case of C—preprocessor commands.

After discarding noise characters, the scanner has to find out what comes next in the input stream. The first step in this process is to examine the first non-noise character found in the stream. When this character is unambiguous, the scanner can just return its token. For example, when it finds a parentheses or a comma, it can just consume that character and return the corresponding token. This works because there are no tokens that begin with unambiguous characters except for the tokens represented by these single characters alone. So this method works fine for all single-character tokens, such as:

```
( ) [ ] { } , ; : ? ~
```

All of SubC's *token classes* can also be identified by their first characters. For instance, when we find a decimal digit, we know that an integer literal will follow in the input. When we find a "`"`" character, we know that we have found a string literal, and when we find a letter or an underscore, we know we are dealing with an identifier. In all of these cases the scanner can just delegate the collection of characters to a specialized function, such as `scanident()` or `scanstr()`. The function will then collect the characters of the class token in the `Text` variable and its value (if any) in the `Value`

variable. The scanner returns the class token. Examples for class tokens would be:

```
"hello, world!"    (STRLIT)
0xdeadbeef         (INTLIT)
strcmp             (IDENT)
'\007'             (INTLIT)
```

The only remaining case is that of an *ambiguous* input character. Because C has a rather rich set of operators, the majority of all *operator* tokens begin with an ambiguous character. For instance, the "<" character may introduce any of the following tokens:

```
<   <<   <=   <<=
```

When the scanner encounters an ambiguous character, it consumes it, because it knows that it must introduce a token of a limited set. For example, "<" must be the first character of a token of the set $\{<, <<, <=, <<=\}$. The scanner then proceeds by examining the next input character, thereby narrowing down the set until the ambiguity is resolved.

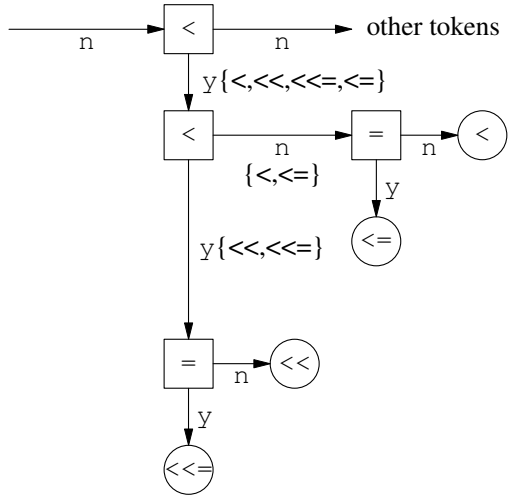


Figure 7.1: Scanning ambiguous input

This process is outlined in figure 7.1. Each square node denotes a comparison, each round node a returned token. Each "y" branch means that the character compared positive and has been consumed, each "n" branch indicates a mismatch (the character will not be consumed).

For instance, when scanning the input "`<x`", the `<` character would compare positive in the topmost node, making the scanner follow the "y" branch. At this point the set of possible tokens would be $\{<, <<, <<=, <=\}$. The next comparison would fail due to $x \neq$ `'<'`, so the "n" branch would be taken, reducing the set to $\{<, <=\}$. The next comparison would also fail due to $x \neq$ `'='`, so the only choice left would be "`<`".

Note that this technique implements *longest match scanning*. When confronted with two options, the scanner will always attempt to extract the longest possible token, so the expression `a+++b` would be decomposed into

```
a ++ + b
```

If `a+ ++b` is what you want, you have to place a blank between the operators. Scanning ends automatically when a leaf in the decision tree is reached, so tokens need not be separated by blanks. The expression `i-->0` scans as

```
i -- > 0
```

because there is no such token as `-->`.

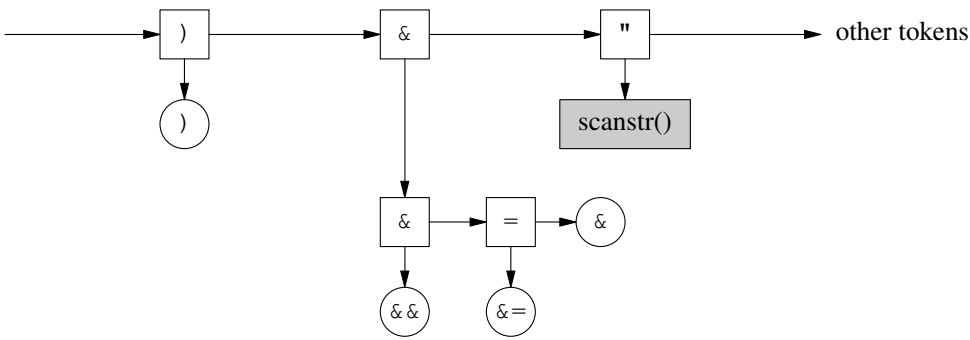


Figure 7.2: A scan tree

Also note that the scan tree can easily be expanded to cover all ambiguous tokens or even *all* tokens of a language, as indicated in figure 7.2. In this figure the "y" and "n" labels have been omitted; down arrows indicate accepting a character, right arrows indicate rejecting it. The top line of the graph checks the current input character against the first character of each possible subset. The set of the complete tree is the set of all tokens of the entire language. When reaching the end of the top line, the current input

character would not be in the *input alphabet* (the set of characters accepted by) the source language. A gray box in figure 7.2 indicates delegation to a specialized scanner function.

The `scanpp()` ("scan and preprocess") function implements the scan tree of the SubC scanner. Scan trees can easily be hand-coded as a sequence of nested `if` statements, but for reasons of efficiency and readability, the "spine" of the tree has been mapped to a single large `switch` statement.

Before `scanpp()` actually attempts to scan some input, it first checks the token *reject queue*. When there is a token in the queue, it clears the queue and returns this token together with its attributes. Otherwise it proceeds to skip over noise characters and then starts scanning.

Most of the function is a straight-forward implementation of the SubC scan tree, but there are a few interesting points:

Character literals are classified as `INTLIT`, because SubC makes no difference between `int`s and `char`s internally (it does distinguish arrays of and pointers to these types, though).

When the scanner finds the name of a *macro* in its input, it will not return `IDENT` but the first token of the macro text instead. To do so, the scanner has to invoke itself to start scanning the macro text. Because recursion tends to be inefficient in C, the main body of the scanner is packaged in an infinite loop (`for (;;)`), so recursion can be initiated by doing a `break` in the `switch` body of the scan tree. When a token has been scanned successfully, the scanner exits via `return`, so the endless loop does not hurt.

Finally: there is one point where the scanner may reach a dead end. This happens when a dot (".") has been found, but no subsequent dots follow. In SubC, the only valid token starting with a dot is the ellipsis (...), so an error has to be reported for single dots and double dots.[1]

```
static int scanpp(void) {
    int     i, c, t;

    if (Rejected != -1) {
        t = Rejected;
        Rejected = -1;
        strcpy(Text, Rejtext);
        Value = Rejval;
        return t;
```

[1] In the full C language, "." and ... would both be valid tokens, but .. would still be an error.

```
    }
    for (;;) {
        Value = 0;
        c = skip();
        memset(Text, 0, 4);
        Text[0] = c;
        switch (c) {
        case '!':
            if ((c = next()) == '=') {
                Text[1] = '=';
                return NOTEQ;
            }
            else {
                putback(c);
                return XMARK;
            }
        case '%':
            if ((c = next()) == '=') {
                Text[1] = '=';
                return ASMOD;
            }
            else {
                putback(c);
                return MOD;
            }
        case '&':
            if ((c = next()) == '&') {
                Text[1] = '&';
                return LOGAND;
            }
            else if ('=' == c) {
                Text[1] = '=';
                return ASAND;
            }
            else {
                putback(c);
                return AMPER;
            }
        case '(':
            return LPAREN;
        case ')':
            return RPAREN;
        case '*':
```

```
            if ((c = next()) == '=') {
                Text[1] = '=';
                return ASMUL;
            }
            else {
                putback(c);
                return STAR;
            }
        case '+':
            if ((c = next()) == '+') {
                Text[1] = '+';
                return INCR;
            }
            else if ('=' == c) {
                Text[1] = '=';
                return ASPLUS;
            }
            else {
                putback(c);
                return PLUS;
            }
        case ',':
            return COMMA;
        case '-':
            if ((c = next()) == '-') {
                Text[1] = '-';
                return DECR;
            }
            else if ('=' == c) {
                Text[1] = '=';
                return ASMINUS;
            }
            else {
                putback(c);
                return MINUS;
            }
        case '/':
            if ((c = next()) == '=') {
                Text[1] = '=';
                return ASDIV;
            }
            else {
                putback(c);
```

```
            return SLASH;
        }
    case ':':
        return COLON;
    case ';':
        return SEMI;
    case '<':
        if ((c = next()) == '<') {
            Text[1] = '<';
            if ((c = next()) == '=') {
                Text[2] = '=';
                return ASLSHIFT;
            }
            else {
                putback(c);
                return LSHIFT;
            }
        }
        else if ('=' == c) {
            Text[1] = '=';
            return LTEQ;
        }
        else {
            putback(c);
            return LESS;
        }
    case '=':
        if ((c = next()) == '=') {
            Text[1] = '=';
            return EQUAL;
        }
        else {
            putback(c);
            return ASSIGN;
        }
    case '>':
        if ((c = next()) == '>') {
            Text[1] = '>';
            if ((c = next()) == '=') {
                Text[1] = '=';
                return ASRSHIFT;
            }
            else {
```

```
                putback(c);
                return RSHIFT;
            }
        }
        else if ('=' == c) {
            Text[1] = '=';
            return GTEQ;
        }
        else {
            putback(c);
            return GREATER;
        }
    case '?':
        return QMARK;
    case '[':
        return LBRACK;
    case ']':
        return RBRACK;
    case '^':
        if ((c = next()) == '=') {
            Text[1] = '=';
            return ASXOR;
        }
        else {
            putback(c);
            return CARET;
        }
    case '{':
        return LBRACE;
    case '|':
        if ((c = next()) == '|') {
            Text[1] = '|';
            return LOGOR;
        }
        else if ('=' == c) {
            Text[1] = '=';
            return ASOR;
        }
        else {
            putback(c);
            return PIPE;
        }
    case '}':
```

```
        return RBRACE;
    case '~':
        return TILDE;
    case EOF:
        strcpy(Text, "<EOF>");
        return XEOF;
    case '\'':
        Text[1] = Value = scanch();
        if ((c = next()) != '\'')
            error(
             "expected '\\'' at end of char literal",
                NULL);
        Text[2] = '\'';
        return INTLIT;
    case '"':
        Value = scanstr(Text);
        return STRLIT;
    case '#':
        Text[0] = '#';
        scanident(next(), &Text[1], TEXTLEN-1);
        if ((t = keyword(Text)) != 0)
            return t;
        error("unknown preprocessor command: %s", Text);
        return IDENT;
    case '.':
        for (i=1; i<3; i++)
            Text[i] = next();
        Text[i] = 0;
        if (strcmp(Text, "..."))
            error("malformed ellipsis: '%s'", Text);
        return ELLIPSIS;
    default:
        if (isdigit(c)) {
            Value = scanint(c);
            return INTLIT;
        }
        else if (isalpha(c) || '_' == c) {
            Value = scanident(c, Text, TEXTLEN);
            if (Expandmac && macro(Text))
                break;
            if ((t = keyword(Text)) != 0)
                return t;
            return IDENT;
```

```
            }
            else {
                cerror("funny input character: %s", c);
                break;
            }
        }
    }
}
```

The `scan()` function is the principal interface of the SubC scanner. It scans tokens using `scanpp()` and discards them as long as the current `#ifdef` context is "frozen"—i.e. as long as the condition of the current `#ifdef` context is not satisfied. The condition inside of the `do` loop makes sure that encountering the EOF inside of the loop will abort compilation *unless* the EOF was found while including a header file. (The `Inclev` variable keeps track of the number of pending `#includes`.) Finally, `scan()` resets `Syntoken` when matching it.

```
int scan(void) {
    int t;

    do {
        t = scanpp();
        if (!Inclev && Isp && XEOF == t)
            fatal("missing #endif at EOF");
    } while (frozen(1));
    if (t == Syntoken)
        Syntoken = 0;
    return t;
}
```

The `scanraw()` function is like `scan()`, but does not expand macros and ignores the current `#ifdef` context (by temporarily setting `Isp` to zero). This function is only used by the preprocessor, which sometimes has to invoke the scanner recursively.

```
int scanraw(void) {
    int t, oisp;

    oisp = Isp;
    Isp = 0;
```

```
    Expandmac = 0;
    t = scan();
    Expandmac = 1;
    Isp = oisp;
    return t;
}
```

The `reject()` function puts the current token and all its attributes into the *reject queue*, so that the next call to `scan()` will produce it again.

```
void reject(void) {
    Rejected = Token;
    Rejval = Value;
    strcpy(Rejtext, Text);
}
```

Chapter 8

Symbol Table Management

The file `sym.c` contains functions that look up *identifiers* and add them to the *symbol table*. Most of the functions exist in two flavors, one for *local symbols* and one for *global* ones.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Symbol table management
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

The `findglob()`, `findloc()`, and `findmac()` functions find the symbol table entries of global and local symbols and (also global) macros, respectively. They all return the slot of the symbol table entry upon success and zero in case the symbol could not be found. All of these functions just perform a linear search, nothing fancy. This is quite sufficient for compiling small programs. To speed things up just a little bit, though, a comparison of the first characters is performed to save a call to `strcmp()` in case they do not match.

```
int findglob(char *s) {
    int i;

    for (i=0; i<Globs; i++)
        if (Types[i] != TMACRO &&
            *s == *Names[i] && !strcmp(s, Names[i])
```

```
        )
            return i;
    return 0;
}

int findloc(char *s) {
    int i;

    for (i=Locs; i<NSYMBOLS; i++) {
        if (*s == *Names[i] && !strcmp(s, Names[i]))
            return i;
    }
    return 0;
}

int findmac(char *s) {
    int i;

    for (i=0; i<Globs; i++)
        if (TMACRO == Types[i] &&
            *s == *Names[i] && !strcmp(s, Names[i])
        )
            return i;
    return 0;
}
```

The `newglob()` and `newloc()` functions allocate a new global or local symbol table entry and return its slot number. Both of them abort compilation in case of a symbol table overflow.

```
int newglob(void) {
    int p;

    if ((p = Globs++) >= Locs)
        fatal("symbol table overflow");
    return p;
}

int newloc(void) {
    int p;

    if ((p = --Locs) <= Globs)
```

```
        fatal("symbol table overflow");
    return p;
}
```

The `globname()` and `locname()` functions insert strings into the global and local segments of the *name list*. They return a pointer to the inserted name upon success and abort compilation in case of a name list overflow.

```
char *globname(char *s) {
    int p, k;

    k = strlen(s) + 1;
    if (Nbot + k >= Ntop)
        fatal("name list overflow");
    p = Nbot;
    Nbot += k;
    strcpy(&Nlist[p], s);
    return &Nlist[p];
}

char *locname(char *s) {
    int p, k;

    k = strlen(s) + 1;
    if (Nbot + k >= Ntop)
        fatal("name list overflow");
    Ntop -= k;
    p = Ntop;
    strcpy(&Nlist[p], s);
    return &Nlist[p];
}
```

The `defglob()` function defines data objects associated with global symbols. It takes a lot of arguments that describe the object to define:

name	The symbol identifying the object.
prim	The primary type of the object.
type	The meta type of the object.
size	The size (number of elements) of the object.
val	The initial value of the object.
scls	The storage class of the object.
init	A flag indicating that the object has an initial value.

For *constants* and *functions* there is nothing to do. Constants exist only at compile time and need not be defined as data objects, and code for functions will be emitted elsewhere. Initialized arrays are defined while parsing their initializers, so in this case there is nothing to do, either.

When the storage class *scls* is `CPUBLIC`, then an object language statement will be emitted to announce that the identifier is public (a `.globl` directive in AT&T 386 assembly language).

When the object is an uninitialized array, the compiler will just allocate space for it in the BSS[1] segment of the executable. Otherwise it will generate a byte, word, or pointer definition, respectively. The storage locations of *atomic* (non-array) variables will be initialized with *val*. All atomic variables will be initialized; when no explicit initial value is specified, zero is used.

Functions beginning with a `gen` prefix are code generator functions. For example, `gendata()` indicates that all subsequent code will describe data and not code, `gendefb()` defines a byte of data, and `genbss()` describes a block in the BSS segment. The code generator functions will be covered in detail later in this book (pg. 159ff).

```
static void defglob(char *name, int prim, int type, int size,
                    int val, int scls, int init)
{
    if (TCONSTANT == type || TFUNCTION == type) return;
    gendata();
    if (CPUBLIC == scls) genpublic(name);
    if (init && TARRAY == type)
        return;
    if (TARRAY != type) genname(name);
    if (PCHAR == prim) {
        if (TARRAY == type)
            genbss(gsym(name), size);
        else
            gendefb(val);
    }
    else if (PINT == prim) {
        if (TARRAY == type)
            genbss(gsym(name), size*INTSIZE);
        else
            gendefw(val);
    }
```

[1] Block Started by Symbol

```
    else {
        if (TARRAY == type)
            genbss(gsym(name), size*PTRSIZE);
        else
            gendefp(val);
    }
}
```

The `addglob()` function is the interface for adding global symbols to the symbol table. Its takes the same parameters as `defglob()` with the addition of *mtext*, which contains the values of macros or the signatures of functions.

`addglob()` does not accept redefinitions except when the existing symbol has the `CEXTERN` storage class and the new symbol is also `CEXTERN` or `CPUBLIC`. Redefinitions must supply the same primitive type as the existing symbol. No new symbol is added when redefining an "extern" symbol, only the storage class of the existing entry is changed.

When the symbol being added to the table has a `CPUBLIC` or `CSTATIC` storage class, then the function will call `defglob()` to emit the proper definition.

```
int addglob(char *name, int prim, int type, int scls,
            int size, int val, char *mtext, int init)
{
    int y;

    if ((y = findglob(name)) != 0) {
        if (Stcls[y] != CEXTERN)
            error("redefinition of: %s", name);
        else if (CSTATIC == scls)
            error("extern symbol redeclared static: %s",
                    name);
        if (TFUNCTION == Types[y])
            mtext = Mtext[y];
    }
    if (y == 0) {
        y = newglob();
        Names[y] = globname(name);
    }
    else if (TFUNCTION == Types[y] || TMACRO == Types[y]) {
        if (Prims[y] != prim || Types[y] != type)
            error(
```

```
                "redefinition does not match prior type: %s",
                name);
    }
    if (CPUBLIC == scls || CSTATIC == scls)
        defglob(name, prim, type, size, val, scls, init);
    Prims[y] = prim;
    Types[y] = type;
    Stcls[y] = scls;
    Sizes[y] = size;
    Vals[y] = val;
    Mtext[y] = mtext;
    return y;
}
```

The `defloc()` function defines data objects associated with *local static* identifiers. No definitions are ever emitted for local "auto" (non-static) identifiers, because these are allocated on the stack at run time.

Local static symbols (with the storage class `CLSTATC`) really identify global objects with a local scope: their identifiers are only visible in local contexts, but their storage is located in the data segment of a program rather than on the stack. In order to avoid name collisions, though, they cannot be referenced by their identifiers internally. So instead of using their name, a unique label is generated from the *val* parameter, and this label is used to reference the local static object internally. This way the same local static identifier can exist in different contexts. For instance, the program

```
f() { static x = 3; x; }
g() { static x = 4; x; }
```

could generate the following code:

```
        .data
L1:     .long   3           # x of f()
        .text
Cf:     pushl   %ebp
        movl    %esp,%ebp
        movl    L1,%eax     # reference to x in f()
L2:     popl    %ebp
        ret
        .data
L3:     .long   4           # x of g()
```

```
        .text
Cg:     pushl   %ebp
        movl    %esp,%ebp
        movl    L3,%eax     # reference to x of g()
L4:     popl    %ebp
        ret
```

Because the *val* parameter is used for the label ID, the initial values of atomic objects (3 and 4 in the above example) are passed to `defloc()` in the *init* parameter.

```
static void defloc(int prim, int type, int size, int val,
                   int init)
{
    gendata();
    if (type != TARRAY) genlab(val);
    if (PCHAR == prim) {
        if (TARRAY == type)
            genbss(labname(val), size);
        else
            gendefb(init);
    }
    else if (PINT == prim) {
        if (TARRAY == type)
            genbss(labname(val), size*INTSIZE);
        else
            gendefw(init);
    }
    else {
        if (TARRAY == type)
            genbss(labname(val), size*PTRSIZE);
        else
            gendefp(init);
    }
}
```

The `addloc()` function adds a new local symbol to the symbol table. Its parameters are similar to those of `addglob()` but, like `defloc()`, it expects initializer values in *init* rather than *val*. Local symbols can never be redefined.

```
int addloc(char *name, int prim, int type, int scls,
           int size, int val, int init)
```

```
{
    int y;

    if (findloc(name))
        error("redefinition of: %s", name);
    y = newloc();
    if (CLSTATC == scls) defloc(prim, type, size, val, init);
    Names[y] = locname(name);
    Prims[y] = prim;
    Types[y] = type;
    Stcls[y] = scls;
    Sizes[y] = size;
    Vals[y] = val;
    return y;
}
```

The `clrlocs()` function clears the local symbol table and removes all local names from the name list.

```
void clrlocs(void) {
    Ntop = POOLSIZE;
    Locs = NSYMBOLS;
}
```

The `objsize()` function returns the size of a data object of the primitive type *prim*. When *type* equals `TARRAY`, it multiplies that value with *size*. It returns an invalid size (zero) for functions and macros.

This function is used to compute the sizes of local objects and it is also used by the `sizeof` operator.

```
int objsize(int prim, int type, int size) {
    int k = 0;

    if (PINT == prim) {
        k = INTSIZE;
    }
    else if (PCHAR == prim) {
        k = CHARSIZE;
    }
    else if (INTPTR == prim || CHARPTR == prim ||
             VOIDPTR == prim
```

```
    ) {
        k = PTRSIZE;
    }
    else if (INTPP == prim || CHARPP == prim ||
              VOIDPP == prim
    ) {
        k = PTRSIZE;
    }
    else if (FUNPTR == prim) {
        k = PTRSIZE;
    }
    if (TFUNCTION == type || TCONSTANT == type ||
        TMACRO == type
    ) {
        return 0;
    }
    if (TARRAY == type)
        k *= size;
    return k;
}
```

The `typename()` function returns a string describing the given primitive type.

```
static char *typename(int p) {
    return  PINT    == p? "INT":
        PCHAR   == p? "CHAR":
        INTPTR  == p? "INT*":
        CHARPTR == p? "CHAR*":
        VOIDPTR == p? "VOID*":
        FUNPTR  == p? "FUN*":
        INTPP   == p? "INT**":
        CHARPP  == p? "CHAR**":
        VOIDPP  == p? "VOID**":
        PVOID   == p? "VOID": "n/a";
}
```

The `dumpsyms()` function is used to print symbol tables to `stdout` when the `gsym` or `lsym` *debug options* are activated. The *title* and *sub* parameters are used to generate the headline of the output. The *from* and *to* parameters specify the range of slots to print (e.g.: $0..Globs$ would print the global table, and $Locs..NSYMBOLS-1$ would print the current local table).

```
void dumpsyms(char *title, char *sub, int from, int to) {
    int i;
    char    *p;

    printf("\n===== %s%s =====\n", title, sub);
    printf(
      "PRIM    TYPE  STCLS   SIZE  VALUE  NAME (MVAL)\n"
      "------  ----  -----  -----  -----  -----------\n");
    for (i = from; i < to; i++) {
        printf("%-6s  %s  %s  %5d  %5d  %s",
            typename(Prims[i]),
            TVARIABLE == Types[i]? "VAR ":
                TARRAY == Types[i]? "ARRY":
                TFUNCTION == Types[i]? "FUN ":
                TCONSTANT == Types[i]? "CNST":
                TMACRO == Types[i]? "MAC ": "n/a ",
            CPUBLIC == Stcls[i]? "PUBLC":
                CEXTERN == Stcls[i]? "EXTRN":
                CSTATIC == Stcls[i]? "STATC":
                CLSTATC == Stcls[i]? "LSTAT":
                CAUTO == Stcls[i]? "AUTO ": "n/a  ",
            Sizes[i],
            Vals[i],
            Names[i]);
        if (TMACRO == Types[i])
            printf(" [\"%s\"]", Mtext[i]);
        if (TFUNCTION == Types[i]) {
            printf(" (");
            for (p = Mtext[i]; *p; p++) {
                printf("%s", typename(*p));
                if (p[1]) printf(", ");
            }
            putchar(')');
        }
        putchar('\n');
    }
}
```

Chapter 9

Syntax and Semantic Analysis

Syntax analysis and semantic analysis make up the largest part of the compiler, so they are spread out across multiple files. There is also a bit of theory to cover in this chapter.

The `prec.h` file is included by the expression parser and the constant expression parser. It describes the *precedence* of most of the binary operators of the C language. Higher values in the `Prec[]` array denote higher precedence. The order of values *must* match the order of the corresponding token definitions in the `enum` in `defs.h` (pg. 30).

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Binary operator precedence
 */

static int Prec[] = {
    7, /* SLASH */
    7, /* STAR */
    7, /* MOD */
    6, /* PLUS */
    6, /* MINUS */
    5, /* LSHIFT */
    5, /* RSHIFT */
    4, /* GREATER */
    4, /* GTEQ */
    4, /* LESS */
    4, /* LTEQ */
```

```
    3, /* EQUAL */
    3, /* NOTEQ */
    2, /* AMPER */
    1, /* CARET */
    0, /* PIPE */
};
```

9.1 A Brief Theory of Parsing

Parsing is in fact quite similar to scanning, it only takes place at a more abstract level. For example, when the scanner encounters the input character "+", we know that the set of possible operators that may follow in the input stream would be $\{+, ++, +=\}$. We could formalize this fact as follows:

```
<plus_set>  :=  + | ++ | +=
```

which would be pronounced as "a *plus_set* is defined as either + or ++ or +=", where the := acts as the "defined-as" operator and the "|" as the logical "or". Names of rules are enclosed in angle brackets to distinguish them from operators of the grammar description and tokens of the described language.

In fact what we have here is exactly a *rule* (also called a *production*) of a *formal grammar*. The grammar of the entire C language consists of rules like these:

```
<sum_op> := PLUS       /* A sum operator is + or - */
          | MINUS

<term_op> := STAR      /* A term operator is * or / or % */
           | SLASH
           | MOD
```

Note that a logical "or" in a formal grammar actually separates *two individual rules*. The *term_op* rules above could also be written as

```
<term_op> := STAR      /* A term operator is * or / or % */
<term_op> := SLASH
<term_op> := MOD
```

When multiple rules have the same name, they do not contradict each other but extend each other. So when the parser attempts to match a token against a *term_op*, it has three different choices.

There is another major difference between decision trees of scanners and formal grammars: in a decision tree, each comparison node refers to a character, but in a grammar not all identifiers on the right-hand side of a rule must refer to a token. Consider the following grammar:

```
<binary_number> := <binary_digit>
                 | <binary_digit> <binary_number>

<binary_digit> := 0 | 1
```

The first two rules say that a *binary_number* is defined as a *binary_digit* or a *binary_digit* followed by another *binary_number*. The next two rules define a *binary_digit* as either 0 or 1.

So rules may refer to *tokens* or to other rules—or even to themselves. A token is also called a *terminal symbol* (or just a "terminal") of a formal grammar, and the name of a rule is called a *non-terminal symbol* (or just a "non-terminal"). A terminal symbol is a symbol that cannot be decomposed any further—it must either be matched or rejected. A non-terminal, though, typically offers a choice. For example, when the "0" cannot be matched in *binary_digit*, the parser may still try the "1". The process of parsing is basically equal to finding a "path" through a grammar. When such a path exists, a *sentence* has been "matched" or "accepted". When no such path exists, the current input is not a sentence that can be formed by the grammar of the source language. This causes what programmers commonly refer to as a "syntax error".

A rule is matched by descending into the grammar until a terminal is found and then matching the current input token against that terminal. When the terminal does not match, the next possible terminal is tried until either a match is found or the parser runs out of rules. When no more rules can be tried, a syntax error is reported. Here is an illustration of the process of matching the input `1 0 1` against the *binary_number* rule:

```
1 0 1  :=  binary_digit( 1 )  binary_number( 0 1 )
  0 1  :=  binary_digit( 0 )  binary_number( 1 )
    1  :=  binary_digit( 1 )
```

In each step one digit is matched and matching the remaining digits is delegated to another instance of *binary_number*. This is a recursive process

similar to that of recursive procedures calling each other. It is illustrated in figure 9.1.

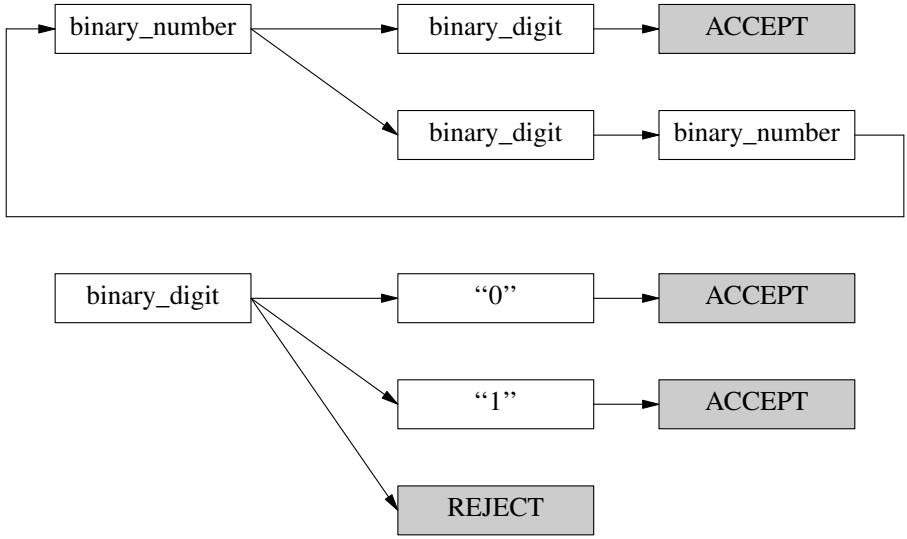


Figure 9.1: The graph of the *binary_number* grammar

To match a *binary_number*, we have to find a path from the *binary_number* node to the ACCEPT node in the same graph. No matter which path we take, the first node we encounter is a *binary_digit* node, so we have to remember where we are and descend into the *binary_digit* rules. Here we have to reach an ACCEPT node of the *binary_digit* graph, which can be done by consuming either a "0" or a "1" token. When none of these rules match, we end up at the REJECT node, indicating that what we are trying to parse is not a *binary_digit*—and therefore not a *binary_number*, because matching *binary_number* requires to match *binary_digit* first.

When we reach an ACCEPT node in *binary_digit*, we go back to the *binary_digit* node of *binary_number*. The next action depends on the path taken initially in *binary_number*. When we have chosen the first production, we would just accept the single digit found and be happy, even if more binary digits actually follow in the input. If we had chosen the second production, though, and no more digits would follow, we would be in trouble, because we had to reject a valid binary number. So in one case we may match the input only partially and in the other case we may reject valid input, which is both not acceptable. We can solve this problem by rewriting the graph slightly as depicted in figure 9.2.

In this graph we have only one choice initially, so we try to accept a *binary_digit*. After accepting it, we have to decide whether we want to keep

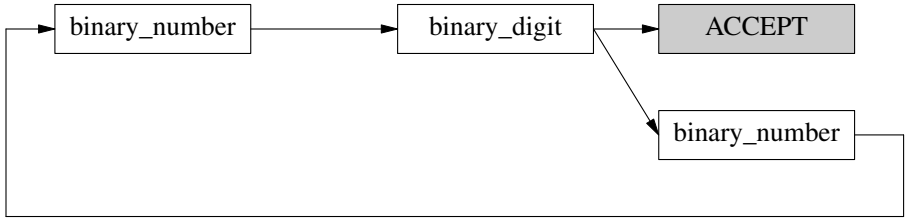


Figure 9.2: An alternative graph of *binary_number*

collecting digits or just accept the input matched so far. This decision is trivial at this point, because the next input token is either a binary digit or not. When it is one, we keep parsing and otherwise we just accept the matched sentence and bail out.

The transformation of the parse graph is valid, because both rules of the original graph started with identical nodes. As long as multiple productions start with identical nodes, we can just combine them in a single path and defer any decisions to the point where the rules start to differ.

In order to decide whether we keep parsing or accept a sentence, we have to know the set of initial tokens that is accepted by a given rule. This set is called the *FIRST set* of a rule. The FIRST set of *binary_digit* is $\{0, 1\}$, and because all productions of *binary_number* begin with *binary_digit*, the FIRST set of *binary_number* is the same.

However, not all FIRST sets are so simple. In general the FIRST set of a rule R is computed as follows:

- Begin with FIRST={}.
- Add all terminals that appear in first positions of right-hand sides of R to FIRST.
- Add the FIRST sets of all non-terminals that appear in first positions of right-hand sides of R to FIRST.

Computing the FIRST sets of non-terminals in first positions of R is done by applying the above algorithm recursively. Here is a more elaborate example:

```
FIRST = {+,-}
<number> := + <digit_list>
          | - <digit_list>
```

```
FIRST = {0,1,2,3,4,5,6,7,8,9,0}
<digit_list> := <digit>
              | <digit> <digit_list>

FIRST = {0,1,2,3,4,5,6,7,8,9,0}
<digit> := <odd>
         | <even>

FIRST = {1,3,5,7,9}
<odd>  := 1 | 3 | 5 | 7 | 9

FIRST = {0,2,4,6,8}
<even> := 0 | 2 | 4 | 6 | 8
```

FIRST(*digit_list*) (the FIRST set of *digit_list*) is equal to FIRST(*digit*), because *digit_list* itself has no terminals in first positions. The FIRST set of *digit* is the union of the FIRST sets of *odd* and *even*, because both of them appear in first positions. The *digits* rules have no terminals, either. To decide whether or not to keep parsing in *digit_list*, the parser checks whether the current input token is in FIRST(*digit_list*).

FIRST(*number*) is just $\{+, -\}$, because these are the only terminals in first positions of *number* and there are no non-terminals in first positions. This means that *number* would only accept numbers with an explicit sign.

When hand-coding a parser, as the one that we will study in the following, the FIRST sets are normally small and easy to derive.

9.1.1 Mapping Grammars to Parsers

There are various different ways to specify grammars, but all of them follow the same basic pattern, which was depicted in the previous section. The notation used to write down formal grammars is widely known as "*Backus Normal Form*"[1] (*BNF*). There are lots of variations of this form. Some of them use angle brackets around non-terminals, others use lower-case names for non-terminals and all-caps names for terminals, some specify terminals as string literals. Here are a few examples:

```
<term> := <number> '*' <number>   /* traditional */
        | <number> '/' <number>
        | <number> '%' <number>
```

[1] Or "Backus-Naur Form"

```
term := number STAR number          /* YACC */
      | number SLASH number
      | number MOD number
      ;

term := number * number               /* simplified */
      | number / number
      | number MOD number
```

The first version is the form traditionally used in many formal language specifications. The second one conforms to the input language of the popular YACC *compiler-compiler*.[2] The third form is a simplified form that uses lowercase names for non-terminals and all-caps and punctuation-only sequences for terminals. We will use the latter in the remainder of this chapter.

Let us now have a look at a more elaborate grammar, one that describes a subset of real C expressions. Here it comes:

```
factor :=
       INTLIT
     | IDENT

term :=
       factor
     | term * factor
     | term / factor
     | term % factor

sum :=
       term
     | sum + term
     | sum - term

shift :=
       sum
     | shift << sum
     | shift >> sum
```

[2] A compiler-compiler or "meta-compiler" is a program that generates a parser from a formal grammar.

```
relation :=
       shift
     | relation < shift
     | relation > shift
     | relation <= shift
     | relation >= shift

equation :=
       relation
     | equation == relation
     | equation != relation

binand :=
       equation
     | binand & equation

binxor :=
       binand
     | binxor ^ binand

binor :=
       binxor
     | binor '|' binxor
```

There are a few fine points to observe in this grammar. The first thing, which is rather obvious, is that there are many rules of the form

```
term  :=  factor  |  term / factor
```

instead of

```
term  :=  factor  |  factor / term
```

This order reflects the actual *associativity* of the corresponding operators. In the former rule the "/" operator associates to the left (so that $a/b/c$ equals $(a/b)/c$) while in the latter rule it associates to the right (so $a/b/c = a/(b/c)$). Of course we want left-associativity, because this is what C does in these cases.

However, left-associativity implies *left-recursion*, which poses another problem: we now have to match *term* first in order to match *term*, which

smells like indefinite recursion. We can get both, right-recursion and left-associativity, though, by rewriting the grammar slightly:

```
term :=
       factor
     | factor term_ops

term_ops :=
       / factor
     | / factor term_ops
```

This grammar always matches a *factor* first and only descends into *term_ops* if the current token is in FIRST(*term_ops*), which is just the "/" operator in this example. The *term_ops* rule recurses as long the current token is in FIRST(*term_ops*).

We could rewrite the entire expression grammar to make right-recursion explicit, but we will just keep in mind that this transformation *can* be performed and then proceed to answer a more practical question: how is a grammar turned into an actual parser?

At this point, this is a rather simple exercise. We already have a `match()` function (pg. 40) that matches the current token against a desired one. When the match succeeds, the current token (`Token`) is consumed and a new one is requested from the scanner. When the match does not succeed, a syntax error is reported. So a function matching *factor* would be as simple as:

```
void factor(void) {
    match(INTLIT == Token? INTLIT: IDENT,
            "number or identifier");
}
```

Of course, a real compiler would do more than that (look up symbols, generate code, etc), but we will get to that later.

What would a matcher for *term* look like? Using our explicit model, it would first match a *factor* and then look at the current token. When that token is in the set of operators accepted by it, it would consume the token, match another *factor* and recurse. This is pretty straight-forward:

```
void term(void) {
    factor();
```

```
    term_ops();
}

void term_ops(void) {
    switch (Token) {
    case STAR:  Token = scan();
                factor();
                term_ops();
                break;
    case SLASH: Token = scan();
                factor();
                term_ops();
                break;
    case MOD:   Token = scan();
                factor();
                term_ops();
                break;
    }
}
```

We *always* call `term_ops()` and let the flow of control fall through the switch statement when the current token is not an operator of *term*. Of course, we can make use of the fact that *recursion* always takes place immediately before returning, so we can do away with it altogether and replace it by an infinite loop, just like in `scanpp()` (pg. 59):

```
void term_ops(void) {
    for (;;) {
        switch (Token) {
        case STAR:  Token = scan();
                    factor();
                    break;
        case SLASH: Token = scan();
                    factor();
                    break;
        case MOD:   Token = scan();
                    factor();
                    break;
        default:    return;
        }
    }
```

```
}
```

Once we have rewritten `term_ops()` in this way, we can move the initial call to `factor()` to `term_ops()`, thereby eliminating the extra function, too:

```
void term(void) {
    factor();
    for (;;) {
        switch (Token) {
        case STAR:  Token = scan();
                    factor();
                    break;
        case SLASH: Token = scan();
                    factor();
                    break;
        case MOD:   Token = scan();
                    factor();
                    break;
        default:    return;
        }
    }
}
```

A parser for *sum* would look quite similar, but it would call `term()` instead of `factor()` and match the operator set $\{PLUS, MINUS\}$ instead of $\{STAR, SLASH, MOD\}$. In fact *all* routines for matching left-associative binary operations follow this very pattern. We will come back to this observation at a later point in this chapter.

9.2 Expression Parsing

The *expression parser* (file `expr.c`) is the largest part of the entire compiler, which is no surprise, given the rich expression syntax and operator semantics of the C language.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Expression parser
 */
```

```
#include "defs.h"
#include "data.h"
#include "decl.h"
#include "prec.h"
```

The following functions will recurse. *Recursion* is a natural process in parsing and it is unavoidable in the end, because it is inherent in the very nature of formal languages: expressions may contain expressions (e.g. in parentheses or array subscripts), statements may be compound statements which in turn contain statements, etc.

```
int asgmnt(int *lv);
int expr(int *lv);
int cast(int *lv);
```

The `primary()` function accepts a *primary factor* of the C language. A primary factor is an *atomic* factor of an expression, a factor that will be evaluated entirely before any operations in an expression can take place. In SubC, this includes *identifiers*, *integer literals*, *string literals*, the special `__argc` keyword, and subexpressions grouped by *parentheses*.[3] Formally, this is what `primary()` accepts:

```
primary :=
          IDENT
        | INTLIT
        | string
        | __ARGC
        | ( expr )

string :=
          STRLIT
        | STRLIT string
```

However, the `primary()` function does not only accept primary factors. Because *syntax analysis*, *semantic analysis*, and *code generation* are tightly coupled in the SubC compiler, it also performs some semantic analysis,

[3] A subexpression is not atomic (indivisible) in the strict sense of the word, but an expression enclosed in parentheses also has to be evaluated completely before its value can be used as a factor.

sets up type information for later processes of the parser, handles implicit function declarations, and does some error reporting and recovery. But let's do this one by one.

Most of the interesting stuff takes place in the `IDENT` case of the `switch`. It first tries to locate an identifier in the symbol table, first in the local one and then in the global one (because local symbols take precedence over globals). It then makes a copy of the symbol name, because scanning the next token will modify the token text (`Text`).

When the identifier is not in the symbol table, there are two cases: when the next token is a left parenthesis, the identifier is a function name and `primary()` adds an implicit *extern* declaration for the function to the symbol table.[4] When there is no subsequent parenthesis, an error ("undeclared variable") is reported. In this case an `int` variable with the given name will be added to the symbol table. Doing so will reduce the number of error messages due to the same undeclared identifier.

The remainder of the `IDENT` case deals mostly with typing and code generation. The `LV` structure *lv*, which is passed as an argument to `primary()`, is filled with type information about the matched identifier.

The `LV` structure has two slots:

(1) `LVSYM` is the slot number of the symbol table entry of an identifier. It is 0 for non-identifier objects.
(2) `LVPRIM` is the primitive type of an object.

The `LVSYM` part is important for *lvalues*. Each *assignment* of the C language has the general form

$$lvalue = rvalue$$

where the lvalue has to specify an *address* and the rvalue just has to evaluate to a value. Many expressions can be both lvalues and rvalues, e.g.:

```
x  x[0]  *x  (*x)[0]
```

An lvalue is turned into an rvalue by an additional step of indirection, e.g. the value of a *variable* is extracted or the address denoted by a *pointer* is replaced by the value pointed to. When we start parsing an expression, we cannot know whether we are dealing with an lvalue or an rvalue.[5] Since

[4] Yes, this method fails in sentences like `(function)()`.

[5] Quick proof: we need to find an assignment operator to know we are parsing an lvalue, but there may be any number of tokens in front of the assignment operator, e.g.: `*x`, `**x`, `***x`, etc.

an indirection cannot be reversed later, the parser assumes each partial expression to be an lvalue. This is why it always fills the `LVSYM` field with the symbol table entry of an identifier when it finds one.

Some identifiers, though, cannot be lvalues, like *functions*, *constants*, and *arrays*. When `primary()` finds one of these types, it will generate code to load their value immediately.

In addition to filling or modifying *lv*, *all* expression parser functions return a flag indicating whether the object described by *lv* is an lvalue (1) or not (0). The following table summarizes the actions of `primary()` for each type of identifier:

Object	LVSYM	LVPRIM	Generate	Return
TVARIABLE	symbol	type	nothing	1
TCONSTANT	0	int	value	0
TFUNCTION	symbol	int(*)()	address	0
(in calls)	symbol	type	nothing	0
TARRAY	symbol	type *	address	0
STRLIT	0	char *	address	0
INTLIT	0	int	value	0

The types of variables, functions, and arrays are extracted from the symbol table. The type information of parenthesized subexpressions is set up by `expr()` (which calls `primary()` in the end).

The `primary()` function handles quite a bit of stuff and introduces several new concepts. Make sure to understand them before proceeding. This is a steep segment in the learning curve, but once you have mastered it things will calm down a bit.

```
static int primary(int *lv) {
    int     a, y, lab;
    char    name[NAMELEN+1];

    lv[LVSYM] = lv[LVPRIM] = 0;
    switch (Token) {
    case IDENT:
        y = findloc(Text);
        if (!y) y = findglob(Text);
        copyname(name, Text);
        Token = scan();
        if (!y) {
            if (LPAREN == Token) {
                y = addglob(name, PINT, TFUNCTION, CEXTERN,
```

```
                -1, 0, NULL, 0);
        }
        else {
            error("undeclared variable: %s", name);
            y = addloc(name, PINT, TVARIABLE, CAUTO,
                0, 0, 0);
        }
    }
    lv[LVSYM] = y;
    lv[LVPRIM] = Prims[y];
    if (TFUNCTION == Types[y]) {
        if (LPAREN != Token) {
            lv[LVPRIM] = FUNPTR;
            genaddr(y);
        }
        return 0;
    }
    if (TCONSTANT == Types[y]) {
        genlit(Vals[y]);
        return 0;
    }
    if (TARRAY == Types[y]) {
        genaddr(y);
        lv[LVPRIM] = pointerto(lv[LVPRIM]);
        return 0;
    }
    return 1;
case INTLIT:
    genlit(Value);
    Token = scan();
    lv[LVPRIM] = PINT;
    return 0;
case STRLIT:
    gendata();
    lab = label();
    genlab(lab);
    while (STRLIT == Token) {
        gendefs(Text, Value);
        Token = scan();
    }
    gendefb(0);
    genldlab(lab);
    lv[LVPRIM] = CHARPTR;
```

```
        return 0;
    case LPAREN:
        Token = scan();
        a = expr(lv);
        rparen();
        return a;
    case __ARGC:
        Token = scan();
        genargc();
        lv[LVPRIM] = PINT;
        return 0;
    default:
        error("syntax error at: %s", Text);
        Token = synch(SEMI);
        return 0;
    }
}
```

The `typematch()` function is the general type matcher of SubC. It checks whether the primitive type $p1$ is compatible to the type $p2$ and returns 1 when the types match and 0 when they do not match. Equal types always match, each integer type (`int`, `char`) matches each integer type, and all pointer (non-integer) types match `void*`.

```
int typematch(int p1, int p2) {
    if (p1 == p2) return 1;
    if (inttype(p1) && inttype(p2)) return 1;
    if (!inttype(p1) && VOIDPTR == p2) return 1;
    if (VOIDPTR == p1 && !inttype(p2)) return 1;
    return 0;
}
```

The `fnargs()` function accepts the arguments of a function call. Formally:

```
fnargs :=
         asgmnt
       | asgmnt , fnargs
```

When the *fn* argument is non-zero, it is the symbol table slot of the object through which the call is performed (either a function or a function

pointer). In this case the `Mtext` field of the symbol table entry is used for type-checking the arguments passed to the function. This works only for functions, though, but not for function pointers (where `Mtext` is `NULL`).

`fnargs()` also collects the types of the actual arguments of the function call in the *signature*[6] array `sgn`. When *fn* is the slot of a function that does not already have a signature, it uses the types of the actual arguments to supply one. Functions without a signature are functions that have been implicitly declared in `primary()`. By adding argument type information to the implicit declaration, the implicit declaration can be used to type-check subsequent forward-calls in the program. Of course, this mechanism is not always accurate, e.g. it will guess `int` even if the actual type is a `char` and may mistake `void` pointers for more specific types. The method will always fail in the case of variable-argument procedures. In these case an explicit prototype declaration has to be supplied.

Also note that the `asgmnt()` function, which accepts an argument,[7] may store an lvalue in *lv*. When it does so (and returns 1), the `rvalue()` function is used to turn it into an rvalue. This pattern is used in all expression parser functions that need rvalues.

Finally note that `fnargs()` passes an additional argument to the called function which contains the number of arguments passed (excluding itself), so the *call frame* of a SubC function call looks like this:

Address	Content
...	...
`16(%ebp)`	second-to-last argument (if any)
`12(%ebp)`	last argument (if any)
`8(%ebp)`	argument count
`4(%ebp)`	return address (saved by `call`)
`0(%ebp)`	caller's frame address (pushed by callee)
`-4(%ebp)`	space for
`-8(%ebp)`	local variables
...	...

The call frame layout differs from traditional C's, which would push the arguments in reverse order, so that the first argument is located at the lowest address above the return address. This facilitates the implementation of *variable-argument functions*, but requires to build a *parse tree* or shuffle the arguments at run time (the SubC compiler does neither).

[6] A "signature" is an n-tuple of the argument types of a function.

[7] In C arguments are technically assignments.

```
static int fnargs(int fn) {
    int     lv[LV];
    int     na = 0;
    char    *types;
    char    msg[100];
    char    sgn[MAXFNARGS+1];

    types = fn? Mtext[fn]: NULL;
    na = 0;
    while (RPAREN != Token) {
        if (asgmnt(lv)) rvalue(lv);
        if (types && *types) {
            if (!typematch(*types, lv[LVPRIM])) {
                sprintf(msg, "wrong type in argument %d"
                    " of call to: %%s",
                    na+1);
                error(msg, Names[fn]);
            }
            types++;
        }
        if (na < MAXFNARGS)
            sgn[na] = lv[LVPRIM], sgn[na+1] = 0;
        na++;
        if (COMMA == Token)
            Token = scan();
        else
            break;
    }
    if (fn && TFUNCTION == Types[fn] && !Mtext[fn])
        Mtext[fn] = globname(sgn);
    rparen();
    genpushlit(na);
    return na;
}
```

The `indirection()` function performs an indirection through a pointer or an array. Note that at this stage an array is just a pointer that is *not* an lvalue (because you cannot assign a value to an array in C; you can only assign to individual elements of an array). An array also loads its address immediately (in `primary()`, pg. 90) while loading the value of a pointer is postponed.

The `rvalue()` function emits code to perform one level of indirection when *lv* describes an lvalue (this is indicated by $a = 1$).

Finally `indirection()` removes one level of indirection from the type field (`LVPRIM`) of *lv* and reports an error when dereferencing a pointer to `void`. It also reports indirection through non-pointers and it always clears the symbol slot of *lv*, because once an indirection has been performed, this information is obsolete.

"Removing a *level of indirection*" from a type means to turn, for instance, an `int**` into an `int*` and an `int*` into an `int`. A `int(*)()` (pointer to an `int` function) is turned into type `char`, so bytes can be extracted from functions. This seems to be common practice.

```
static int indirection(int a, int *lv) {
    if (a) rvalue(lv);
    if (INTPP == lv[LVPRIM]) lv[LVPRIM] = INTPTR;
    else if (CHARPP == lv[LVPRIM]) lv[LVPRIM] = CHARPTR;
    else if (VOIDPP == lv[LVPRIM]) lv[LVPRIM] = VOIDPTR;
    else if (INTPTR == lv[LVPRIM]) lv[LVPRIM] = PINT;
    else if (CHARPTR == lv[LVPRIM]) lv[LVPRIM] = PCHAR;
    else if (VOIDPTR == lv[LVPRIM]) {
        error("dereferencing void pointer", NULL);
        lv[LVPRIM] = PCHAR;
    }
    else if (FUNPTR == lv[LVPRIM]) lv[LVPRIM] = PCHAR;
    else {
        if (lv[LVSYM])
            error("indirection through non-pointer: %s",
                Names[lv[LVSYM]]);
        else
            error("indirection through non-pointer", NULL);
    }
    lv[LVSYM] = 0;
    return lv[LVPRIM];
}
```

The `badcall()` function reports a call to a non-function.

```
static void badcall(int *lv) {
    if (lv[LVSYM])
        error("call of non-function: %s",
            Names[lv[LVSYM]]);
```

```
    else
        error("call of non-function", NULL);
}
```

The `argsok()` function checks the number of arguments (na) to a function call against the number of formal arguments (nf) of the function. When nf is negative, the function is a *variable-argument function* that expects at least $(-nf)-1$ arguments. (Note: variable-argument functions may expect *zero or more* arguments, so we cannot just negate nf to indicate variable arguments).

```
static int argsok(int na, int nf) {
    return na == nf || nf < 0 && na >= -nf-1;
}
```

The `postfix()` function accepts all unary postfix operators of the SubC language:

```
postfix :=
         primary
       | postfix [ expr ]
       | postfix ( )
       | postfix ( fnargs )
       | postfix ++
       | postfix --
```

The post-increment and post-decrement operators are handled by the code generator function `geninc()`, whose second parameter indicates whether the operation is an increment or decrement and whose third parameter indicates whether it is a pre-increment or post-increment. After handling an increment/decrement operator, the a (address) flag is set to zero, because incremented or decremented values are not valid lvalues.

Note that the above grammar indicates recursion once again, so most of the postfix operations can be chained together. In the case of `++` and `--`, this is syntactically fine, but not at the semantic level, because increment[8] operators need an lvalue and the increment operators clear the lvalue flag a. So expressions like `x++++` or `++x--` will be reported as errors, even if

[8] When we talk about "increment" operators, the *negative increment*—i.e. decrement—is typically included.

they are covered by the formal specification. This is common practice in compiler-land.

Function calls are quite straight-forward. The `fnargs()` function handles the function arguments, direct calls are generated for functions and indirect ones for function pointers. Note that the explicit indirect call syntax `(*fnptr)()` is *not* supported. The `genstack()` function generates code to remove the function arguments from the stack at run time.

Finally, `postfix()` handles array and pointer *subscripts*. It accepts the expression between square brackets, scales it when the subscripted object is an array of or pointer to a non-byte-sized object, and adds the result to the address of the object. Note that a subscripted object is always a valid lvalue, so `postfix()` sets $a = 1$ in this case.

```
static int postfix(int *lv) {
    int a, lv2[LV], p, na;

    a = primary(lv);
    for (;;) {
        switch (Token) {
        case LBRACK:
            while (LBRACK == Token) {
                p = indirection(a, lv);
                Token = scan();
                if (expr(lv2))
                    rvalue(lv2);
                if (PINT != lv2[LVPRIM])
                    error("non-integer subscript", NULL);
                if (PINT == p || INTPTR == p ||
                    CHARPTR == p || VOIDPTR == p
                )
                    genscale();
                genadd(PINT, PINT);
                rbrack();
                lv[LVSYM] = 0;
                a = 1;
            }
            break;
        case LPAREN:
            Token = scan();
            na = fnargs(lv[LVSYM]);
            if (lv[LVSYM] && TFUNCTION == Types[lv[LVSYM]]) {
                if (!argsok(na, Sizes[lv[LVSYM]]))
```

```
                    error("wrong number of arguments: %s",
                        Names[lv[LVSYM]]);
                gencall(lv[LVSYM]);
            }
            else {
                if (lv[LVPRIM] != FUNPTR) badcall(lv);
                clear();
                rvalue(lv);
                gencalr();
                lv[LVPRIM] = PINT;
            }
            genstack((na + 1) * INTSIZE);
            a = 0;
            break;
        case INCR:
        case DECR:
            if (a) {
                if (INCR == Token)
                    geninc(lv, 1, 0);
                else
                    geninc(lv, 0, 0);
            }
            else
                error("lvalue required before '%s'", Text);
            Token = scan();
            a = 0;
            break;
        default:
            return a;
        }
    }
}
```

Prefix operators are next. They are handled by the `prefix()` function. Their formal rules are these:

```
prefix :=
          postfix
        | ++ prefix
        | -- prefix
        | & cast
        | * cast
```

```
        | + cast
        | - cast
        | ~ cast
        | ! cast
        | SIZEOF ( primtype )
        | SIZEOF ( primtype * )
        | SIZEOF ( primtype * * )
        | SIZEOF ( INT ( * ) ( ) )
        | SIZEOF ( IDENT )

primtype :=
          INT
        | CHAR
        | VOID
```

Pre-increments and decrements are handled in the same way as their postfix counterparts. All other arithmetic operators follow the same pattern: make sure that the argument is an integer type, accept the operand, generate code, return 0 (not an lvalue). The only exceptions are the `!` operator, which accepts any type, and the unary `+`, which is in fact a null-operation.

The `sizeof` operator accepts an identifier or attempts to parse a subset of the type cast syntax and tries to figure out the size of the specified object. The code is lengthy, but straight-forward.

The `*` (*indirection*) operator simply calls `indirection()` and sets $a = 1$, because indirect objects are always valid lvalues.

The `&` (*address*-of) operator, finally, expects an lvalue, generates its address, and promotes the *lv* type (`LVRIM`) to an additional level of indirection (e.g. `char*` to `char**`, etc). Because `&` expects an lvalue, `&array` must be written as `&array[0]` (this is a violation of TCLP2). The value of `&object` is an *rvalue*.

Note that the *prefix operators* have a lower precedence than the *postfix operators* in C (*prefix* descends into *postfix* and not vice versa). This is why `*a--` parses as `*(a--)`[9] and `*x[1]` as `*(x[1])`.

```
static int prefix(int *lv) {
    int k, y, t, a;
```

[9]The `--` operator is still computed after performing the indirection, but it is applied to `a` and not to `*a`.

```
    switch (Token) {
    case INCR:
    case DECR:
        t = Token;
        Token = scan();
        if (prefix(lv)) {
            if (INCR == t)
                geninc(lv, 1, 1);
            else
                geninc(lv, 0, 1);
        }
        else {
            error("lvalue expected after '%s'",
                t == INCR? "++": "--");
        }
        return 0;
    case STAR:
        Token = scan();
        a = cast(lv);
        indirection(a, lv);
        return 1;
    case PLUS:
        Token = scan();
        if (cast(lv))
            rvalue(lv);
        if (!inttype(lv[LVPRIM]))
            error("bad operand to unary '+'", NULL);
        return 0;
    case MINUS:
        Token = scan();
        if (cast(lv))
            rvalue(lv);
        if (!inttype(lv[LVPRIM]))
            error("bad operand to unary '-'", NULL);
        genneg();
        return 0;
    case TILDE:
        Token = scan();
        if (cast(lv))
            rvalue(lv);
        if (!inttype(lv[LVPRIM]))
            error("bad operand to '~'", NULL);
        gennot();
```

```
        return 0;
    case XMARK:
        Token = scan();
        if (cast(lv))
            rvalue(lv);
        genlognot();
        lv[LVPRIM] = PINT;
        return 0;
    case AMPER:
        Token = scan();
        if (!cast(lv))
            error("lvalue expected after unary '&'", NULL);
        if (lv[LVSYM]) genaddr(lv[LVSYM]);
        lv[LVPRIM] = pointerto(lv[LVPRIM]);
        return 0;
    case SIZEOF:
        Token = scan();
        lparen();
        if (CHAR == Token || INT == Token || VOID == Token) {
            k = CHAR == Token? CHARSIZE:
                INT == Token? INTSIZE: 0;
            Token = scan();
            if (STAR == Token) {
                k = PTRSIZE;
                Token = scan();
                if (STAR == Token) Token = scan();
            }
            else if (0 == k) {
                error("cannot take sizeof(void)", NULL);
            }
        }
        else if (IDENT == Token) {
            y = findloc(Text);
            if (!y) y = findglob(Text);
            if (!y || !(k = objsize(Prims[y], Types[y],
                                    Sizes[y]))
            )
                error("cannot take sizeof object: %s",
                    Text);
            Token = scan();
        }
        else {
            error("cannot take sizeof object: %s", Text);
```

```
            Token = scan();
        }
        genlit(k);
        rparen();
        lv[LVPRIM] = PINT;
        return 0;
    default:
        return postfix(lv);
    }
}
```

The `cast()` function handles *type casts*:

```
cast :=
         prefix
       | ( primtype ) prefix
       | ( primtype * ) prefix
       | ( primtype * * ) prefix
       | ( INT ( * ) ( ) ) prefix
```

Again, the code is rather straight-forward. There is one interesting point to observe here, though: `cast()` contains the only place in the SubC compiler where the parser may have to *backtrack*, i.e. reject an already-accepted token.

This is because an opening *parenthesis* can introduce either a type cast or a parenthesized subexpression. So `cast()` accepts the parenthesis and advances to the subsequent token. When this token is a valid type name, it accepts the rest of a type cast and then descends into *prefix*.

When the first token after the opening parenthesis is not a type name, though, the parser has entered a dead end, because after accepting the parenthesis the parenthesis can no longer be accepted in *primary*, which handles subexpression grouping. So it has to *reject* the current token and re-load the previous one, which is known to be a left parenthesis.

```
int cast(int *lv) {
    int t, a;

    if (LPAREN == Token) {
        Token = scan();
        if (INT == Token) {
            t = PINT;
```

```
            Token = scan();
        }
        else if (CHAR == Token) {
            t = PCHAR;
            Token = scan();
        }
        else if (VOID == Token) {
            t = PVOID;
            Token = scan();
        }
        else {
            reject();
            Token = LPAREN;
            strcpy(Text, "(");
            return prefix(lv);
        }
        if (PINT == t && LPAREN == Token) {
            Token = scan();
            match(STAR, "int(*)()");
            rparen();
            lparen();
            rparen();
            t = FUNPTR;
        }
        else if (STAR == Token) {
            t = pointerto(t);
            Token = scan();
            if (STAR == Token) {
                t = pointerto(t);
                Token = scan();
            }
        }
        rparen();
        a = prefix(lv);
        lv[LVPRIM] = t;
        return a;
    }
    else {
        return prefix(lv);
    }
}
```

We will now come back to the observation that all parser functions that handle *left-associative* binary *operators* look pretty much the same. The following grammar describes 16 operators with 8 different levels of *precedence*. To implement this grammar as a traditional *recursive descent* parser, where one function handles one level of precedence, we would have to write eight functions, one for each set of rules in the following grammar.

```
term :=
          cast
        | term * cast
        | term / cast
        | term % cast

sum :=
          term
        | sum + term
        | sum - term

shift :=
          sum
        | shift << sum
        | shift >> sum

relation :=
          shift
        | relation < shift
        | relation > shift
        | relation <= shift
        | relation >= shift

equation :=
          relation
        | equation == relation
        | equation != relation

binand :=
          equation
        | binand & equation

binxor :=
          binand
```

```
        | binxor ^ binand

binor :=
          binxor
        | binor '|' binxor

binexpr :=
          binor
```

Recursive-descent can be a real performance killer in languages with lots of different levels of precedence, such as C. For example, when parsing an expression like `x^y^z`, the parser would have to descend all the way down from *binxor* to *primary* for each variable in the expression. All the functions in between would do nothing but propagate the call toward its final destination. However, recursive descent (*RD*) parsers are practical and easy to maintain: there is one function for each set of rules in the formal grammar and precedence is implemented by descending down the hierarchy of rules (or functions). The only real alternative would be use use a different approach to parsing altogether, which would most certainly mean to resort to using a *meta-compiler*. If we want to avoid this step, it would be nice if we could eliminate some levels of descent from the RD parser. Indeed, we can, and it is not even that hard.

The `binexpr()` function handles all the binary operators covered by the above rules. It basically uses the pattern described in the short introduction to the theory of parsing at the beginning of this chapter, but in addition it keeps track of eight levels of precedence inside of one single function.

The frameworks are identical: First *binexpr* descends into *cast* and then it accepts a set of operators, descending again into *cast* to collect the right-hand-side operands. However, it does not immediately emit code for the operations, but puts them on a stack instead. The local `ops[]` array is used as a stack for operations.

The algorithm works as follows:

(1) Accept the first operand.
(2) When the current token is not in the operator set, go to (8).
(3) When there is an operator O_{top} on the top of the stack *and* the precedence of O_{top} is not lower than the precedence of the current operator O, emit code for O_{top} and pop it off the stack.[10]
(4) Push the current operator O to the stack.

[10] The operands are already in place; we will discuss this later (pg. 168ff).

(5) Accept the current operator O.
(6) Accept the next operand.
(7) Go to (2).
(8) While there is an operator O_{top} on the top of the stack, emit code for O_{top} and pop it off the stack.

That's it. In addition to the operator stack `ops[]`, the types of the arguments are kept in the primitive type stack `prs[]`. The operator in `ops[sp-1]` (where *sp* is the stack pointer) uses the types in `prs[sp-1]` and `prs[sp]`, so the `genbinop()` function generates code that applies `ops[sp-1]` to objects of the types `prs[sp-1]` and `prs[sp]`. The resulting type is stored in `prs[sp-1]`, overwriting the left-hand operand. The operator and the right-hand-side operand type are removed from the stack (by decrementing `sp`). When the algorithm is done, the type of the entire accepted expression is in `prs[0]`.

I have invented[11] this algorithm in the 1990's, when speed still mattered, even in recursive-descent parsers, and the *bullshit-tolerant* user had not yet been invented or, at least, they had not yet infested the nîche of academia that I inhabited at that time. I dubbed it "*falling-precedence parsing*", for the lack of a better name and described it later in my 1996 book "Lightweight Compiler Techniques".

Let's see the algorithm in action. The expression `a==b+c-d*e|f` contains a lot of different operators with various precedence values, which are attached to the operators in the following formula (higher values indicate stronger precedence):

a $==_3$ b $+_6$ c $-_6$ d $*_7$ e $|_0$ f

So `a==b+c-d*e|f` should in fact be interpreted as `(a==(b+c-(d*e)))|f`. Let's see:

INPUT	OUTPUT (RPN)	STACK
a [==] b + c - d * e | f	a	==
a == b [+] c - d * e | f	a b	== +
a == b + c [-] d * e | f	a b c +	== -
a == b + c - d [*] e | f	a b c + d	== - *
a == b + c - d * e [|] f	a b c + d e * - ==	|
a == b + c - d * e | f []	a b c + d e * - == f |	

The above output is in *reverse polish notation* (*RPN*), where the operands precede the operator, so $a+b$ would be written as $ab+$ and $a+b*c$

[11] The idea is quite obvious, so other people may have invented it before me.

as $abc*+$. The `[]` marker indicates the current operator. Each operand is output immediately, but operators are only output when precedence does not increase (falls back to or stays at the same level). RPN can be converted to infix notation by moving an operator between its operands and adding parentheses around the operation:

```
a b c + d e * - == f |
a (b+c) d e * - == f |
a (b+c) (d*e) - == f |
a ((b+c)-(d*e)) == f |
(a==((b+c)-(d*e))) f |
((a==((b+c)-(d*e)))|f)
```

After removing redundant parentheses, the resulting expression is equal to the explicitly grouped expression `(a==(b+c-(d*e)))|f`, so the algorithm seems to be sound. A formal proof is left as an exercise to the reader.

BTW, there can never be more than eight operators on the stack, because precedence cannot increase more then seven times in a row. (Make the array size nine for reasons of superstition.) The following formula would trigger the worst case:

```
a | b ^ c & d == e < f >> g + h * i
```

```
static int binexpr(int *lv) {
    int     ops[9], prs[10], sp = 0;
    int     a, a2 = 0, lv2[LV];

    a = cast(lv);
    prs[0] = lv[LVPRIM];
    while (SLASH == Token || STAR == Token || MOD == Token ||
        PLUS == Token || MINUS == Token || LSHIFT == Token ||
        RSHIFT == Token || GREATER == Token ||
        GTEQ == Token || LESS == Token || LTEQ == Token ||
        EQUAL == Token || NOTEQ == Token || AMPER == Token ||
        CARET == Token || PIPE == Token
    ) {
        if (a) rvalue(lv);
        if (a2) rvalue(lv2);
        while (sp > 0 && Prec[Token] <= Prec[ops[sp-1]]) {
            prs[sp-1] = genbinop(ops[sp-1], prs[sp-1],
                                 prs[sp]);
            sp--;
        }
```

```
        ops[sp++] = Token;
        Token = scan();
        a2 = cast(lv2);
        prs[sp] = lv2[LVPRIM];
        a = 0;
    }
    if (a2) rvalue(lv2);
    while (sp > 0) {
        prs[sp-1] = genbinop(ops[sp-1], prs[sp-1], prs[sp]);
        sp--;
    }
    lv[LVPRIM] = prs[0];
    return a;
}
```

The `cond2()` function handles the logical *short-circuit*[12] operators `&&` ("and") and `||` ("or"). Formally:

```
logand :=
         binexpr
       | logand && binexpr

logor :=
         logand
       | logor || logand
```

The `&&` and `||` operators may be considered to be "set value and jump" instructions rather than logic operators. When the left side of a `&&` operator evaluates to zero, it jumps immediately over its right operand (never evaluating it) and when the left side of `||` is non-zero, it changes it to one and jumps over its right operand. When multiple `&&` or `||` operators are found in a row, each of the operators jumps to the end of the entire row instead of just skipping its current right-hand operator, as depicted in figure 9.3.

As can be seen in the figure, both operators implement the same control structure, but with different parameters: `&&` branches to the end when its left side operand is zero (delivering 0) and `||` branches when its left operand is non-zero (delivering 1).

[12] They are called so because they "short-circuit" evaluation by skipping the remaining operands as soon as their expression can no longer become "true" in an "and" or "false" in an "or".

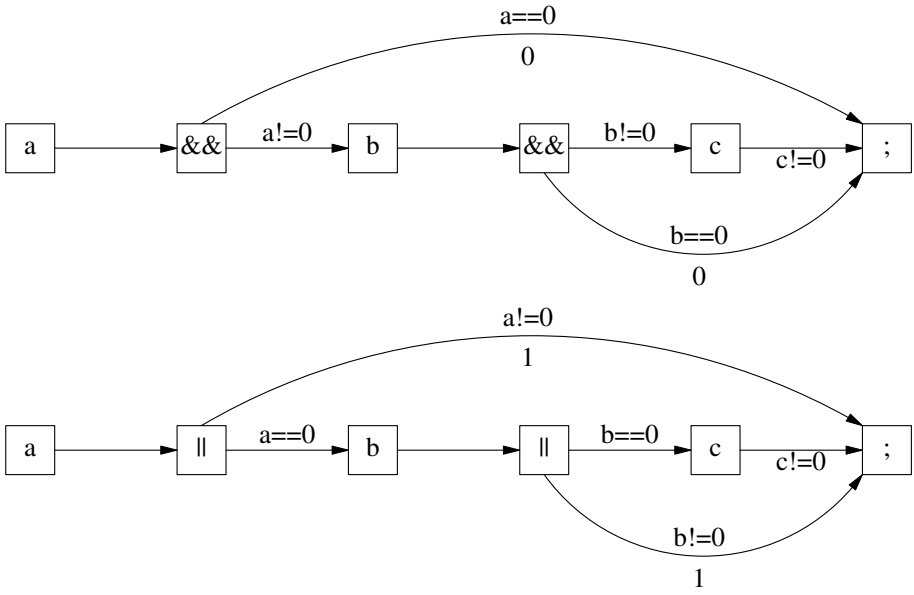


Figure 9.3: The control structures of the short-circuit operators

BTW, the text above an arrow describes the condition upon which the path denoted by the arrow is taken. The text below the arrow specifies the value delivered to its destination.

The `cond2()` function implements this control structure by emitting "*branch-on-false*" or "*branch-on-true*" instructions, respectively. The *op* parameter of `cond2()` determines the type of branch instruction to emit. It is entered from `cond3()` (below) with *op=LOGOR*, indicating that a sequence of `||` operators is to be parsed, and then re-enters itself with *op=LOGAND* to handle sequences of `&&` operators. The operands of the latter are accepted by `binexpr()`.

At the end of a sequence `cond2()` emits a common exit label for all operators in the same sequence. At this point the value of the last operand is in the primary register. Because the short-circuit operators always return 0 or 1, though, it has to be *normalized*. The corresponding code is emitted by the `genbool()` function.

Note that `cond2()` only emits a label after actually processing some `&&` or `||` operators.

Also note that `cond2()` implements two levels of precedence by recursing into itself.

```
static int cond2(int *lv, int op) {
    int  a, a2 = 0, lv2[LV];
```

```
    int  lab = 0;

    a = op == LOGOR? cond2(lv, LOGAND): binexpr(lv);
    while (Token == op) {
        if (!lab) lab = label();
        if (a) rvalue(lv);
        if (a2) rvalue(lv2);
        if (op == LOGOR)
            genbrtrue(lab);
        else
            genbrfalse(lab);
        clear();
        Token = scan();
        a2 = op == LOGOR? cond2(lv2, LOGAND): binexpr(lv2);
        a = 0;
    }
    if (lab) {
        if (a2) rvalue(lv2);
        genlab(lab);
        genbool();
        load();
    }
    return a;
}
```

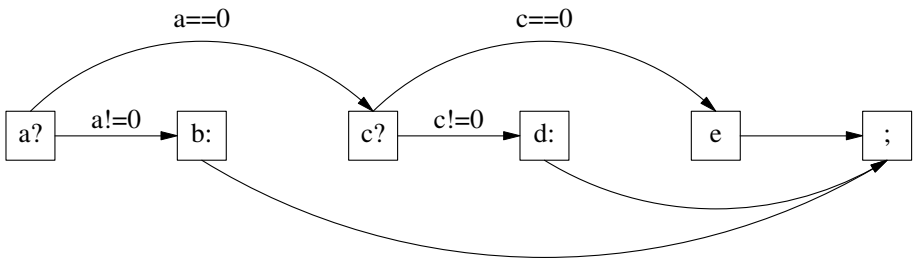


Figure 9.4: The control structure of the `?:` operator

When `&&` is thought of as an "if" and `||` as an "if-not", then the ternary `?:` operator can be viewed as an "if-else". It is handled by the `cond3()` ("ternary conditional") function. Its syntax is the following:

```
condexpr :=
          logor
        | logor ? logor : condexpr
```

Its control flow structure is depicted in figure 9.4. As can be seen here, the "?" part of the operator can be interpreted as a "branch on false" to the third operand (the "else part") of the operation, and the ":" part can be viewed as a branch to a common end label, just as in `cond2()`.

The `cond3()` function has to perform some type checking, too, because the *b* and *c* parts of `a?b:c` and the *b*, *d*, and *e* parts in `a?b:c?d:e` must all have the same type. Otherwise, how would you type-check a function call like `strcmp("foo", x? y: z)`?

```
static int cond3(int *lv) {
    int  a, lv2[LV], p;
    int  l1 = 0, l2 = 0;

    a = cond2(lv, LOGOR);
    p = 0;
    while (QMARK == Token) {
        l1 = label();
        if (!l2) l2 = label();
        if (a) rvalue(lv);
        a = 0;
        genbrfalse(l1);
        clear();
        Token = scan();
        if (cond2(lv, LOGOR))
            rvalue(lv);
        if (!p) p = lv[LVPRIM];
        if (!typematch(p, lv[LVPRIM]))
            error("incompatible types in '?:'", NULL);
        genjump(l2);
        genlab(l1);
        clear();
        colon();
        if (cond2(lv2, LOGOR))
            rvalue(lv2);
        if (QMARK != Token)
            if (!typematch(p, lv2[LVPRIM]))
                error("incompatible types in '?:'", NULL);
    }
    if (l2) {
        genlab(l2);
        load();
    }
```

```
    return a;
}
```

There are lots of assignment operators in C, and they are all accepted by the `asgmnt()` function, which implements these rules:

```
asgmnt :=
          condexpr
        | condexpr = asgmnt
        | condexpr *= asgmnt
        | condexpr /= asgmnt
        | condexpr %= asgmnt
        | condexpr += asgmnt
        | condexpr -= asgmnt
        | condexpr <<= asgmnt
        | condexpr >>= asgmnt
        | condexpr &= asgmnt
        | condexpr ^= asgmnt
        | condexpr |= asgmnt
```

Assignment operators actually *do* associate to the right in C, so `a -= b -= c` actually means `a -= (b -= c)`. Therefore, the `asgmnt()` function does not handle chains of assignment operators in a loop, but actually recurses to accept the right-hand side of an assignment operator.

Type checking of the `=` operator is done by the `typematch()` function, but type checking of the combined operators is performed in `genstore()` because it is a bit more elaborate.

When the left operand of a combined assignment operator is an indirect operand (an expression involving a `*` or `[]` operator), then the `asgmnt()` function saves a copy of the address of this operand on the stack before performing the indirection, so the address is available for storing the result of the right-hand side later. The resulting code would look like this:

```
# f(int *a) { *a += 5; }
    ...
    movl    12(%ebp),%eax    # location pointed to by a
    pushl   %eax             # save location (a)
    movl    (%eax),%eax      # fetch *a
    pushl   %eax             # add 5 to a
    movl    $5,%eax
```

```
    popl    %ecx
    addl    %ecx,%eax
    popl    %edx            # restore location to %edx
    movl    %eax,(%edx)     # save result (*a+5) to *a
    ...
```

```
int asgmnt(int *lv) {
    int a, lv2[LV], op;

    a = cond3(lv);
    if (ASSIGN == Token || ASDIV == Token ||
        ASMUL == Token || ASMOD == Token ||
        ASPLUS == Token || ASMINUS == Token ||
        ASLSHIFT == Token || ASRSHIFT == Token ||
        ASAND == Token | ASXOR == Token ||
        ASOR == Token
    ) {
        op = Token;
        Token = scan();
        if (ASSIGN != op && !lv[LVSYM]) {
            genpush();
            genind(lv[LVPRIM]);
        }
        if (asgmnt(lv2)) rvalue(lv2);
        if (ASSIGN == op)
            if (!typematch(lv[LVPRIM], lv2[LVPRIM]))
                error("assignment from incompatible type",
                    NULL);
        if (a)
            genstore(op, lv, lv2);
        else
            error("lvalue expected in assignment", Text);
        a = 0;
    }
    return a;
}
```

The `expr()` function, finally, accepts an expression:

```
expr :=
        asgmnt
      | asgmnt , expr
```

A C expression is a sequence of assignments separated by commas. The assignment operators bind stronger than the comma, so the expression `a=1,2` would mean `(a=1),2`: assign 1 to a and then return 2.

The `clear()` function tells the code generator that the primary register is "empty", i.e. the value in it does not have to be saved when a new value is to be loaded. Without this hint each value that is followed by a comma would end up on the stack.

A comma automatically turns an expression into an rvalue; sentences of the form `(a,b)=1` are not valid C expressions.

```
int expr(int *lv) {
    int a, a2 = 0, lv2[LV];

    a = asgmnt(lv);
    while (COMMA == Token) {
        Token = scan();
        clear();
        a2 = asgmnt(lv2);
        a = 0;
    }
    if (a2) rvalue(lv2);
    return a;
}
```

The `rexpr()` function is a convenience function that accepts exactly the same input as `expr()`. However, it sets up an `LV` structure internally, automatically transforms the result into an rvalue, and returns the type of the expression. It comes in handy when we just want to compile an expression that is known to be an rvalue. "Rexpr" stands for "rvalue expression".

```
int rexpr(void) {
    int lv[LV];

    if (expr(lv))
        rvalue(lv);
    return lv[LVPRIM];
}
```

9.3 Constant Expression Parsing

A *constant expression* is an expression whose value can be computed at compile time. A full C compiler would just use the expression parser to parse a constant expression, pass the resulting abstract program though the constant expression folder, and complain when it does not reduce to a single constant.

The SubC compiler cannot do this for three reasons: (1) it does not generate an abstract program, (2) it does not have an optimizer, and (3) it generates code while parsing expressions, so it would generate dead code while parsing a constant expression. So we have to use a different approach.

The file `cexpr.c` contains a parser for a subset of C expressions. The subset is formalized in the following grammar, which is basically the grammar implemented by `binexpr()` plus a few unary operators and expression grouping. The only primary factors accepted by the constant expression parser are identifiers and integer literals.

```
cfactor :=
          INTLIT
        | IDENT
        | - cfactor
        | ~ cfactor
        | ( constexpr )

cterm :=
          cfactor
        | cterm * cprefix
        | cterm / cprefix
        | cterm % cprefix

csum :=
          cterm
        | csum + cterm
        | csum - cterm

cshift :=
          csum
        | cshift << csum
        | cshift >> csum
```

```
crelation :=
          cshift
        | crelation < cshift
        | crelation > cshift
        | crelation <= cshift
        | crelation >= cshift

cequation :=
          crelation
        | cequation == crelation
        | cequation != crelation

cbinand :=
          cequation
        | cbinand & cequation

cbinxor :=
          cbinand
        | cbinxor ^ cbinand

cbinor :=
          cbinxor
        | cbinor ^ cbinxor

constexpr :=
          cbinor
```

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Constant expression parser
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
#include "prec.h"
```

The `constfac()` function accepts a *constant factor*. A constant factor is an identifier, an integer literal, or a unary operator applied to a constant factor. When a constant factor is an identifier, then the identifier must be the name of a constant (`enum`).

Note that numeric values defined by `#define` expand to integer literals before the constant expression parser sees them.

```
static int constfac(void) {
    int y, v;

    v = Value;
    if (INTLIT == Token) {
        Token = scan();
        return v;
    }
    if (MINUS == Token) {
        Token = scan();
        return -constfac();
    }
    if (TILDE == Token) {
        Token = scan();
        return ~constfac();
    }
    if (LPAREN == Token) {
        Token = scan();
        v = constexpr();
        rparen();
        return v;
    }
    if (Token == IDENT) {
        y = findglob(Text);
        if (!y || Types[y] != TCONSTANT)
            error("not a constant: %s", Text);
        Token = scan();
        return y? Vals[y]: 0;
    }
    else {
        error("constant "expression expected at: %s", Text);
        Token = scan();
        return 1;
    }
}
```

The `constop()` ("constant operation") function applies the operation *op* (identified by its token) to the values $v1$ and $v2$ and returns the result. It reports the attempt to perform a division by zero, so a declaration like `int x=1/0;` will not crash the compiler.

```
static int constop(int op, int v1, int v2) {
    if ((SLASH == op || MOD == op) && 0 == v2) {
        error("constant divide by zero", NULL);
        return 0;
    }
    switch (op) {
    case SLASH:   v1 /= v2; break;
    case STAR:    v1 *= v2; break;
    case MOD:     v1 %= v2; break;
    case PLUS:    v1 += v2; break;
    case MINUS:   v1 -= v2; break;
    case LSHIFT:  v1 <<= v2; break;
    case RSHIFT:  v1 >>= v2; break;
    case GREATER: v1 = v1 > v2; break;
    case GTEQ:    v1 = v1 >= v2; break;
    case LESS:    v1 = v1 < v2; break;
    case LTEQ:    v1 = v1 <= v2; break;
    case EQUAL:   v1 = v1 == v2; break;
    case NOTEQ:   v1 = v1 != v2; break;
    case AMPER:   v1 &= v2; break;
    case CARET:   v1 ^= v2; break;
    case PIPE:    v1 |= v2; break;
    }
    return v1;
}
```

The `constexpr()` function, which accepts a constant expression and returns its value, is a modified version of the `binexpr()` function (pg. 107) used in the expression parser. Instead of keeping track of types, though, it keeps track of values. (We know that all constant expression are of the type `int`.)

```
int constexpr(void) {
    int v, ops[9], vals[10], sp = 0;

    vals[0] = constfac();
    while (SLASH == Token || STAR == Token ||
           MOD == Token || PLUS == Token ||
           MINUS == Token || LSHIFT == Token ||
           RSHIFT == Token || GREATER == Token ||
           GTEQ == Token || LESS == Token ||
```

```
            LTEQ == Token || EQUAL == Token ||
            NOTEQ == Token || AMPER == Token ||
            CARET == Token || PIPE == Token
    ) {
        while (sp > 0 && Prec[Token] <= Prec[ops[sp-1]]) {
            v = constop(ops[sp-1], vals[sp-1], vals[sp]);
            vals[--sp] = v;
        }
        ops[sp++] = Token;
        Token = scan();
        vals[sp] = constfac();
    }
    while (sp > 0) {
        v = constop(ops[sp-1], vals[sp-1], vals[sp]);
        vals[--sp] = v;
    }
    return vals[0];
}
```

9.4 Statement Parsing

Statement parsing is implemented in the `stmt.c` file. The C statement parser is quite small and simple when compared to the expression parser.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Statement parser
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

The `stmt()` function will recurse.

```
void stmt(void);
```

At this point we define a *compound statement* simply as a brace-delimited list of statements:

```
compound :=
          { stmt_list }
        | { }

stmt_list:
          stmt
        | stmt stmt_list
```

Function bodies with local declarations will be handled separately. This means that generic compound statements cannot have local variables in SubC.

```
void compound(int lbr) {
    if (lbr) Token = scan();
    while (RBRACE != Token) {
        if (eofcheck()) return;
        stmt();
    }
    Token = scan();
}
```

The `pushbreak()` function pushes a label onto the *break stack* and the `pushcont()` function pushes a label onto the *continue stack*. The topmost elements of these stacks hold the current destinations for `break` and `continue` statements, respectively.

```
static void pushbrk(int id) {
    if (Bsp >= MAXBREAK)
        fatal("too many nested loops/switches");
    Breakstk[Bsp++] = id;
}

static void pushcont(int id) {
    if (Csp >= MAXBREAK)
        fatal("too many nested loops/switches");
    Contstk[Csp++] = id;
}
```

The `break_stmt()` and `continue_stmt()` functions handle `break` and `continue` statements. Their syntaxes are trivial:

```
break_stmt    := BREAK ;
continue_stmt := CONTINUE ;
```

When the break stack is empty, a `break` statement occurs outside of a loop or switch context. This is syntactically OK, but a semantic error. The same applies to `continue` when it appears outside of a loop context.

```
static void break_stmt(void) {
    Token = scan();
    if (!Bsp) error("'break' not in loop/switch context",
                    NULL);
    genjump(Breakstk[Bsp-1]);
    semi();
}

static void continue_stmt(void) {
    Token = scan();
    if (!Csp) error("'continue' not in loop context", NULL);
    genjump(Contstk[Csp-1]);
    semi();
}
```

The `do_stmt()` function accepts a `do` loop:

```
do_stmt := DO stmt WHILE ( expr ) ;
```

The code of all loop-accepting functions is similar. The functions generate labels for re-entering the loop via `continue` and exiting the loop via `break` and push them onto the break and continue stacks. They may also set up some additional labels that are used internally. Then they accept a statement (recursively) which forms the *body* of the loop. Finally they remove the labels that they have pushed onto the stacks.

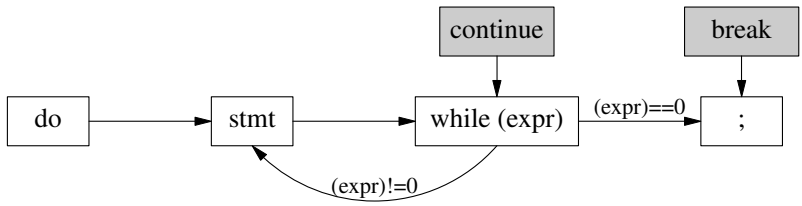


Figure 9.5: The control structure of the `do` statement

Figure 9.5 illustrates the control structure implemented by the `do` loop.

```
static void do_stmt(void) {
    int ls, lb, lc;

    Token = scan();
    ls = label();
    pushbrk(lb = label());
    pushcont(lc = label());
    genlab(ls);
    stmt();
    match(WHILE, "'while'");
    lparen();
    genlab(lc);
    rexpr();
    genbrtrue(ls);
    genlab(lb);
    rparen();
    semi();
    Bsp--;
    Csp--;
}
```

The `for` statement is a bit tricky to implement without building an abstract syntax tree first. The `for_stmt()` function gives its best shot. First, here is the formal syntax of `for`:

```
for_stmt := FOR ( opt_expr ; opt_expr ; opt_expr ) stmt

opt_expr :=
       | expr
```

Note the *empty right-hand side* in the first *opt_expr* rule! It is easy to miss between the := and the subsequent "|". The *opt_expr* rule defines either an expression or "nothing", i.e. an empty subsentence. An empty production is often denoted by an ϵ (*epsilon*) in formal grammars, and an empty production is also called an ϵ-production. So we could write *opt_expr* more readably as

opt_expr := ϵ | *expr*

In the *for_stmt* rule *opt_expr* is used to indicate that each of the three expression in the *head*[13] of the `for` loop may be omitted.

[13]The part that comes before the body.

The reason why implementing `for` in a naïve way is hard is the order of evaluation of its components: In

```
for (init ; test ; modify) stmt
```

the *modify* part is executed *after* the *stmt* part. However, the *modify* part has to be parsed first, because it comes first in the input stream, and because the SubC parser emits code *while* parsing, we have to emit the code of *modify* before that of *stmt*. So we have to insert a couple of jumps to make things work out right, as shown in figure 9.6;

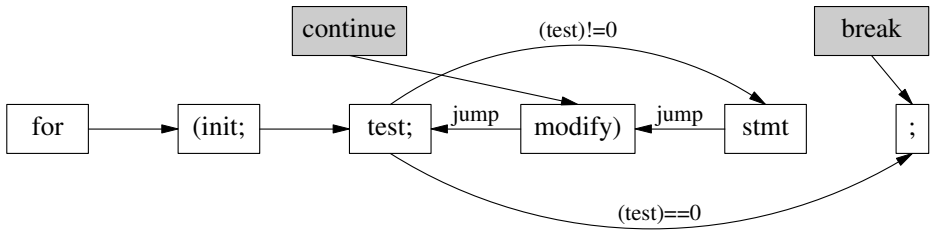


Figure 9.6: The control structure of the `for` statement

When the *test* evaluates positive, the code emitted by `for_stmt()` jumps over the *modify* part to the body (*stmt*). At the end of the statement, the code jumps back to *modify*, which in turn jumps back to *test*. This is certainly not an ideal solution, but it works. Note in particular that this approach emits a lot of redundant jumps for the traditional C rendition of the *endless loop* (`for(;;)`).

Also note that `continue` jumps to the *modify* part of a `for` loop.

Finally note that the `for` loop does not have a delimiting *semicolon* at the end. Because its rule ends with a *stmt*, the semicolon is already being matched by the statement handler. Expecting an additional one would be a violation of the spec.

```
static void for_stmt(void) {
    int ls, lbody, lb, lc;

    Token = scan();
    ls = label();
    lbody = label();
    pushbrk(lb = label());
    pushcont(lc = label());
    lparen();
    if (Token != SEMI) {
```

```
        rexpr();
        clear();
    }
    semi();
    genlab(ls);
    if (Token != SEMI) {
        rexpr();
        clear();
        genbrfalse(lb);
    }
    genjump(lbody);
    semi();
    genlab(lc);
    if (Token != RPAREN) {
        rexpr();
        clear();
    }
    genjump(ls);
    rparen();
    genlab(lbody);
    stmt();
    genjump(lc);
    genlab(lb);
    Bsp--;
    Csp--;
}
```

The `if_stmt()` function accepts the `if` and `if/else` statements:

```
if_stmt :=
           IF ( expr ) stmt
         | IF ( expr ) stmt ELSE stmt
```

The `if/else` statement is illustrated in figure 9.7. The simple `if` statement simply jumps over the "consequent" part directly to the ";" in case the *expr* is false.

```
static void if_stmt(void) {
    int l1, l2;

    Token = scan();
    lparen();
```

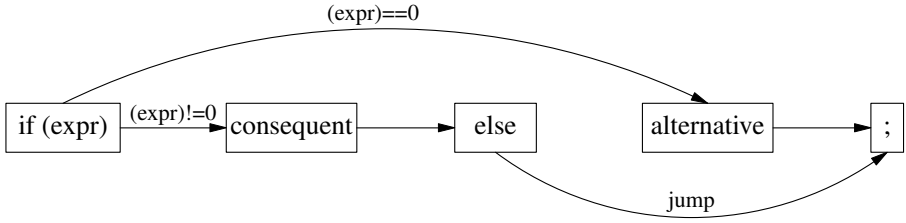


Figure 9.7: The control structure of the `if/else` statement

```
    rexpr();
    clear();
    rparen();
    l1 = label();
    genbrfalse(l1);
    stmt();
    if (ELSE == Token) {
        l2 = label();
        genjump(l2);
        genlab(l1);
        l1 = l2;
        Token = scan();
        stmt();
    }
    genlab(l1);
}
```

The `return_stmt()` function handles `return` statements:

```
return_stmt :=
            RETURN ;
          | RETURN expr ;
```

It makes sure that the type of the *expr*—if any—matches the return type of the function in which the `return` appears. In addition, functions returning `void` are not allowed to return a value, while functions returning non-`void` must return a value. The code emitted by `return_stmt()` simply jumps to the exit label (`Retlab`) of the current function.

```
static void return_stmt(void) {
    int lv[LV];
```

```
    Token = scan();
    if (Token != SEMI) {
        if (expr(lv))
            rvalue(lv);
        if (!typematch(lv[LVPRIM], Prims[Thisfn]))
            error("incompatible type in 'return'", NULL);
    }
    else {
        if (Prims[Thisfn] != PVOID)
            error("missing value after 'return'", NULL);
    }
    genjump(Retlab);
    semi();
}
```

There are two functions that handle the syntax analysis of `switch` statements: the `switch_stmt()` function, which accepts the head of the statement, and the `switch_block()` function, which accepts its body. Together they implement the following rules:

```
switch_stmt := SWITCH ( expr ) { switch_block }

switch_block :=
         switch_block_stmt
       | switch_block_stmt switch_block

switch_block_stmt :=
         CASE constexpr :
       | DEFAULT :
       | stmt
```

Note that we actually create new syntactical context here, called a "*switch block*", which may contain all types of regular statements plus the `case` and `default` statements. This means that the `case` and `default` keywords will only be accepted inside of the bodies of `switch`. Their appearance in any other location would simply be a syntax error, so there is no need to figure out the contexts of `case` and `default` at the semantic level.

The advantage of this approach is that it performs context checking at a purely syntactic level, so we do not need to maintain additional semantic information. The drawback of the approach is that an out-of-context key-

word is simply a "syntax error". We can improve error reporting, though, by inserting an *error production* in the general statement handler (see pg. 132).

The principal control flow of `switch` is depicted in figure 9.8.

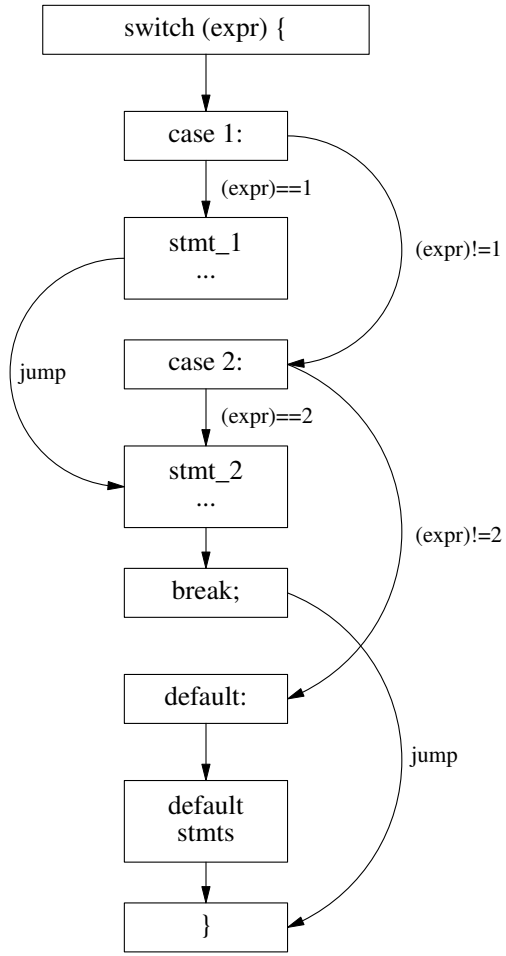


Figure 9.8: The control structure of the `switch` statement

However, the model shown in figure 9.8 is rather cumbersome, because it has to jump around `case`s when the flow of control *falls through* to the next case. This would lead to particularly nasty code in `switch`es with multiple subsequent `case`s that lead to a common block of code, like in:

```
switch (x) {
case 1: case 3: case 5: case 7: case 9:
    return "odd";
}
```

This is why most compilers use a *switch table* to implement `switch`. The principle is illustrated in figure 9.9. A switch table maps the values of the individual `case` statements to labels in the code. Each label points to the statement block of the corresponding case value. Because no branch code is contained in the switch block itself, `case` fall-through happens naturally. A `break` statement is simply a jump to the end of the switch block.

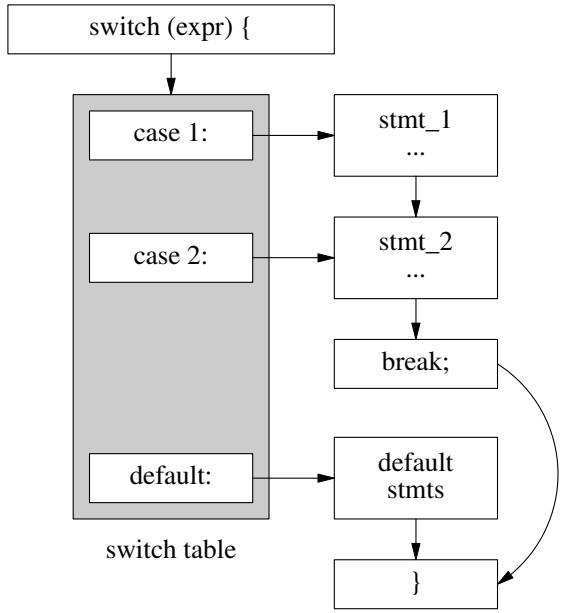


Figure 9.9: The implementation of the `switch` statement

The `switch_block()` function collects the `case` values and associated labels in the `cval[]` and `clab[]` arrays. After parsing the entire block, it passes the arrays to the `genswitch()` function of the code generator, which generates the switch table and the code to evaluate it.

Full C compilers often use quite fancy techniques for evaluating switch tables as fast as possible, which is why `switch` is usually faster than nested `if`. Some of these techniques include:

- *Direct lookup*, where the length of the switch table is equal to the highest `case` value minus the smallest value, and there is a slot for each case in between.[14] This is the fastest method, but it is only practical for dense and/or short sets of values.
- *Hash tables* are quite efficient for large sets, but impose a high fixed cost, so they are not suitable for small sets of values.

[14] Unused slots simply point to the `default` label.

- *Binary search*[15] is not quite as efficient as a hash table, but also works fine for small sets.

A clever compiler could even select the proper method for each individual table, but this is far beyond the scope of SubC, which uses a simple linear search through the entire table. See the `genswitch()` function (pg. 183) for further details.

```
static void switch_block(void) {
    int     lb, ls, ldflt = 0;
    int     cval[MAXCASE];
    int     clab[MAXCASE];
    int     nc = 0;

    Token = scan();
    pushbrk(lb = label());
    ls = label();
    genjump(ls);
    while (RBRACE != Token) {
        if (eofcheck()) return;
        if ((CASE == Token || DEFAULT == Token) &&
            nc >= MAXCASE
        ) {
            error("too many 'case's in 'switch'", NULL);
            nc = 0;
        }
        if (CASE == Token) {
            Token = scan();
            cval[nc] = constexpr();
            genlab(clab[nc++] = label());
            colon();
        }
        else if (DEFAULT == Token) {
            Token = scan();
            ldflt = label();
            genlab(ldflt);
            colon();
        }
        else
            stmt();
    }
```

[15] See the `bsearch()` library function in the SubC source code package for details.

```
    if (!nc) {
        if (ldflt) {
            cval[nc] = 0;
            clab[nc++] = ldflt;
        }
        else
            error("empty switch", NULL);
    }
    genjump(lb);
    genlab(ls);
    genswitch(cval, clab, nc, ldflt? ldflt: lb);
    gentext();
    genlab(lb);
    Token = scan();
    Bsp--;
}
```

The `switch_stmt()` function accepts the expression part of the `switch` statement and then delegates further compilation to `switch_block()`.

```
static void switch_stmt(void) {
    Token = scan();
    lparen();
    rexpr();
    clear();
    rparen();
    if (Token != LBRACE)
        error("'{' expected after 'switch'", NULL);
    switch_block();
}
```

The `while_stmt()` function accepts a `while` statement:

```
while_stmt := WHILE ( expr ) stmt
```

Its control structure is shown in figure 9.10. There are no surprises here.

```
static void while_stmt(void) {
    int lb, lc;

    Token = scan();
```

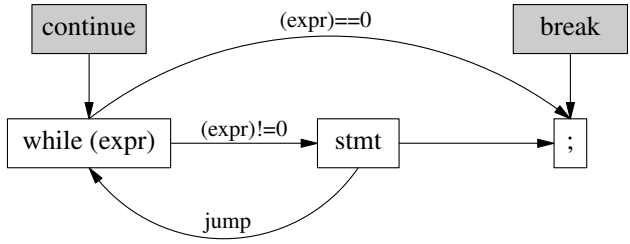


Figure 9.10: The control structure of the `while` statement

```
    pushbrk(lb = label());
    pushcont(lc = label());
    genlab(lc);
    lparen();
    rexpr();
    clear();
    genbrfalse(lb);
    rparen();
    stmt();
    genjump(lc);
    genlab(lb);
    Bsp--;
    Csp--;
}
```

The `wrong_ctx()` function reports a `case` or `default` statement in a wrong (non-`switch`) context.

```
void wrong_ctx(int t) {
        if (DEFAULT == t) {
                error("'default' not in 'switch' context",
                        NULL);
                Token = scan();
                colon();
        }
        else {
                error("'case' not in 'switch' context",
                        NULL);
                Token = scan();
                constexpr();
                colon();
        }
```

```
}
```

The `stmt()` function, finally, is the entry point of the SubC *statement parser*. It implements the following rules:

```
stmt :=
          break_stmt
        | continue_stmt
        | do_stmt
        | for_stmt
        | if_stmt
        | return_stmt
        | switch_stmt
        | while_stmt
        | compound
        | ;
        | expr ;
```

In addition to the statements accepted in this part of the parser, it also accepts *stand-alone expressions* (expressions ending with a semicolon) and empty statements (free-standing semicolons). The other empty statement, the empty compound statement `{}`, is handled in `compound()`.

The `stmt()` function also contains two *error productions* for the `case` and `default` keywords. These productions are here for the sole purpose of catching and reporting a specific error, namely an out-of-context `case` or `default`. The parser assumes that the syntax of the `case` or `default` is correct, accepts it, and then reports the error. Error productions are often used to provide better error messages and/or avoid the loss of parser synchronization.

```
void stmt(void) {
    switch (Token) {
    case BREAK:    break_stmt(); break;
    case CONTINUE: continue_stmt(); break;
    case DO:       do_stmt(); break;
    case FOR:      for_stmt(); break;
    case IF:       if_stmt(); break;
    case RETURN:   return_stmt(); break;
    case SWITCH:   switch_stmt(); break;
    case WHILE:    while_stmt(); break;
    case LBRACE:   compound(1); break;
```

```
    case SEMI:      Token = scan(); break;
    case DEFAULT:   wrong_ctx(DEFAULT); break;
    case CASE:      wrong_ctx(CASE); break;
    default:        rexpr(); semi(); break;
    }
    clear();
}
```

9.5 Declaration Parsing

The file `decl.c` contains the last part of the SubC parser, the *declaration parser*. It is the top-level part of the entire parser and accepts whole programs as its input.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Declaration parser
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

The `declarator()` function recurses.

```
int declarator(int arg, int scls, char *name, int *pprim,
               int *psize, int *pval, int *pinit);
```

The `enumdecl()` function accepts an `enum` declaration and adds the declared symbols to the global symbol table. There is no local `enum`.

```
enumdecl := ENUM { enumlist }

enumlist :=
          enumerator
        | enumerator , enumlist

enumerator :=
          IDENT
        | IDENT = constexpr
```

```
static void enumdecl(void) {
    int     v = 0;
    char    name[NAMELEN+1];

    Token = scan();
    if (IDENT == Token)
        Token = scan();
    lbrace();
    for (;;) {
        copyname(name, Text);
        ident();
        if (ASSIGN == Token) {
            Token = scan();
            v = constexpr();
        }
        addglob(name, PINT, TCONSTANT, 0, 0, v++, NULL, 0);
        if (Token != COMMA)
            break;
        Token = scan();
        if (eofcheck()) return;
    }
    match(RBRACE, "'}'");
    semi();
}
```

An "init list" is used to *initialize* an *automatically-sized* array. An automatically-sized array is a non-`extern` array with no size specified between the square brackets of its declarator, e.g.:

```
int small_primes[] = { 2, 3, 5, 7, 11 };
```

The init list is used to supply the array elements, thereby implicitly specifying the size of the array. The above array would have a size of five `int`s. Automatically-sized arrays of `char` can be initialized with string literals.

```
initlist :=
          { const_list }
        | STRLIT
```

```
const_list :=
          constexpr
        | constexpr , const_list
```

The `initlist()` function returns the number of elements found in an initializer. For string literals, this is the number of their characters plus one (for the delimiting NUL character).

```
static int initlist(char *name, int prim) {
    int     n = 0, v;
    char    buf[30];

    gendata();
    genname(name);
    if (STRLIT == Token) {
        if (PCHAR != prim)
            error("initializer type mismatch: %s", name);
        gendefs(Text, Value);
        gendefb(0);
        Token = scan();
        return Value - 1;
    }
    lbrace();
    while (Token != RBRACE) {
        v = constexpr();
        if (PCHAR == prim) {
            if (v < 0 || v > 255) {
                sprintf(buf, "%d", v);
                error("initializer out of range: %s", buf);
            }
            gendefb(v);
        }
        else {
            gendefw(v);
        }
        n++;
        if (COMMA == Token)
            Token = scan();
        else
            break;
        if (eofcheck()) return 0;
    }
    Token = scan();
```

```
    if (!n) error("too few initializers", NULL);
    return n;
}
```

The `primtype()` function turns a token into its corresponding *primitive type*.

```
int primtype(int t) {
    return t == CHAR? PCHAR:
           t == INT? PINT:
           PVOID;
}
```

The `pmtrdecl()` function accepts a *parameter declaration* within the *head* of a function declaration and adds the declared symbols to the local symbol table. After parsing the parameter list, it assigns addresses to the parameters. Note that `pmtrdecls()` passes 1 in the first argument to `declarator()`, which activates some magic in that function (pg. 140).

```
pmtrdecls :=
          ( )
        | ( pmtrlist )
        | ( pmtrlist , ... )

pmtrlist :=
          opt_primtype declarator
        | opt_primtype declarator , pmtrlist

opt_primtype :=
        | primtype
```

Also note that `pmtrdecls()` does not require a primitive type specifier in front of a parameter declarator. This is a violation of TCPL2 that has been added because the author finds it useful; it makes `f(a, b) {}` a valid declaration (which would be "K&R"—pre-ANSI—syntax) but, as a side-effect, also allows declarations like `f(a, void *p) {}`, which are not valid C, no matter which standard you refer to.

```
static int pmtrdecls(void) {
    char    name[NAMELEN+1];
```

```
int     prim, type, size, na, y, addr;
int     dummy;

if (RPAREN == Token)
    return 0;
na = 0;
for (;;) {
    if (na > 0 && ELLIPSIS == Token) {
        Token = scan();
        na = -(na + 1);
        break;
    }
    else if (IDENT == Token) {
        prim = PINT;
    }
    else {
        if (Token != CHAR && Token != INT &&
            Token != VOID
        ) {
            error("type specifier expected at: %s",
                    Text);
            Token = synch(RPAREN);
            return na;
        }
        name[0] = 0;
        prim = primtype(Token);
        Token = scan();
        if (RPAREN == Token && prim == PVOID && !na)
            return 0;
    }
    size = 1;
    type = declarator(1, CAUTO, name, &prim, &size,
                        &dummy, &dummy);
    addloc(name, prim, type, CAUTO, size, 0, 0);
    na++;
    if (COMMA == Token)
        Token = scan();
    else
        break;
}
addr = INTSIZE*2;
for (y = Locs; y < NSYMBOLS; y++) {
    addr += INTSIZE;
```

```
        Vals[y] = addr;
    }
    return na;
}
```

The `pointerto()` function promotes the type *prim* to *prim** (pointer to *prim*). It will not allow more than two levels of indirection and cannot create a pointer to a function pointer.

```
int pointerto(int prim) {
    if (CHARPP == prim || INTPP == prim || VOIDPP == prim ||
        FUNPTR == prim
    )
        error("too many levels of indirection", NULL);
    return PINT == prim? INTPTR:
           PCHAR == prim? CHARPTR:
           PVOID == prim? VOIDPTR:
           INTPTR == prim? INTPP:
           CHARPTR == prim? CHARPP: VOIDPP;
}
```

A *declarator* is what is found in declarations on the right-hand side of the *type specifier* (`int`, `void`, etc). In the declarations

```
int   n, a[17], (*fp)();
void  f(int x) {}
```

the following subsentences would be declarators: `n`, `a[17]`, `(*fp)()`, `f(int x)`, and the `x` of `f(int x)`. The `declarator()` function is a general parser for all kinds of declarators that may appear in various contexts: at the top-level, inside of function bodies, or in parameter lists of functions.

Declarators are inherently recursive, because declarators may contain function declarations and function declarations may contain declarators (in their parameter lists).

Full C declarators are an ugly mess, both from a programmer's as well as from a compiler-writer's point of view. Decomposing full declarators is virtually impossible without creating an AST. SubC limits declarators to the following rules:

```
declarator :=
          IDENT
        | * IDENT
        | * * IDENT
        | * IDENT [ constexpr ]
        | IDENT [ constexpr ]
        | IDENT = constexpr
        | IDENT [ ] = initlist
        | IDENT pmtrdecl
        | IDENT [ ]
        | * IDENT [ ]
        | ( * IDENT ) ( )
```

The `declarator()` function expects the following arguments:

pmtr	flag: the declarator is a function parameter
scls	the storage class of the declarator
name	a pointer to the name being declared
pprim	a pointer to the primitive type of the declared object
psize	a pointer to the size of the object
pval	a pointer to the initial value of the object
pinit	a pointer to the "initialized flag" of the object

The function implements a rather large set of rules and performs quite a bit of semantic analysis. It returns the *meta type* of the declared object and also fills in the *name*, *pprim*, *psize*, *pval*, and *pinit* fields.

It accepts two levels of indirection via `*`, but when declaring a *pointer-pointer* with two `*` prefixes, it will not accept an array declaration (because an array of pointer-pointers would be an instance of triple indirection). Function pointers with the syntax `(*name)()` are accepted as a special case.

All *atomic* (non-array) variables may be initialized with a constant expression, but pointers may only be initialized with 0.

When the name of the declared object is followed by an opening parenthesis, the declarator is assumed to be a function definition or prototype. In this case `declarator()` accepts a parameter list via `pmtrdecls()` and returns `TFUNCTION` immediately.

Finally the declarator parser accepts various combinations of array declarations (using square brackets) and initialization lists.

In global declarations, an array must be either `extern` or have an explicit size or an initialization list, e.g.

```
int         a1[5];
int         a2[] = { 1,2,3,4,5 };
extern int a3[];
```

When a size is supplied in an `extern` declaration, it is accepted but otherwise ignored.

In function parameters, an array size may not be specified (in violation of TCPL2) and the construct `f(int x[])` is perfectly equal to `f(int *x)` (conforming to TCPL2). This is why the following two declarations are equivalent:

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

However, the statements

```
extern char *env[];
```

and

```
extern char **env;
```

are *not* equivalent, because outside of parameter lists array declarations are not equal to pointer declarations. The first declaration of `env` declares *the address of an array of pointers to* `char`, while the second one declares a *pointer to an array of pointers to* `char`. The difference is subtle. In the first case, compiling the factor `env` will load the *address* of `env` into the primary register and in the second case, it will load the *value* of `env` into the register. So when `env` is defined as an array of pointers and later declared as an `extern` pointer to pointer, loading `env` will actually load `*env` and `*env` will actually load `**env`, etc. The difference is one level of indirection, even if the levels of indirection of the two instances of `env` are formally the same. See the `primary()` function (pg. 90) in the expression parser for the odds and ends of loading arrays and pointers.

In fact the *pmtr* parameter of `declarator()` makes the function accept two different sets of sentences. It turns off initializers, function declarations, and array sizes and activates the conversion of `x[]` to `*x`.

```
int declarator(int pmtr, int scls, char *name, int *pprim,
               int *psize, int *pval, int *pinit)
```

```
{
    int     type = TVARIABLE;
    int     ptrptr = 0;

    if (STAR == Token) {
        Token = scan();
        *pprim = pointerto(*pprim);
        if (STAR == Token) {
            Token = scan();
            *pprim = pointerto(*pprim);
            ptrptr = 1;
        }
    }
    else if (LPAREN == Token) {
        if (*pprim != PINT)
            error("function pointers are limited to 'int'",
                NULL);
        Token = scan();
        *pprim = FUNPTR;
        match(STAR, "(*name)()");
    }
    if (IDENT != Token) {
        error("missing identifier at: %s", Text);
        name[0] = 0;
    }
    else {
        copyname(name, Text);
        Token = scan();
    }
    if (FUNPTR == *pprim) {
        rparen();
        lparen();
        rparen();
    }
    if (!pmtr && ASSIGN == Token) {
        Token = scan();
        *pval = constexpr();
        if (*pval && !inttype(*pprim))
            error("non-null pointer initialization", NULL);
        *pinit = 1;
    }
    else if (!pmtr && LPAREN == Token) {
        Token = scan();
```

```
        *psize = pmtrdecls();
        rparen();
        return TFUNCTION;
    }
    else if (LBRACK == Token) {
        if (ptrptr)
            error("too many levels of indirection: %s",
                    name);
        Token = scan();
        if (RBRACK == Token) {
            Token = scan();
            if (pmtr) {
                *pprim = pointerto(*pprim);
            }
            else {
                type = TARRAY;
                *psize = 1;
                if (ASSIGN == Token) {
                    Token = scan();
                    if (!inttype(*pprim))
                        error("initialization of"
                                " pointer array not"
                                " supported",
                                NULL);
                    *psize = initlist(name, *pprim);
                    if (CAUTO == scls)
                        error("initialization of"
                                " local arrays"
                                " not supported: %s",
                                name);
                    *pinit = 1;
                }
                else if (CEXTERN != scls) {
                    error("automatically-sized array"
                            " lacking initialization: %s",
                        name);
                }
            }
        }
        else {
            if (pmtr) error("array size not supported in "
                            "parameters: %s", name);
            *psize = constexpr();
```

```
            if (*psize <= 0) {
                error("invalid array size", NULL);
                *psize = 1;
            }
            type = TARRAY;
            rbrack();
        }
    }
    if (PVOID == *pprim)
        error("'void' is not a valid type: %s", name);
    return type;
}
```

The `signature()` function turns the local *symbol table* entries from *from* to *to* into a function signature for the function with the symbol table slot *fn*. When the function does not already have a *signature*, the generated signature is added to it. Otherwise, the compiler makes sure that the generated signature matches the one already present in the function entry.

```
void signature(int fn, int from, int to) {
    char    types[MAXFNARGS+1];
    int     i;

    if (to - from > MAXFNARGS)
        error("too many function parameters", Names[fn]);
    for (i=0; i<MAXFNARGS && from < to; i++)
        types[i] = Prims[--to];
    types[i] = 0;
    if (NULL == Mtext[fn])
        Mtext[fn] = globname(types);
    else if (Sizes[fn] >= 0 && strcmp(Mtext[fn], types))
        error("redefinition does not match prior type: %s",
            Names[fn]);
}
```

The `localdecls()` function accepts all *local declarations* at the beginning of a *function body* at once:

```
localdecls :=
          ldecl
        | ldecl localdecls
```

```
ldecl :=
          primtype ldecl_list ;
        | STATIC ldecl_list ;
        | STATIC primtype ldecl_list ;

ldecl_list :=
          declarator
        | declarator , ldecl_list
```

It assigns stack addresses to the declared objects and collects automatic (non-static) initializer values in the `LIaddr[]` and `LIval[]` arrays for later code generation. Local addresses are aligned to machine words.

`localdecls()` returns the amount of memory to allocate in automatic storage, i.e. on the stack. Because the stack grows down, this amount is negative.

```
static int localdecls(void) {
    char    name[NAMELEN+1];
    int     prim, type, size, addr = 0, val, ini;
    int     stat;
    int     pbase, rsize;

    Nli = 0;
    while ( STATIC == Token ||
            INT == Token || CHAR == Token ||
            VOID == Token
    ) {
        stat = 0;
        if (STATIC == Token) {
            Token = scan();
            stat = 1;
            if (INT == Token || CHAR == Token ||
                VOID == Token
            ) {
                prim = primtype(Token);
                Token = scan();
            }
            else
                prim = PINT;
        }
        else {
```

```
            prim = primtype(Token);
            Token = scan();
        }
        pbase = prim;
        for (;;) {
            prim = pbase;
            if (eofcheck()) return 0;
            size = 1;
            ini = val = 0;
            type = declarator(0, CAUTO, name, &prim, &size,
                    &val, &ini);
            rsize = objsize(prim, type, size);
            rsize = (rsize + INTSIZE-1) / INTSIZE * INTSIZE;
            if (stat) {
                addloc(name, prim, type, CLSTATC, size,
                    label(), val);
            }
            else {
                addr -= rsize;
                addloc(name, prim, type, CAUTO, size,
                        addr, 0);
            }
            if (ini && !stat) {
                if (Nli >= MAXLOCINIT) {
                    error("too many local initializers",
                        NULL);
                    Nli = 0;
                }
                LIaddr[Nli] = addr;
                LIval[Nli++] = val;
            }
            if (COMMA == Token)
                Token = scan();
            else
                break;
        }
        semi();
    }
    return addr;
}
```

The `decl()` function accepts a top-level declaration (but without the type specifier part). This may be either a function definition or a list of

variable declarations:

```
decl :=
          declarator { localdecls stmt_list }
        | decl_list ;

decl_list :=
          declarator
        | declarator , decl_list
```

When the object being defined is a *function*, `decl()` accepts any local declarations at the beginning of the function body and emits code to perform the following actions at run time:

- Set up a *local context* (*call frame*);
- Allocate space for local variables;
- Initialize automatic local variables;
- Deallocate space for local symbols;
- Restore the caller's context and return.

It also compiles the remainder of the function body after emitting code to initialize automatic locals.

`decl()` also emits the name of the function and announces its name as a *public symbol* (in case the function is not `static`). When a function declaration is `extern` or ends with a semicolon, then it is assumed to be a *prototype* and no function body will be accepted. All prototypes are implicitly `extern` in SubC (but not in C).

After accepting a function declaration including its function body, if any, the `decl()` function clears all symbols from the local segment of the symbol table.

```
void decl(int clss, int prim) {
    char    name[NAMELEN+1];
    int     pbase, type, size = 0, val, init;
    int     lsize;

    pbase = prim;
    for (;;) {
        prim = pbase;
        val = 0;
        init = 0;
```

```
        type = declarator(0, clss, name, &prim, &size, &val,
                          &init);
        if (TFUNCTION == type) {
            if (SEMI == Token) {
                Token = scan();
                clss = CEXTERN;
            }
            Thisfn = addglob(name, prim, type, clss, size, 0,
                    NULL, 0);
            signature(Thisfn, Locs, NSYMBOLS);
            if (clss != CEXTERN) {
                lbrace();
                lsize = localdecls();
                gentext();
                if (CPUBLIC == clss) genpublic(name);
                genname(name);
                genentry();
                genstack(lsize);
                genlocinit();
                Retlab = label();
                compound(0);
                genlab(Retlab);
                genstack(-lsize);
                genexit();
                if (O_debug & D_LSYM)
                    dumpsyms("LOCALS: ", name, Locs,
                        NSYMBOLS);
            }
            clrlocs();
            return;
        }
        if (CEXTERN == clss && init) {
            error("initialization of 'extern': %s", name);
        }
        addglob(name, prim, type, clss, size, val, NULL,
                init);
        if (COMMA == Token)
            Token = scan();
        else
            break;
    }
    semi();
```

```
}
```

The `top()` function, finally, accepts any kind of *top-level declaration* and takes appropriate action. It implements the following rules:

```
top :=
          enumdecl
        | decl
        | primtype decl
        | storclass decl
        | storclass primtype decl

storclass :=
          EXTERN
        | STATIC
```

By calling `top()` repeatedly until the scanner returns *EOF* an entire C program is accepted and compiled to assembly language.

```
void top(void) {
    int prim, clss = CPUBLIC;

    switch (Token) {
    case EXTERN:    clss = CEXTERN; Token = scan(); break;
    case STATIC:    clss = CSTATIC; Token = scan(); break;
    }
    switch (Token) {
    case ENUM:
        enumdecl();
        break;
    case CHAR:
    case INT:
    case VOID:
        prim = primtype(Token);
        Token = scan();
        decl(clss, prim);
        break;
    case IDENT:
        decl(clss, PINT);
        break;
    default:
        error("type specifier expected at: %s", Text);
```

```
        Token = synch(SEMI);
        break;
    }
}
```

Chapter 10

Preprocessing

The built-in preprocessor of the SubC compiler is contained in the file `prep.c`. Preprocessor command execution is initiated by the scanner. The scanner also expands (replaces by their values) the identifiers defined by the preprocessor. This module just handles the preprocessor commands themselves.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Preprocessor
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
```

The `playmac()` function starts the replay of the macro s, that is, it feeds the characters of s to the *scanner*. The scanner is responsible for extracting characters from `Macp[]` and stopping the replay when the end of the macro text is reached.

```
void playmac(char *s) {
    if (Mp >= MAXNMAC) fatal("too many nested macros");
    Macc[Mp] = next();
    Macp[Mp++] = s;
}
```

The `defmac()` function parses a `#define` command and defines a new macro. It uses the `scanraw()` function to scan the name of the macro,

because using `scan()` would expand the macro to its *text* (value) when it already is defined.

Because the entire rest of the line constitutes the text of the new macro, it is read with a single `fgets()` without scanning it. Macro texts are tokenized when they are fed to the scanner at macro expansion time.

Note that macro redefinition is only allowed when the existing macro text is identical to the new one.

```
static void defmac(void) {
    char    name[NAMELEN+1];
    char    buf[TEXTLEN+1], *p;
    int     k, y;

    Token = scanraw();
    if (Token != IDENT)
        error("identifier expected after '#define': %s",
                Text);
    copyname(name, Text);
    if ('\n' == Putback)
        buf[0] = 0;
    else
        fgets(buf, TEXTLEN-1, Infile);
    k = strlen(buf);
    if (k) buf[k-1] = 0;
    for (p = buf; isspace(*p); p++)
        ;
    if ((y = findmac(name)) != 0) {
        if (strcmp(Mtext[y], buf))
            error("macro redefinition: %s", name);
    }
    else {
        addglob(name, 0, TMACRO, 0, 0, 0, globname(p), 0);
    }
    Line++;
}
```

The `undef()` function handles the "undefinition" of a macro via `#undef`. It does not remove entries from the global symbol table, but just changes their names to "`#undef'd`", which cannot be a valid C identifier. We do not recycle `#undef`-ined symbol table slots, assuming that `#undef` is a rare operation and the little space freed up by such an approach would not be worth the extra effort.

```
static void undef(void) {
    char    name[NAMELEN+1];
    int     y;

    Token = scanraw();
    copyname(name, Text);
    if (IDENT != Token)
        error("identifier expected after '#undef': %s",
                Text);
    if ((y = findmac(name)) != 0)
        Names[y] = "#undef'd";
}
```

The `include()` function includes files via `#include`. Like `defmac()` it extracts its parameter using `fgets()`. When the include file name is delimited by angle brackets, it loads the include file from the path

SCCDIR`/include/`*file*

where *file* is the file name extracted from the parameter.

Processing of the include file commences as follows:

- the include level (`Inclev`) is incremented;
- the current *put-back* character is saved;
- the line number and file name information are saved;
- the line number is reset to 1;
- the file name is changed to the name of the include file;
- the content of the include file is fed to the declaration parser;
- line number and file name information are restored;
- the saved put-back character is restored;
- the include level is decremented.

```
static void include(void) {
    char    file[TEXTLEN+1], path[TEXTLEN+1];
    int     c, k;
    FILE    *inc, *oinfile;
    char    *ofile;
    int     oc, oline;

    if ((c = skip()) == '<')
        c = '>';
    fgets(file, TEXTLEN-strlen(SCCDIR)-9, Infile);
```

```
    Line++;
    k = strlen(file);
    file[k-1] = 0;
    if (file[k-2] != c)
        error("missing delimiter in '#include'", NULL);
    file[k-2] = 0;
    if (c == '"')
        strcpy(path, file);
    else {
        strcpy(path, SCCDIR);
        strcat(path, "/include/");
        strcat(path, file);
    }
    if ((inc = fopen(path, "r")) == NULL)
        error("cannot open include file: %s", path);
    else {
        Inclev++;
        oc = next();
        oline = Line;
        ofile = File;
        oinfile = Infile;
        Line = 1;
        putback('\n');
        File = path;
        Infile = inc;
        Token = scan();
        while (XEOF != Token)
            top();
        Line = oline;
        File = ofile;
        Infile = oinfile;
        fclose(inc);
        putback(oc);
        Inclev--;
    }
}
```

The `ifdef()` function handles the `#ifdef` and `#ifndef` commands. The *expected* parameter specifies the expected result of the symbol table lookup (non-zero = found, zero = not found), so *expected* = 0 implements `#ifndef` and *expected* = 1 implements `#ifdef`.

The function puts `P_IFDEF` (the `#ifdef` token) on the *ifdef stack* to signal that its condition was satisfied. Otherwise it pushes `P_IFNDEF`.

```
static void ifdef(int expect) {
    char    name[NAMELEN+1];

    if (Isp >= MAXIFDEF)
        fatal("too many nested #ifdef's");
    Token = scanraw();
    copyname(name, Text);
    if (IDENT != Token)
        error("identifier expected in '#ifdef'", NULL);
    if (frozen(1))
        Ifdefstk[Isp++] = P_IFNDEF;
    else if ((findmac(name) != 0) == expect)
        Ifdefstk[Isp++] = P_IFDEF;
    else
        Ifdefstk[Isp++] = P_IFNDEF;
}
```

The `p_else()` function reverses the condition code on the top of the `#ifdef` stack. It also reports out-of-context `#else` commands. The function maps `P_IFDEF` to `P_ELSENOT` and `P_IFNDEF` to `P_ELSE`. It has to invent the virtual[1] `P_ELSENOT` token in order to make subsequent appearances of `#else` invalid when they appear in the same `#ifdef`/`#ifndef` context.

The `p_else()` function does nothing when an `#else` appears in a *frozen context*, i.e. when there is an outer `#ifdef` or `#ifndef` whose condition is not satisfied.

```
static void p_else(void) {
    if (!Isp)
        error("'#else' without matching '#ifdef'", NULL);
    else if (frozen(2))
        ;
    else if (P_IFDEF == Ifdefstk[Isp-1])
        Ifdefstk[Isp-1] = P_ELSENOT;
    else if (P_IFNDEF == Ifdefstk[Isp-1])
        Ifdefstk[Isp-1] = P_ELSE;
    else
```

[1] A token is "virtual" when it has no corresponding textual representation, so it cannot be generated by the scanner.

```
        error("'#else' without matching '#ifdef'", NULL);
}
```

The `endif()` function handles `#endif` commands by simply removing the topmost element from the ifdef stack.

```
static void endif(void) {
    if (!Isp)
        error("'#endif' without matching '#ifdef'", NULL);
    else
        Isp--;
}
```

The `junkln()` function reads a line from the input file and throws it away.

```
static void junkln(void) {
    while (!feof(Infile) && fgetc(Infile) != '\n')
        ;
    Line++;
}
```

The `frozen()` function checks whether the *depth*'th element (counting from the top of the `#ifdef` stack) is a frozen context.

```
int frozen(int depth) {
    return Isp >= depth &&
        (P_IFNDEF == Ifdefstk[Isp-depth] ||
         P_ELSENOT == Ifdefstk[Isp-depth]);
}
```

The `preproc()` function is the preprocessor interface. It executes the preprocessor command that follows in the input stream of the compiler. Because the scanner has already consumed the "`#`" character when identifying the preprocessor command, the first thing this function does is to put it back and scan the preprocessor command.

When the command is a `#define`, `#include`, or `#undef` command in a frozen context, it just skips the command and returns.

Otherwise it dispatches the command according to the preprocessor keyword scanned initially. When the scanned token is not a valid preprocessor

command, `preproc()` simply discards the current line and returns. Invalid preprocessor commands are reported by the scanner.

```
void preproc(void) {
    putback('#');
    Token = scanraw();
    if (frozen(1) &&
        (P_DEFINE == Token || P_INCLUDE == Token ||
         P_UNDEF == Token)
    ) {
        junkln();
        return;
    }
    switch (Token) {
    case P_DEFINE:  defmac(); break;
    case P_UNDEF:   undef(); break;
    case P_INCLUDE: include(); break;
    case P_IFDEF:   ifdef(1); break;
    case P_IFNDEF:  ifdef(0); break;
    case P_ELSE:    p_else(); break;
    case P_ENDIF:   endif(); break;
    default:        junkln(); break;
                    break;
    }
}
```

Chapter 11

Code Generation

11.1 A Brief Theory of Code Generation

While the previous chapters of this book dealt with the compiler front-end, we are now moving to the *back-end*, the part of the compiler that emits code for the *target platform*. The target platform of a compiler is basically defined by two factors: the *processor* on which the emitted code will run and the *operating system* that will form the *runtime environment* of the object program.

However, there are lots of things that can be done in a portable way in the back-end even before one single line of object code is generated. Note that "*object code*" in compiler-speak does not necessarily mean what is commonly called an "object file". The object language may be anything from human-readable output in the case of a source code analysis tool to a stand-alone binary executable to be flashed to the non-volatile memory chip of some mobile device. The various relocatable object file formats, like OMAGIC, COFF, OMF, or ELF are just a tiny fraction of possible object languages.

What is common to all formal language outputs, though, is that they are just another program, only in a different—and typically more low-level—language. The most common output file formats these days are C (which in this case takes the role of a "portable assembler"), various forms of assembly language source code, and relocatable object file formats like those listed above. The SubC compiler back-end discussed in this chapter will emit assembly language source code.

But even this is a detail that can be filled in later. Viewed in a more abstract way, a code generator is no more than a collection of functions that generate fragments of target code to perform tasks that are described

in the source program being analyzed by the compiler front-end. (We could even write a C-to-C compiler, implementing a "virtual processor" as a C program.)

A very simple code generator would just consist of a set of functions like `genadd()` and `genmul()`, which would emit instructions to perform tasks like adding or multiplying numbers. This is the kind of code generator that will be described in this chapter. More elaborate approaches will be discussed in chapter 16 (pg. 281ff).

No matter which approach we take, what we need first is a *model* of the target platform. We will assume that our target has a single register on which most of the operations will be performed. We call this register the *primary register* or *accumulator*. The accumulator-based model is, of course, an over-simplification these days, because it does not make use the large register sets that are typically present in modern CPUs. However, it is very simple model and can even be tweaked to generate quite efficient code. More on this later, though.

Our model will assume a CPU with the following features:

- a word-sized primary register A;
- a word-sized auxiliary register X;
- a push down stack;
- a stack pointer register S;
- a frame pointer register F;
- instructions for the usual arithmetic operations that work on A;
- a jump instruction with unlimited range;
- conditional jump instructions;
- register/register addressing;
- register/memory addressing;
- register/immediate addressing;
- register/indirect register addressing.

The number of bits per *machine word* specifies the *natural word size* of the target machine. Typical word sizes are 8 bits for the Z80 processor, 16 bits for the 8086 processor, 32 bits for the 386 processor, and 64 bits for the AXP 21164 (Alpha) and the x86-64 processors. The size of a machine word is also the typical size of the operands of arithmetic instructions. The A and X registers must be large enough to hold a machine word. SubC's `int` type maps directly to a machine word.

A *push-down stack* is a stack whose *pointer* (S) is *decremented* when pushing an element onto the stack. The push-down stack does not have to

be implemented in hardware, but since a hardware stack exists, we will use it.

A *frame pointer* is a register that points to the call frame of the *active* (i.e. most recently called and not yet returned) function. Some processors offer a special frame pointer register, but any general-purpose register will do otherwise.

A *jump instruction* with unlimited range can reach any point in the entire program space with a single "jump". Conditional jump instructions do not need unlimited range, because they can be combined with unconditional jump instructions that *do* have unlimited range, e.g.

```
      je foo     # jump on equal to 'foo'
```

could be emitted as

```
      jne tmp1    # jump on NOT equal to 'tmp1'
      jmp foo     # jump to foo
tmp1:
```

We also assume that various *addressing modes* are present on the target machine. "Register/register" addressing means to move machine words between registers. "Register/memory" denotes the transfer of machine words from memory to registers and vice versa. We also need instructions for moving single bytes to/from memory. "Register/immediate" mode combines a value in a register with an operand that is part of the instruction itself, for example for subtracting a constant value from a register. "Register/indirect" mode, finally, means to load or store machine words or bytes from/to addresses contained in a register.

This is basically it. In the SubC compiler, the code generator is called whenever the compiler has enough information for emitting a specific sequence of instructions. For example, when a global integer variable named `foo` appears in an rvalue, it will emit code to load the value of a global machine word at the location `foo`. When an integer literal appears in its input, it will emit code to load the value of that literal, etc.

Each call to the SubC code generator emits code immediately. No code synthesis takes place. This is a very simple but also a very robust approach. We will discuss code synthesis later in this book.

11.2 The Code Generator

The *code generator* is contained in the file `gen.c`. This is the portable part of the code generator. All code generator functions are in this file, while the *target description* is in the file `cg386.c`. When the compiler is ported to a different machine, the —gen.c— file should not change (at least, the required changes should be minimal).

`gen.c` includes `cgen.h`, which contains the prototypes of the functions emitting the target instructions.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Code generator (emitter)
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
#include "cgen.h"
```

`Acc` is the "accumulator loaded" flag. When this flag is "on", the value currently in the accumulator is "important" and may not be overwritten. When the accumulator is needed while `Acc` = 1, its value must be saved before it can be overwritten safely.

The compiler may call the `load()` and `clear()` functions to set the state of the accumulator accordingly.

```
int Acc = 0;

void clear(void) {
    Acc = 0;
}

void load(void) {
    Acc = 1;
}
```

Whenever a fresh *label* is required, the `label()` function can be called to generate one. Labels are numeric; the `LPREFIX` ("label prefix") character is attached to them to form a symbol (see `lgen()` and friends below).

```
int label(void) {
    static int id = 1;

    return id++;
}
```

When a value has to be loaded while all registers are in use, the value stored in one register has to be moved to a less favorable location, typically *memory*. This process is called *spilling*: the values are "spilled" to temporary locations. Spilling can be a highly complex process, but in this generator it is as simple as pushing the value of the primary register to the stack when the accumulator flag is set:

```
void spill(void) {
    if (Acc) genpush();
}
```

The following functions form the *emitter interface*. All output of the SubC compiler goes through this interface. The functions generate various forms of output:

Function	Output Format	Arguments
`genraw()`	`%s`	string
`gen()`	`\t%s\n`	string
`ngen()`	`...%d...`	number
`ngen2()`	`...%d...%d...`	number, address
`lgen()`	`...L%d...`	label ID
`lgen2()`	`...%d...L%d...`	number, label ID
`sgen()`	`...%s...`	string
`genlab()`	`L%d:`	label ID

The `genraw()` function is most general, while the output of the `gen()` function is most common, as we assume the following wide-spread assembly language format:

```
label:  instruction operands
```

where both the "label" and the "operands" are optional.

The `ngen()`, `lgen()`, and `sgen()` functions expect a custom format in the first argument, and an instruction in the second argument. For example,

```
ngen("%s\t$%d,%%eax", "movl", 49);
```

would generate the output

```
        movl    $49,%eax
```

Check the file `cg386.c` for more examples than you ever wanted to see.

```
void genraw(char *s) {
    if (NULL == Outfile) return;
    fprintf(Outfile, "%s", s);
}

void gen(char *s) {
    if (NULL == Outfile) return;
    fprintf(Outfile, "\t%s\n", s);
}

void ngen(char *s, char *inst, int n) {
    if (NULL == Outfile) return;
    fputc('\t', Outfile);
    fprintf(Outfile, s, inst, n);
    fputc('\n', Outfile);
}

void ngen(char *s, char *inst, int n, int a) {
    if (NULL == Outfile) return;
    fputc('\t', Outfile);
    fprintf(Outfile, s, inst, n, a);
    fputc('\n', Outfile);
}

void lgen(char *s, char *inst, int n) {
    if (NULL == Outfile) return;
    fputc('\t', Outfile);
    fprintf(Outfile, s, inst, LPREFIX, n);
    fputc('\n', Outfile);
}

void lgen2(char *s, int v1, int v2) {
    if (NULL == Outfile) return;
    fputc('\t', Outfile);
    fprintf(Outfile, s, v1, LPREFIX, v2);
```

```
    fputc('\n', Outfile);
}

void sgen(char *s, char *inst, char *s2) {
    if (NULL == Outfile) return;
    fputc('\t', Outfile);
    fprintf(Outfile, s, inst, s2);
    fputc('\n', Outfile);
}

void genlab(int id) {
    if (NULL == Outfile) return;
    fprintf(Outfile, "%c%d:", LPREFIX, id);
}
```

The `labname()` function returns a symbol representing a label.

```
char *labname(int id) {
    static char name[100]

    sprintf(name, "%c%d", LPREFIX, id);
    return name;
}
```

The `gsym()` function returns an *external name* for a given internal name, e.g. it turns `main` into `Cmain` (given that `PREFIX` is `'C'`).

```
char *gsym(char *s) {
    static char     name[NAMELEN+2];

    name[0] = PREFIX;
    name[1] = 0;
    strcat(name, s);
    return name;
}
```

11.3 Framework

The `gendata()` and `gentext()` functions select the output segment by emitting a corresponding assembler directive. Program data go to the *data segment* and instructions go to the *text segment* (a.k.a. *code segment*). We

memorize the currently selected segment to avoid the emission of duplicate directives.

```
void gendata(void) {
    if (Textseg) cgdata();
    Textseg = 0;
}

void gentext(void) {
    if (!Textseg) cgtext();
    Textseg = 1;
}
```

The `genprelude()` and `genpostlude()` functions perform the set-up and shutdown of the back-end, and they are the first and last code generator functions that should be called by the compiler.

Small sections of boiler-plate code required by the system assembler should be added to the corresponding functions in the back-end description (`cgprelude()`, `cgpostlude()`).

```
void genprelude(void) {
    Textseg = 0;
    gentext();
    cgprelude();
}

void genpostlude(void) {
    cgpostlude();
}
```

The `genname()` function emits a *symbol* name (not a label name!). The `genpublic()` function emits an announcement that makes the given name "public", i.e. visible outside of the current module.

```
void genname(char *name) {
    genraw(gsym(name));
    genraw(":");
}

void genpublic(char *name) {
    cgpublic(gsym(name));
}
```

11.4 Load Operations

The `genaddr()` function loads the address of the object described by symbol table entry y into the primary register. Different instructions have to be emitted for *automatic* objects which reside on the stack, *local static* objects which are identified by labels, and *public*, *static*, and *external* objects which are referenced by their names.

All instructions call `gentext()` first, and all load instruction will spill the accumulator—if necessary—before loading a value.

```
void genaddr(int y) {
    gentext();
    spill();
    if (CAUTO == Stcls[y])
        cgldla(Vals[y]);
    else if (CLSTATC == Stcls[y])
        cgldsa(Vals[y]);
    else
        cgldga(gsym(Names[y]));
    load();
}
```

The `genldlab()` function loads the address of a *label*. In the subsequent text the term "loading" always implies the primary register unless stated otherwise.

```
void genldlab(int id) {
    gentext();
    spill();
    cgldlab(id);
    load();
}
```

The `genlit()` function loads a constant value.

```
void genlit(int v) {
    gentext();
    spill();
    cglit(v);
    load();
}
```

The `genargc()` function loads the number of arguments passed to the active function.

```
void genargc(void) {
    gentext();
    spill();
    cgargc();
    load();
}
```

11.5 Binary Operators

The `genand()`, `genior()`, and `genxor()` functions emit code for the binary "and", "(inclusive) or", and "exclusive or" operations.

At this point we shall have a closer look at our computational model. When the compiler finds a *binary operator*, like "`&`", it knows that the left-hand side of the operator is an rvalue and emits code to load that value to the primary register. It then proceeds to parse the right-hand side of the operator, which must also be an rvalue. So the statement `a&b;` will generate the following code up to (but not including) the semicolon:

- load a to the primary register;
- spill A (push);
- load b to the primary register.

When the semicolon is finally parsed, the operator stack in `binexpr()` (pg. 107) is flushed and eventually the `genand()` function is called to generate the "binary and" fragment. At this point the first operand a is on the top of the stack and the second operand b is in the primary register. Because the binary "and" is a commutative operation, we can simply fetch the first operand from the stack and "and" it to the second one:

- pop X;
- perform $A \mathbin{\&}= X$.

This is exactly what the `genand()` function accomplishes: `cgpop2()` emits code to pop the *top of the stack* (TOS) into the auxiliary register X and the code generated by `cgand()` performs a binary "and" on X and A, leaving the result in A. All binary operator functions basically work in this way.

```
void genand(void) {
    gentext();
    cgpop2();
    cgand();
}

void genior(void) {
    gentext();
    cgpop2();
    cgior();
}

void genxor(void) {
    gentext();
    cgpop2();
    cgxor();
}
```

However, the `genshl()` and `genshr()` functions, which handle arithmetic left and right *shift operations*, cannot simply perform $A <<= X$ and $A >>= X$, respectively, because the order of the operands does matter in these operations.

Hence these generator functions insert a *swap* operation that exchanges the contents of the X and A registers. So the `a<<b;` statement would result in the following code:

- load a;
- spill A;
- load b;
- pop X;
- swap X and A;
- perform $A <<= X$.

We will later (pg. 184ff) discover, though, that there are situations where the operands already are in the correct order. This is why functions that implement non-commutative operations take an additional argument, *swap*, which tells them whether or not to swap their operands.

```
void genshl(int swap) {
    gentext();
```

```
    cgpop2();
    if (swap) cgswap();
    cgshl();
}

void genshr(int swap) {
    gentext();
    cgpop2();
    if (swap) cgswap();
    cgshr();
}
```

The `ptr()` function returns truth when its argument specifies a *pointer* type and `needscale()` returns truth when the type passed to it needs *scaling* during pointer arithmetics. The `char*` type does not need scaling because it points to byte-sized objects. The same applies to function pointers (although this might be controversial.)

```
static int ptr(int p) {
    return INTPTR == p || INTPP == p ||
           CHARPTR == p || CHARPP == p ||
           VOIDPTR == p || VOIDPP == p ||
           FUNPTR == p;
}

static int needscale(int p) {
    return INTPTR == p || INTPP == p || CHARPP == p ||
           VOIDPP == p
```

The `genadd()` function emits code for adding two numbers. This is not as straight-forward as one might think, because C can do *pointer arithmetics*. When p is a pointer and n is an integer, then $p + n$ and $n + p$ would in fact result in $p + n{*}sizeof(*p)$, where $*p$ is the type to which p points. For the type `char*`, its size would be `sizeof(char)` (one), but for `int*` and the various of pointer-pointers, the argument n would have to be "scaled" to a multiple of $sizeof(*p)$.

In addition, the *type* resulting from a pointer addition is the type of the pointer rather than `int`. The `genadd()` function has to take care of these details. It accepts the types of its operands in the $p1$ and $p2$ parameters and returns the type of the result.

```
int genadd(int p1, int p2) {
    int rp = PINT;

    gentext();
    cgpop2();
    if (ptr(p1)) {
        if (needscale(p1)) cgscale();
        rp = p1;
    }
    else if (ptr(p2)) {
        if (needscale(p2)) cgscale2();
        rp = p2;
    }
    cgadd();
    return rp;
}
```

An integer n can be subtracted from a pointer p, giving $p - n*sizeof(*p)$ (but a pointer may not be subtracted from an integer). The `gensub()` function handles this case. C may also subtract two pointers (p, q) of the same type, resulting in $(p-q)/sizeof(*p)$. In this case, the *result* has to be "scaled back" or "unscaled". This case is also covered by `gensub()`.

```
int gensub(int p1, int p2, int swap) {
    int rp = PINT;

    gentext();
    cgpop2();
    if (swap) cgswap();
    if (!inttype(p1) && !inttype(p2) && p1 != p2)
        error("incompatible pointer types in binary '-'",
                NULL);
    if (ptr(p1) && !ptr(p2)) {
        if (needscale(p1)) cgscale2();
        rp = p1;
    }
    cgsub();
    if (needscale(p1) && needscale(p2))
        cgunscale();
    return rp;
}
```

`genmul()`, `gendiv()`, and `genmod()` implement the multiplication, integer division, and modulo operations, respectively. Nothing special to see here.

```
void genmul(void) {
    gentext();
    cgpop2();
    cgmul();
}

void gendiv(int swap) {
    gentext();
    cgpop2();
    if (swap) cgswap();
    cgdiv();
}

void genmod(int swap) {
    gentext();
    cgpop2();
    if (swap) cgswap();
    cgmod();
}
```

The `binopchk()` function type-checks the operation *p1 op p2*, where *p*1 and *p*2 are operand types and *op* is a token indicating an arithmetic operation. *Op* may denote a combined assignment operation, like "`+=`", as well.

All binary operations accept two integer types as their operands. In the case of "`+`" one of the operands may be a pointer and in "`-`" operations the first operand or both operands may be a pointer. Comparative operations can be applied to pointers of the same type or to a pointer of any type as long as the other operand is a *void pointer*.

```
static void binopchk(int op, int p1, int p2) {
    if (ASPLUS == op)
        op = PLUS;
    else if (ASMINUS == op)
        op = MINUS;
    if (inttype(p1) && inttype(p2))
        return;
    if (PLUS == op && (inttype(p1) || inttype(p2)))
```

```
        return;
    if (MINUS == op && (!inttype(p1) || inttype(p2)))
        return;
    if ((EQUAL == op || NOTEQ == op || LESS == op ||
         GREATER == op || LTEQ == op || GTEQ == op)
        &&
        (p1 == p2 ||
         VOIDPTR == p1 && !inttype(p2) ||
         VOIDPTR == p2 && !inttype(p1))
    )
        return;
    error("invalid operands to binary operator", NULL);
}
```

The `genbinop()` function emits code for the operation *p1 op p2*.

```
int genbinop(int op, int p1, int p2) {
    binopchk(op, p1, p2);
    switch (op) {
    case PLUS:    return genadd(p1, p2);
    case MINUS:   return gensub(p1, p2, 1);
    case STAR:    genmul(); break;
    case SLASH:   gendiv(1); break;
    case MOD:     genmod(1); break;
    case LSHIFT:  genshl(1); break;
    case RSHIFT:  genshr(1); break;
    case AMPER:   genand(); break;
    case CARET:   genxor(); break;
    case PIPE:    genior(); break;
    case EQUAL:   cgeq(); break;
    case NOTEQ:   cgne(); break;
    case LESS:    cglt(); break;
    case GREATER: cggt(); break;
    case LTEQ:    cgle(); break;
    case GTEQ:    cgge(); break;
    }
    return PINT;
}
```

11.6 Unary Operators

All *unary operators* are quite straight-forward. They simply apply their corresponding instruction sequences to the primary register (or, in the case of `genscale2()`, to the auxiliary register). No spilling is ever necessary. Here is a summary of unary operations:

`genbool()`	Normalize the truth value in A
`genind()`	Perform indirection through A
`genlognot()`	Perform a "logical not" on A
`genneg()`	Invert the sign of A
`gennot()`	Invert all bits of A
`genscale()`	Scale A (multiply by machine word size)
`genscale2()`	Scale X (multiply by machine word size)

Indirection means that the value pointed to by A will be loaded into A. `genind()` takes an argument that indicates the type of its operand. When its type is *PCHAR*, the function will load a byte, otherwise it will load a machine word.

"*Normalizing* a truth value" means to turn any non-zero value into one and to keep zero. The "*logical not*" turns zero into one and non-zero into zero. These operations are quite similar and some machines may implement one on top of the other. Clever sequences for these operations will be defined in the target description (pg. 199).

```
void genbool(void) {
    gentext();
    cgbool();
}

void genind(int p) {
    gentext();
    if (PCHAR == p)
        cgindb();
    else
        cgindw();
}

void genlognot(void) {
    gentext();
    cglognot();
}
```

```
void genneg(void) {
    gentext();
    cgneg();
}

void gennot(void) {
    gentext();
    cgnot();
}

void genscale(void) {
    gentext();
    cgscale();
}

void genscale2(void) {
    gentext();
    cgscale2();
}
```

11.7 Jumps and Function Calls

The `genjump()` function emits code that jumps to the label *dest*. Likewise, `cgbrfalse()` generates a conditional "*branch-on-false*" (branch when $A = 0$) and `cgbrtrue()` generates a "*branch-on-true*" (branch when $A \neq 0$).

```
void genjump(int dest) {
    gentext();
    cgjump(dest);
}

void genbrfalse(int dest) {
    gentext();
    cgbrfalse(dest);
}

void genbrtrue(int dest) {
    gentext();
    cgbrtrue(dest);
}
```

The `gencall()` and `gencalr()` functions generate direct and indirect *function calls*. A direct function call, as emitted by `gencall()`, is a call to the function described by the symbol table slot y. Functions are referenced by their names. An indirect call is generated by `gencalr()` ("call through register"); it performs a call to the function whose address is contained in A.

```
void gencall(int y) {
    gentext();
    cgcall(gsym(Names[y]));
    load();
}

void gencalr(void) {
    gentext();
    cgcalr();
    load();
}
```

The `genentry()` and `genexit()` functions emit code to establish and remove a function call frame. The "entry" function generates code that

- pushes the frame pointer F;
- loads F with S, thereby creating a new call frame.

The "exit" function restores the previous call frame by popping F off the stack again.

```
void genentry(void) {
    gentext();
    cgentry();
}

void genexit(void) {
    gentext();
    cgexit();
}
```

The code emitted by `genpush()` pushes the accumulator onto the stack; `genpushlit()` pushes an integer value directly onto the stack. It is used to push argument counts, which are known at compile time. `genpushlit()`

calls `spill()` to make sure that the current accumulator value (if any) is pushed *before* its own argument. The `genpushlit()` function is equal to `genlit()` followed by `genpush()`, but allows to push an immediate operand on machines that support it.

```
void genpush(void) {
    gentext();
    cgpush();
}

void genpushlit(int n) {
    gentext();
    spill();
    cgpushlit(n);
}
```

The `genstack()` function generates code that moves the stack pointer, thereby allocating or releasing memory on the stack. Because we assume a *push-down stack*, *negative* values of n will allocate and positive values will release memory. `genstack()` is also used to remove function arguments after a call.

```
void genstack(int n) {
    if (n) {
        gentext();
        cgstack(n);
    }
}
```

The `genlocinit()` function emits code to store the initial values contained in the `LIval[]` array in the local addresses in the `LIaddr[]` array. `cginitlw()` could be implemented as a sequence of `cglit()` and `cgstorlw()` ("store local word") calls, but it allows to store an immediate value directly in a local storage location on machines that support this address combination.

```
void genlocinit(void) {
    int i;

    gentext();
    for (i=0; i<Nli; i++)
```

```
        cginitlw(LIval[i], LIaddr[i]);
}
```

11.8 Data Definitions

The generator functions in this section emit the declarations of public, static, and local static data objects, which are, from the assembler's point of view, just machine words, bytes, or sequences of words or bytes.

Arrays are uninitialized storage blocks that are located in the *BSS* segment. The `genbss()` function names and allocates a block of storage of the given size.

```
void genbss(char *name, int len) {
    gendata();
    cgbss(name, len);
}
```

The `gendefb()` function declares a byte and `gendefl()`, `gendefp()`, and `gendefw()` declare a machine word holding the address of a *label*, a *pointer* object, and a generic *machine word*, respectively. All of these functions with the exception of `gendefb()` actually allocate and initialize machine words, just with different initialization values. The differentiation between `gendefw()` and `gendefp()` is a rather conceptual one.

```
void gendefb(int v) {
    gendata();
    cgdefb(v);
}

void gendefl(int id) {
    gendata();
    cgdefl(id);
}

void gendefp(int v) {
    gendata();
    cgdefp(v);
}

void gendefw(int v) {
```

```
    gendata();
    cgdefw(v);
}
```

The `gendefs()` function emits the characters of the string *s* which has the length *len*. The length is specified explicitly, because we may need to emit strings containing `'\0'` characters. `gendefs()` emits alpha-numeric characters in readable form to make back-end debugging easier.

```
void gendefs(char *s, int len) {
    int i;

    gendata();
    for (i=1; i<len-1; i++) {
        if (isalnum(s[i]))
            cgdefc(s[i]);
        else
            cgdefb(s[i]);
    }
}
```

11.9 Increment Operators

The *increment operators* of C may look innocent, but generating code for them is quite hairy under the hood. This is mostly due to the variety of instructions to which the "`++`" and "`--`" operators map. First, there are four *storage classes*:

- public/static, referenced by symbol;
- local static, referenced by label;
- local automatic, referenced by stack frame address;
- indirect, referenced by a register.

Then we have

- *pre-increments* (`++x`);
- *post-increments* (`x++`);
- *pre-decrements* (`--x`);
- *post-decrements* (`x--`).

Each of them can be combined with each of the storage classes, giving 16 variations. Furthermore, the operand of each of these operations can be a *pointer* that needs to be scaled or an integer type (32 variations). Finally, the integer types may be either an `int` or a `char`, giving another 16 options, resulting in 48 different instruction sequences in total.[1]

The `genincptr()` function generates code to increment pointers by adding or subtracting the size of a machine word. The *lv* parameter is the lvalue structure describing the operand, *inc* is a flag indicating increment (0 = decrement), and *pre* is a flag indicating a "pre-access" operation, i.e. when set, the operand is modified *before* extracting its value. Because pointers are always word-sized this function emits 16 different instruction sequences.

We make use of the fact that objects described by lvalue structures with a non-zero `LVSYM` field have not yet been loaded. So we can just increment them before loading when generating a pre-increment and just load them before incrementing when emitting code for a post-increment.

The only case in which this does not work occurs when we want to increment in indirect object, because the address of the indirect object will already be in the primary register at this point. For pre-increments we can simply increment the value pointed to by A in this case, but for post-operations, things are a bit more complicated:

- load X with A;
- load the object pointed to by A;
- increment the value pointed to by X.

The `cdldinc()` ("load increment pointer") function emits code to copy A to X for later use. The various `cginc...()` and `cgdec...()` functions emit specific increment instructions, like `cginc1pi()` ("pre-increment pointer indirect") and `cgdecpl()` ("decrement pointer local"). These functions will be explained in the following chapter.

```
static void genincptr(int *lv, int inc, int pre) {
    int y;

    gentext();
    y = lv[LVSYM];
    if (!y && !pre) cgldinc();
```

[1] Just imagine adding in `long`, `short`, `unsigned`, `float`, `double`, etc. to get an idea of the complexity of C code synthesis.

```
    if (!pre) rvalue(lv);
    if (!y) {
        if (pre)
            if (inc)
                cginc1pi();
            else
                cgdec1pi();
        else
            if (inc)
                cginc2pi();
            else
                cgdec2pi();
    }
    else if (CAUTO == Stcls[y]) {
        if (inc)
            cgincpl(Vals[y]);
        else
            cgdecpl(Vals[y]);
    }
    else if (CLSTATC == Stcls[y]) {
        if (inc)
            cgincps(Vals[y]);
        else
            cgdecps(Vals[y]);
    }
    else {
        if (inc)
            cgincpg(gsym(Names[y]));
        else
            cgdecpg(gsym(Names[y]));
    }
    if (pre) rvalue(lv);
}
```

The `geninc()` function handles *all* the increment operators. The first thing it does is to figure out whether its argument needs scaling and if so, it delegates further code generation to `genincptr()`. The remainder of the function works in the same way as the pointer increment generator, but it also has to consider `char` operands. All the increment operations generated by it come in a "word" and a "byte" flavor, i.e. one for incrementing `int`s and one for incrementing `char`s. For example, `cginc2ib()` would post-increment an indirect byte and `cginc2iw()` would post-increment an

indirect word. This function can generate 32 different sequences in total.

Note: computing the rvalue of an indirect `char` may move A to X, because we have to clear the "upper" bytes of A before loading the `char`:

- load X with A;
- load A with 0;
- load the least significant byte of A with the value pointed to by X.

Post-incrementing an indirect value also involves the "load X with A" operation, as shown in the description of `genincptr()`. So combining a post-increment with an indirect `char` (`(*p)++`) gives:

- load X with A (generated by `geninc()`);
- load X with A (generated by `rvalue()`);
- load A with 0;
- load the least significant byte of A with the value pointed to by X.
- increment the value pointed to by X.

Of course we could optimize this case away, but we won't. Such specific optimizations tend to make the code hard to follow and may introduce subtle errors. This is better covered by a more general approach, like code synthesis (pg. 281ff) or an emitter that filters out duplicate instructions (q.v. peephole optimization, pg. 303ff).

```
void geninc(int *lv, int inc, int pre) {
    int y, b;

    gentext();
    y = lv[LVSYM];
    if (needscale(lv[LVPRIM])) {
        genincptr(lv, inc, pre);
        return;
    }
    b = PCHAR == lv[LVPRIM];
    /* will duplicate move to aux register in (*char)++ */
    if (!y && !pre) cgldinc();
    if (!pre) rvalue(lv);
    if (!y) {
        if (pre)
            if (inc)
                b? cginc1ib(): cginc1iw();
            else
```

```
                b? cgdec1ib(): cgdec1iw();
        else
            if (inc)
                b? cginc2ib(): cginc2iw();
            else
                b? cgdec2ib(): cgdec2iw();
    }
    else if (CAUTO == Stcls[y]) {
        if (inc)
            b? cginclb(Vals[y]): cginclw(Vals[y]);
        else
            b? cgdeclb(Vals[y]): cgdeclw(Vals[y]);
    }
    else if (CLSTATC == Stcls[y]) {
        if (inc)
            b? cgincsb(Vals[y]): cgincsw(Vals[y]);
        else
            b? cgdecsb(Vals[y]): cgdecsw(Vals[y]);
    }
    else {
        if (inc)
            b? cgincgb(gsym(Names[y])):
               cgincgw(gsym(Names[y]));
        else
            b? cgdecgb(gsym(Names[y])):
               cgdecgw(gsym(Names[y]));
    }
    if (pre) rvalue(lv);
}
```

11.10 Switch Table Generation

The `genswitch()` function writes a switch table to the output. The code generated by it loads the address of the switch table to the X register and then invokes the `switch` magic (typically a small routine in the C startup module) that processes the table. At this point the `switch` expression is in A.

The remainder of `genswitch()` emits the table itself to the data segment. The switch table layout is depicted in figure 11.1.

```
void genswitch(int *vals, int *labs, int nc, int dflt) {
```

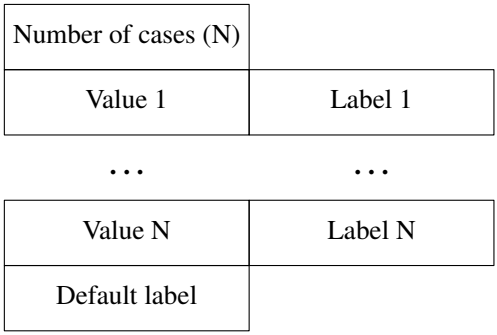


Figure 11.1: The layout of a switch table

```
    int i, ltbl;

    ltbl = label();
    gentext();
    cgldswtch(ltbl);
    cgcalswtch();
    gendata();
    genlab(ltbl);
    gendefw(nc);
    for (i = 0; i < nc; i++)
        cgcase(vals[i], labs[i]);
    gendefl(dflt);
}
```

11.11 Store Operations

The instructions emitted in this section will store values in locations, that is, they generate code for the various *assignment operators*.

The `genasop()` function generates the operation implied by a *combined assignment operator* such as "`+=`" or "`>>=`". Note that when processing a combined assignment, the operands are in memory in reverse order, because the compilation of a statement like `a-=b;` proceeds as follows:

- the address of a is held in an `LV` structure;
- the right-hand side of "`-=`" is being processed, loading A with b.

So when we want to emit code for the "`-=`" operator, the *second* operand, b, is in the primary register. The b is the second operand of the implied "`-`" because `a-=b` is short for `a=a-b`).

The next step is to load a, giving a configuration with the *second* operand (b) spilled to the stack and the *first* one (a) in the primary register. Since we actually want to subtract b from a, we do not need to swap the operands when generating a "subtract" instruction. This is why `genasop()` passes $swap = 0$ to the generator functions that emit non-commutative operations.

```
void genasop(int op, int p1, int p2) {
    binopchk(op, p1, p2);
    switch (op) {
    case ASDIV:    gendiv(0); break;
    case ASMUL:    genmul(); break;
    case ASMOD:    genmod(0); break;
    case ASPLUS:   genadd(p2, p1); break;
    case ASMINUS:  gensub(p2, p1, 0); break;
    case ASLSHIFT: genshl(0); break;
    case ASRSHIFT: genshr(0); break;
    case ASAND:    genand(); break;
    case ASXOR:    genxor(); break;
    case ASOR:     genior(); break;
    }
}
```

The `genstore()` function stores a value in the location described by the lvalue structure *lv*. When the *op* parameter is not equal to `ASSIGN` (the token of the "`=`" operator), then the function will apply the operation implied by the combined assignment token *op* to the operands *lv* and *lv2* and then store the result in *lv*.

When *lv* does not describe a symbol (but an *anonymous storage location*), then the value will be stored in the location pointed to by the TOS (top of stack). The address is at the top of the stack in this case because it had to be spilled when compiling the right-hand side of the assignment operator.

In all other cases the function will simply emit the proper instructions to store byte or word-sized values in local, local static, or public/static objects.

```
void genstore(int op, int *lv, int *lv2) {
    if (NULL == lv) return;
    gentext();
    if (ASSIGN != op) {
```

```
        if (lv[LVSYM]) rvalue(lv);
        genasop(op, lv[LVPRIM], lv2[LVPRIM]);
    }
    if (!lv[LVSYM]) {
        cgpopptr();
        if (PCHAR == lv[LVPRIM])
            cgstorib();
        else
            cgstoriw();

    }
    else if (CAUTO == Stcls[lv[LVSYM]]) {
        if (PCHAR == lv[LVPRIM])
            cgstorlb(Vals[lv[LVSYM]]);
        else
            cgstorlw(Vals[lv[LVSYM]]);
    }
    else if (CLSTATC == Stcls[lv[LVSYM]]) {
        if (PCHAR == lv[LVPRIM])
            cgstorsb(Vals[lv[LVSYM]]);
        else
            cgstorsw(Vals[lv[LVSYM]]);
    }
    else {
        if (PCHAR == lv[LVPRIM])
            cgstorgb(gsym(Names[lv[LVSYM]]));
        else
            cgstorgw(gsym(Names[lv[LVSYM]]));
    }
}
```

11.12 Rvalue Computation

The computation of an *rvalue* ("right-hand-side value") is done in the `rvalue()` function. Computing an rvalue is the inverse operation of a "store" operation. It fetches a value from a location, thereby performing one level of *indirection*.

Like `genstore()` this function has to distinguish the usual storage classes and handle indirect values. When *lv* describes an anonymous location, it assumes that indirection is to be performed through a pointer in the primary register. This is why `rvalue()` can be used to perform

pointer indirection for the "*" and "[]" operators. See the `indirection()` function for details (pg. 95).

Because `rvalue()` may change the status of the accumulator from "clear" to "loaded", it calls `load()` at the end.

```
void rvalue(int *lv) {
    if (NULL == lv) return;
    gentext();
    if (!lv[LVSYM]) {
        genind(lv[LVPRIM]);
    }
    else if (CAUTO == Stcls[lv[LVSYM]]) {
        spill();
        if (PCHAR == lv[LVPRIM]) {
            cgclear();
            cgldlb(Vals[lv[LVSYM]]);
        }
        else {
            cgldlw(Vals[lv[LVSYM]]);
        }
    }
    else if (CLSTATC == Stcls[lv[LVSYM]]) {
        spill();
        if (PCHAR == lv[LVPRIM]) {
            cgclear();
            cgldsb(Vals[lv[LVSYM]]);
        }
        else
            cgldsw(Vals[lv[LVSYM]]);
    }
    else {
        spill();
        if (PCHAR == lv[LVPRIM]) {
            cgclear();
            cgldgb(gsym(Names[lv[LVSYM]]));
        }
        else
            cgldgw(gsym(Names[lv[LVSYM]]));
    }
    load();
}
```

Chapter 12

Target Description

The file `cgen.h` contains the prototypes of all target description functions. It forms the interface between the code generator and the target description itself, which is contained in `cg386.c`.

There is nothing special to see on the following pages; feel free to skip ahead.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Code generator interface
 */

void cgadd(void);
void cgand(void);
void cgargc(void);
void cgbool(void);
void cgbrfalse(int n);
void cgbrtrue(int n);
void cgbss(char *s, int z);
void cgcall(char *s);
void cgcalr(void);
void cgcalswtch(void);
void cgcase(int v, int l);
void cgclear(void);
void cgdata(void);
void cgdec1ib(void);
void cgdec1iw(void);
void cgdec1pi(void);
void cgdec2ib(void);
void cgdec2iw(void);
```

```
void cgdec2pi(void);
void cgdecgb(char *s);
void cgdecgw(char *s);
void cgdeclb(int a);
void cgdeclw(int a);
void cgdecpg(char *s);
void cgdecpl(int a);
void cgdecps(int a);
void cgdecsb(int a);
void cgdecsw(int a);
void cgdefb(int v);
void cgdefc(int c);
void cgdefl(int v);
void cgdefp(int v);
void cgdefw(int v);
void cgdiv(void);
void cgentry(void);
void cgeq(void);
void cgexit(void);
void cgge(void);
void cggt(void);
void cginc1ib(void);
void cginc1iw(void);
void cginc1pi(void);
void cginc2ib(void);
void cginc2iw(void);
void cginc2pi(void);
void cgincgb(char *s);
void cgincgw(char *s);
void cginclb(int a);
void cginclw(int a);
void cgincpg(char *s);
void cgincpl(int a);
void cgincps(int a);
void cgincsb(int a);
void cgincsw(int a);
void cgindb(void);
void cgindw(void);
void cginitlw(int v, int a);
void cgior(void);
void cgjump(int n);
void cgldga(char *s);
void cgldgb(char *s);
```

```
void cgldgw(char *s);
void cgldinc(void);
void cgldla(int n);
void cgldlab(int id);
void cgldlb(int n);
void cgldlw(int n);
void cgldsa(int n);
void cgldsb(int n);
void cgldsw(int n);
void cgldswtch(int n);
void cgle(void);
void cglit(int v);
void cglognot(void);
void cglt(void);
void cgmod(void);
void cgmul(void);
void cgne(void);
void cgneg(void);
void cgnot(void);
void cgpop2(void);
void cgpopptr(void);
void cgpostlude(void);
void cgprelude(void);
void cgpublic(char *s);
void cgpush(void);
void cgpushlit(int n);
void cgscale(void);
void cgscale2(void);
void cgshl(void);
void cgshr(void);
void cgstack(int n);
void cgstorgb(char *s);
void cgstorgw(char *s);
void cgstorib(void);
void cgstoriw(void);
void cgstorlb(int n);
void cgstorlw(int n);
void cgstorsb(int n);
void cgstorsw(int n);
void cgsub(void);
void cgswap(void);
void cgtext(void);
void cgunscale(void);
```

```
void cgxor(void);
```

12.1 The 386 Target

Here we are, finally, down to the hardware level. The file `cg386.c` describes the various fragments that are used to generate code for the *386 processor*. When *porting* the SubC compiler to a different processor type, this is the file that has to be replaced. For instance, when creating an AXP 21164 backend, a new file (e.g.: `cgalpha.c`) would have to be created, containing the same function names, but emitting code for the Alpha instead of the 386. The SubC compiler for the Alpha would then contain this new file instead of `cg386.c`. Of course, this would only be the beginning. A lot of fine-tuning would probably follow.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  386 target description
 */

#include "defs.h"
#include "data.h"
#include "decl.h"
#include "cgen.h"
```

This file is mostly a collection of one-liners that describe brief fragments of machine code with very specific meanings. The meaning of each individual fragment is designed to be simple, intending to make the adaption of the target description to different *processors* as easy as possible.

12.2 Framework

This section contains general assembler directives.

`cgdata` — Subsequent output will describe data.

```
void cgdata(void)       { gen(".data"); }
```

`cgtext` — Subsequent output will be instructions.

Be careful to place labels in the proper segments, too! Consider:

The left side describes a label `L1` in the data segment that points to a machine word with the value 0. In the right side the label `L1` points to an unknown location in the text segment, because the `.data` directive appears *after* the label.

```
void cgtext(void)       { gen(".text"); }
```

`cgprelude` — Boilerplate instructions and directives.

```
void cgprelude(void)    { }
```

`cgpostlude` — Closing instructions and directives.

```
void cgpostlude(void)   { }
```

`cgpublic` — Announce s as a public name.

```
void cgpublic(char *s)  { ngen(".globl\t%s", s, 0); }
```

12.3 Load Operations

Operations that load values into the primary register.

`cglit` — Load literal (immediate) value v.

```
void cglit(int v)       { ngen("%s\t$%d,%%eax", "movl", v); }
```

`cgclear` — Load zero.

```
void cgclear(void)      { gen("xorl\t%eax,%eax"); }
```

`cgldgb` — Load global (public/static) byte.

Global locations are referenced by symbolic names.

Note: byte load instructions do not need to clear the extra bytes (all but the least significant byte) of the destination register. The high-level part of the generator will take care of this.

```
void cgldgb(char *s)    { sgen("%s\t%s,%%al", "movb", s); }
```

`cgldgw` — Load global (public/static) word.

```
void cgldgw(char *s)    { sgen("%s\t%s,%%eax", "movl", s); }
```

`cgldlb` — Load local byte.

Local locations are referenced by stack frame offsets ($F + n$).

```
void cgldlb(int n)      { ngen("%s\t%d(%%ebp),%%al", "movb",
                                n); }
```

`cgldlw` — Load local word.

```
void cgldlw(int n)      { ngen("%s\t%d(%%ebp),%%eax", "movl",
                                n); }
```

`cgldsb` — Load local static byte.

Local static locations are referenced by label IDs.

```
void cgldsb(int n)      { lgen("%s\t%c%d,%%al", "movb", n); }
```

`cgldsw` — Load local static word.

```
void cgldsw(int n)      { lgen("%s\t%c%d,%%eax", "movl", n);}
```

`cgldla` — Load local address.

Load the absolute address of a local location.

```
void cgldla(int n)      { ngen("%s\t%d(%%ebp),%%eax", "leal",
                                n); }
```

`cgldsa` — Load local static address.

Load the address of a local static label.

```
void cgldsa(int n)      { lgen("%s\t$%c%d,%%eax", "movl",
                                n); }
```

`cgldga` — Load global address.

Load the address denoted by a public/static symbol.

```
void cgldga(char *s)    { sgen("%s\t$%s,%%eax", "movl", s); }
```

`cgindb` — Load indirect byte.

This instruction *does* clear out all but the least significant byte of the destination register.

```
void cgindb(void)       { gen("movl\t%eax,%edx");
                          cgclear();
                          gen("movb\t(%edx),%al"); }
```

`cgindw` — Load indirect word.

```
void cgindw(void)       { gen("movl\t(%eax),%eax"); }
```

`cgargc` — Load argument count.

```
void cgargc(void)       { gen("movl\t8(%ebp),%eax"); }
```

`cgldlab` — Load label address.

In the 386 back-end, this is basically the same as `cgldsa` (load local static address); `cgldlab` is used to load the addresses of string literals.

```
void cgldlab(int id)    { lgen("%s\t$%c%d,%%eax", "movl",
                               id); }
```

12.4 Miscellanea

These do not fit anywhere else.

`cgpush` — Push A onto the stack.

```
void cgpush(void)       { gen("pushl\t%eax"); }
```

`cgpushlit` — Push the literal (immediate value) n onto the stack.

```
void cgpushlit(int n)   { ngen("%s\t$%d", "pushl", n); }
```

`cgpop2` — Pop X off the stack.

```
void cgpop2(void)       { gen("popl\t%ecx"); }
```

`cgswap` — Exchange the contents of X and A.

```
void cgswap(void)       { gen("xchgl\t%eax,%ecx"); }
```

12.5 Binary Operations

These instructions implement the binary operators. They all operate on X and A and leave the result in A: $A = X\ op\ A$, where op is the operation to perform.

`cgand` — Bitwise "and".

```
void cgand(void)        { gen("andl\t%ecx,%eax"); }
```

`cgxor` — Bitwise "exclusive or".

```
void cgxor(void)        { gen("xorl\t%ecx,%eax"); }
```

`cgior` — Bitwise (inclusive) "or".

```
void cgior(void)        { gen("orl\t%ecx,%eax"); }
```

`cgadd` — Addition.

```
void cgadd(void)        { gen("addl\t%ecx,%eax"); }
```

`cgmul` — Multiplication.

```
void cgmul(void)        { gen("imull\t%ecx,%eax"); }
```

`cgsub` — Subtraction.

```
void cgsub(void)        { gen("subl\t%ecx,%eax"); }
```

`cgdiv` — Signed integer division.

```
void cgdiv(void)        { gen("cdq");
                          gen("idivl\t%ecx"); }
```

`cgmod` — Remainder of signed integer division.

```
void cgmod(void)        { cgdiv();
                          gen("movl\t%edx,%eax"); }
```

`cgshl` — Arithmetic left-shift.

```
void cgshl(void)        { gen("shll\t%cl,%eax"); }
```

`cgshr` — Arithmetic right-shift.

```
void cgshr(void)        { gen("sarl\t%cl,%eax"); }
```

The `cgcmp()` function generates code that generalizes the *complement* of the instruction passed to it, e.g. when passed a "jump on less" instruction (`jl`), it will emit code that implements a generalized "greater/equal" (or "*not*-less-than") operator:

```
        xorl    %edx,%edx
        popl    %ecx
        cmpl    %eax,%ecx
        jl      L1
        inc     %edx
L1:     movl    %edx,%eax
```

The resulting code will leave 1 in the primary register exactly if the TOS (operand 1) is *not less than* A (operand 2). Otherwise, it will leave 0 in A. Therefore, it implements the C operator `>=` (greater/equal, not less).

The `cgcmp()` function is used below to implement all the comparison operators of C. It is used internally only, so it may be tweaked as necessary or even omitted, e.g. on machines that do not use status (flags) registers and leave comparison results in A anyway.

```
void cgcmp(char *inst)  { int lab;
                          lab = label();
                          gen("xorl\t%edx,%edx");
                          cgpop2();
                          gen("cmpl\t%eax,%ecx");
                          lgen("%s\t%c%d", inst, lab);
                          gen("incl\t%edx");
                          genlab(lab);
                          gen("movl\t%edx,%eax"); }
```

`cgeq` — Equal to.

```
void cgeq()             { cgcmp("jne"); }
```

`cgne` — Not equal to.

```
void cgne()             { cgcmp("je"); }
```

`cglt` — Less than.

```
void cglt()             { cgcmp("jge"); }
```

`cggt` — Greater than.

```
void cggt()             { cgcmp("jle"); }
```

`cgle` — Less than or equal to.

```
void cgle()             { cgcmp("jg"); }
```

`cgge` — Greater than or equal to.

```
void cgge()             { cgcmp("jl"); }
```

12.6 Unary Operations

These instructions implement the unary operators. Most of them operate on A and one (`cgscale2`) operates on X.

`cgneg` — Negate.

```
void cgneg(void)        { gen("negl\t%eax"); }
```

`cgnot` — Bitwise "not" (invert).

```
void cgnot(void)        { gen("notl\t%eax"); }
```

`cglognot` — Logical "not".

Map zero to one and non-zero to zero. Have fun figuring out how it works!

```
void cglognot(void)     { gen("negl\t%eax");
                          gen("sbbl\t%eax,%eax");
                          gen("incl\t%eax"); }
```

`cgscale` — Scale (multiply by machine word size).

```
void cgscale(void)      { gen("shll\t$2,%eax"); }
```

`cgscale2` — Scale X.

```
void cgscale2(void)     { gen("shll\t$2,%ecx"); }
```

`cgunscale` — Unscale (divide by machine word size).

```
void cgunscale(void)    { gen("shrl\t$2,%eax"); }
```

`cgbool` — Normalize truth value.

Map zero to zero and non-zero to one.

```
void cgbool(void)       { gen("negl\t%eax");
                          gen("sbbl\t%eax,%eax");
                          gen("negl\t%eax"); }
```

12.7 Increment Operations

These are the various *increment* fragments emitted by `geninc()`.

`cgldinc` — Load increment pointer to X.

Create a copy of the address of an indirect operand before indirection. Used for post-incrementing indirect operands.

Note: some operations use the `%edx` register as X, some use the `%ecx` register as X. This does not matter, because the X register is only ever used temporarily in a few subsequent instructions. Both `%ecx` and `%edx` are unavailable for general purposes anyway, because they are bound to the shift, divide, and multiply operations.

```
void cgldinc(void)      { gen("movl\t%eax,%edx"); }
```

`cginc1pi` — Pre-increment pointer indirect.

```
void cginc1pi(void)     { ngen("%s\t$4,(%%eax)", "addl", 0);}
```

`cgdec1pi` — Pre-decrement pointer indirect.

```
void cgdec1pi(void)     { ngen("%s\t$4,(%%eax)", "subl", 0);}
```

`cginc2pi` — Post-increment pointer indirect.

```
void cginc2pi(void)     { ngen("%s\t$4,(%%edx)", "addl", 0);}
```

`cgdec2pi` — Post-decrement pointer indirect.

```
void cgdec2pi(void)     { ngen("%s\t$4,(%%edx)", "subl", 0);}
```

`cgincpl` — Increment pointer local.

```
void cgincpl(int a)     { ngen("%s\t$4,%d(%%ebp)", "addl",
                                a); }
```

`cgdecpl` — Decrement pointer local.

```
void cgdecpl(int a)     { ngen("%s\t$4,%d(%%ebp)", "subl",
                                a); }
```

`cgincps` — Increment pointer local static.

```
void cgincps(int a)     { lgen("%s\t$4,%c%d", "addl", a); }
```

`cgdecps` — Decrement pointer local static.

```
void cgdecps(int a)     { lgen("%s\t$4,%c%d", "subl", a); }
```

`cgincpg` — Increment pointer global.

```
void cgincpg(char *s)   { sgen("%s\t$4,%s", "addl", s); }
```

`cgdecpg` — Decrement pointer global.

```
void cgdecpg(char *s)   { sgen("%s\t$4,%s", "subl", s); }
```

`cginc1iw` — Pre-increment indirect word.

```
void cginc1iw(void)     { ngen("%s\t(%%eax)", "incl", 0); }
```

`cgdec1iw` — Pre-decrement indirect word.

```
void cgdec1iw(void)     { ngen("%s\t(%%eax)", "decl", 0); }
```

`cginc2iw` — Post-increment indirect word.

```
void cginc2iw(void)     { ngen("%s\t(%%edx)", "incl", 0); }
```

`cgdec2iw` — Post-decrement indirect word.

```
void cgdec2iw(void)     { ngen("%s\t(%%edx)", "decl", 0); }
```

`cginclw` — Increment local word.

```
void cginclw(int a)     { ngen("%s\t%d(%%ebp)", "incl", a); }
```

`cgdeclw` — Decrement local word.

```
void cgdeclw(int a)     { ngen("%s\t%d(%%ebp)", "decl", a); }
```

`cgincsw` — Increment local static word.

```
void cgincsw(int a)     { lgen("%s\t%c%d", "incl", a); }
```

`cgdecsw` — Decrement local static word.

```
void cgdecsw(int a)     { lgen("%s\t%c%d", "decl", a); }
```

`cgincgw` — Increment global word.

```
void cgincgw(char *s)   { sgen("%s\t%s", "incl", s); }
```

`cgdecgw` — Decrement global word.

```
void cgdecgw(char *s)   { sgen("%s\t%s", "decl", s); }
```

`cginc1ib` — Pre-increment indirect byte.

```
void cginc1ib(void)     { ngen("%s\t(%%eax)", "incb", 0); }
```

`cgdec1ib` — Pre-decrement indirect byte.

```
void cgdec1ib(void)     { ngen("%s\t(%%eax)", "decb", 0); }
```

`cginc2ib` — Post-increment indirect byte.

```
void cginc2ib(void)     { ngen("%s\t(%%edx)", "incb", 0); }
```

`cgdec2ib` — Post-decrement indirect byte.

```
void cgdec2ib(void)     { ngen("%s\t(%%edx)", "decb", 0); }
```

`cginclb` — Increment local byte.

```
void cginclb(int a)     { ngen("%s\t%d(%%ebp)", "incb", a); }
```

`cgdeclb` — Decrement local byte.

```
void cgdeclb(int a)     { ngen("%s\t%d(%%ebp)", "decb", a); }
```

`cgincsb` — Increment local static byte.

```
void cgincsb(int a)     { lgen("%s\t%c%d", "incb", a); }
```

`cgdecsb` — Decrement local static byte.

```
void cgdecsb(int a)     { lgen("%s\t%c%d", "decb", a); }
```

`cgincgb` — Increment global byte.

```
void cgincgb(char *s)   { sgen("%s\t%s", "incb", s); }
```

`cgdecgb` — Decrement global byte.

```
void cgdecgb(char *s)   { sgen("%s\t%s", "decb", s); }
```

12.8 Jumps and Branches

This sections contains various jump and conditional branch operations.

The `cgbr` function may be used internally when the target CPU does not support *unlimited-range conditional branches*. It extends the range of a conditional short branch by performing a short conditional branch (with the complementary condition) around an unconditional long jump, e.g. a long "branch on zero" to *dest* on the 386 would be implemented as

```
        orl     %eax,%eax
        jnz     L1
        jmp     dest
L1:
```

See also: cgcmp(), pg. 197.

```
void cgbr(char *how, int n)
                        { int lab;
                          lab = label();
                          gen("orl\t%eax,%eax");
                          lgen("%s\t%c%d", how, lab);
                          lgen("%s\t%c%d", "jmp", n);
                          genlab(lab); }
```

`cgbrtrue` — Branch on A not zero.

```
void cgbrtrue(int n)    { cgbr("jz", n); }
```

`cgbrfalse` — Branch on A zero.

```
void cgbrfalse(int n)   { cgbr("jnz", n); }
```

`cgjump` — Jump to label n.

```
void cgjump(int n)      { lgen("%s\t%c%d", "jmp", n); }
```

`cgldswtch` — Load switch table address to X.

The n parameter is the label ID of the switch table.

```
void cgldswtch(int n)   { lgen("%s\t$%c%d,%%edx", "movl",
                                n); }
```

`cgcalswtch` — Evaluate switch table.

This is typically a call to an internal routine (defined in the C startup module; pg. 223). This implementation expects the `switch` expression in A and the switch table address in X (`%edx`).

```
void cgcalswtch(void)   { gen("jmp\tswitch"); }
```

`cgcase` — Case template for switch table.

The v argument is the `case` value, l is the associated label.

```
void cgcase(int v, int l)
                        { lgen2(".long\t%d,%c%d", v, l); }
```

12.9 Store Operations

These instructions are used for storing values in locations.

`cgpopptr` — Pop indirect address to X.

Used for storing to anonymous locations.

```
void cgpopptr(void)     { gen("popl\t%edx"); }
```

`cgstorib` — Store indirect byte.

```
void cgstorib(void)     { ngen("%s\t%%al,(%%edx)", "movb",
                                0); }
```

`cgstoriw` — Store indirect word.

```
void cgstoriw(void)     { ngen("%s\t%%eax,(%%edx)", "movl",
                                0); }
```

`cgstorlb` — Store local byte.

```
void cgstorlb(int n)    { ngen("%s\t%%al,%d(%%ebp)", "movb",
                                n); }
```

`cgstorlw` — Store local word.

```
void cgstorlw(int n)    { ngen("%s\t%%eax,%d(%%ebp)", "movl",
                                n); }
```

`cgstorsb` — Store local static byte.

```
void cgstorsb(int n)    { lgen("%s\t%%al,%c%d", "movb", n); }
```

`cgstorsw` — Store local static word.

```
void cgstorsw(int n)    { lgen("%s\t%%eax,%c%d", "movl", n);}
```

`cgstorgb` — Store global byte.

```
void cgstorgb(char *s)  { sgen("%s\t%%al,%s", "movb", s); }
```

`cgstorgw` — Store global word.

```
void cgstorgw(char *s)  { sgen("%s\t%%eax,%s", "movl", s); }
```

12.10 Functions and Function Calls

These instructions set up call frames and local variables, perform function calls, allocate and deallocate *automatic storage*, etc.

`cginitlw` — Initialize local word.

There is no corresponding byte instruction; `char`s on the stack allocate one word.

```
void cginitlw(int v, int a)
                        { ngen2("%s\t$%d,%d(%%ebp)", "movl",
                                v, a); }
```

`cgcall` — Call function *s*.

```
void cgcall(char *s)    { sgen("%s\t%s", "call", s); }
```

`cgcalr` — Call register.

Invoke the function pointed to by *A*.

```
void cgcalr(void)       { gen("call\t*%eax"); }
```

`cgstack` — Modify stack pointer (S).

Add n to the stack pointer.

```
void cgstack(int n)      { ngen("%s\t$%d,%%esp", "addl", n); }
```

`cgentry` — Function entry point.

Set up a new context (F).

```
void cgentry(void)       { gen("pushl\t%ebp");
                           gen("movl\t%esp,%ebp"); }
```

`cgexit` — Function exit point.

Restore the caller's context and return.

```
void cgexit(void)        { gen("popl\t%ebp");
                           gen("ret"); }
```

12.11 Data Definitions

These are assembler directives for setting up public, static, and local static data objects.

`cgdefb` — Define byte.

```
void cgdefb(int v)       { ngen("%s\t%d", ".byte", v); }
```

`cgdefw` — Define word.

```
void cgdefw(int v)       { ngen("%s\t%d", ".long", v); }
```

`cgdefp` — Define pointer.

This is actually the same as `cgdefw`.

```
void cgdefp(int v)       { ngen("%s\t%d", ".long", v); }
```

`cgdefl` — Define label.

Define a word-size cell holding the address of a label; v is a label ID.

```
void cgdefl(int v)      { lgen("%s\t%c%d", ".long", v); }
```

`cgdefc` — Define character.

Like `cgdefb`, but for alpha-numeric characters only.

```
void cgdefc(int c)      { ngen("%s\t'%c'", ".byte", c); }
```

`cgbss` — Define BSS object.

Define block started by symbol (*BSS*); s is the name of the block, z is its length.

```
void cgbss(char *s, int z)
                        { ngen(".lcomm\t%s,%d", s, z); }
```

Chapter 13

The Compiler Controller

All that is missing at this point is the part that brings it all together. This part is called the *compiler controller*. It accepts arguments from the command line, compiles C to assembly language, and invokes the system assembler and linker to generate *executable files*.

The SubC compiler controller is contained in the file `main.c`. The module defines `_extern` before including `data.h`, which disables the `extern` keywords in that file (pg. 30ff), resulting in public instead of external declarations. So `main.c` defines all the global variables used by the compiler.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  Main program
 */

#include "defs.h"
#define _extern
 #include "data.h"
#undef _extern
#include "decl.h"
```

The `init()` function initializes the internal structures of the compiler. It is called before compiling each individual input file. The function adds a dummy entry to slot 0 of the *symbol table* because in the rest of the compiler, slot 0 will indicate a non-existent symbol. `init()` also adds the `__SUBC__` macro, which identifies the compiler.

```
static void init(void) {
    Line = 1;
```

```
    Putback = '\n';
    Rejected = -1;
    Errors = 0;
    Mp = 0;
    Expandmac = 1;
    Syntoken = 0;
    Isp = 0;
    Inclev = 0;
    Globs = 0;
    Locs = NSYMBOLS;
    Nbot = 0;
    Ntop = POOLSIZE;
    Bsp = 0;
    Csp = 0;
    addglob("", 0, 0, 0, 0, 0, NULL, 0);
    addglob("__SUBC__", 0, TMACRO, 0, 0, 0, globname(""), 0);
    Infile = stdin;
    File = "(stdin)";
    Basefile = NULL;
    Outfile = stdout;
}
```

The `cmderror()` function prints an error message that is related to command line options or internal limits of the compiler controller. It then aborts compilation.

```
static void cmderror(char *s, char *a) {
    fprintf(stderr, "scc: ");
    fprintf(stderr, s, a);
    fputc('\n', stderr);
    exit(EXIT_FAILURE);
}
```

The `filetype()` function returns the *file type* of the argument *file*. The file type is the last character of the file name, e.g. `'c'` for `"main.c"` or `'o'` for `"stmt.o"`. We assume that all suffixes are two-character ones with a dot in the first position (such as "`.c`"). When *file* has no such suffix, `filetype` returns 0.

```
static int filetype(char *file) {
    int k;
```

```
    k = strlen(file);
    if ('.' == file[k-2]) return file[k-1];
    return 0;
}
```

The `defarg()` function defines a macro before compiling a file. When the argument *s* contains an "`=`" sign (e.g. `"name=value"`) then it defines a *macro* `name` with with value `value`; otherwise is just defines a macro named *s* with an empty value.

```
static void defarg(char *s) {
    char    *p;

    if (NULL == s) return;
    if ((p = strchr(s, '=')) != NULL)
        *p++ = 0;
    else
        p = "";
    addglob(s, 0, TMACRO, 0, 0, 0, globname(p), 0);
    if (*p) *--p = '=';
}
```

The `stats()` function prints some *memory usage* data.

```
static void stats(void) {
    printf( "Memory usage: "
        "Symbols: %5d/%5d, "
        "Name pool: %5d/%5d\n",
        Globs, NSYMBOLS,
        Nbot, POOLSIZE);
}
```

The `compile()` function *compiles* the file *file* with the optional macro definition *def* in place. When *file* is `NULL`, it compiles `stdin` to `stdout`. When the compiler is in *test mode*, it suppresses output by setting `Outfile` to `NULL`.

When a non-`NULL` *file* was specified, then the output of `compile()` will be written to a file with the same base name and a "`.s`" suffix. However, `compile()` will *never* overwrite an existing "`.s`" file, because it might be a user-supplied assembly language program. So the user must remove left-over compiler output files manually—better safe than sorry!

On the other hand, the compiler considers "`.o`" (relocatable object) files to be worthless and it will overwrite or delete them silently.

```
static void compile(char *file, char *def) {
    char    *ofile;

    init();
    defarg(def);
    if (file) {
        ofile = newfilename(file, 's');
        if ((Infile = fopen(file, "r")) == NULL)
            cmderror("no such file: %s", file);
        Basefile = File = file;
        if (!O_testonly) {
            if ((Outfile = fopen(ofile, "r")) != NULL)
                cmderror("will not overwrite: %s", ofile);
            if ((Outfile = fopen(ofile, "w")) == NULL)
                cmderror("cannot create file: %s", ofile);
        }
    }
    if (O_testonly) Outfile = NULL;
    if (O_verbose) {
        if (O_testonly)
            printf("testing %s\n", file);
        else
            printf("compiling %s\n", file);
    }
    genprelude();
    Token = scan();
    while (XEOF != Token)
        top();
    genpostlude();
    if (file) {
        fclose(Infile);
        if (Outfile) fclose(Outfile);
    }
    if (O_debug & D_GSYM) dumpsyms("GLOBALS", "", 1, Globs);
    if (O_debug & D_STAT) stats();
}
```

The `collect()` function collects a file name for later use (i.e.: linking).

```
static void collect(char *file) {
```

```
    if (O_componly || O_asmonly) return;
    if (Nf >= MAXFILES)
        cmderror("too many input files", NULL);
    Files[Nf++] = file;
}
```

The `assemble()` function submits the file *file* to the *system assembler*. When the *delete* flag is set, it deletes its input file when finished.

```
static void assemble(char *file, int delete) {
    char    *ofile;
    char    cmd[TEXTLEN+1];

    file = newfilename(file, 's');
    collect(ofile = newfilename(file, 'o'));
    if (strlen(file) + strlen(ofile) + strlen(ASCMD)
        >= TEXTLEN
    )
        cmderror("assembler command too long", NULL);
    sprintf(cmd, ASCMD, ofile, file);
    if (O_verbose > 1) printf("%s\n", cmd);
    if (system(cmd))
        cmderror("assembler invocation failed", NULL);
    if (delete) remove(file);
}
```

The `concat()` function concatenates s and the buffer *buf* by appending s to *buf*. The parameter k specifies the current length of *buf*. The `concat()` function makes sure that the buffer length will not exceed `TEXTLEN` characters. It returns the new length of the string in the buffer.

```
static int concat(int k, char *buf, char *s) {
    int n;

    n = strlen(s);
    if (k + n + 2 >= TEXTLEN)
        cmderror("command too long", buf);
    strcat(buf, " ");
    strcat(buf, s);
    return k + n + 1;
}
```

The `link()` function *links* all files collected during the compilation phase. The command submitted to the *system linker* will be the following:

`ld -o` *a.out* *SCCDIR*`/lib/crt0.o` *file1.o* ... *fileN.o* *SCCLIBC* \
SYSLIBC

where `file1.o` through `fileN.o` are the names of the relocatable *object files* generated by the assembler during the same invocation of the compiler (see `collect()`, `main()`).

The default output file name, `a.out`, can be changed on the command line.

```
static void link(void) {
    int     i, k;
    char    cmd[TEXTLEN+1], *msg;
    char    cmd2[TEXTLEN+1];

    if (strlen(O_outfile) + strlen(LDCMD) + strlen(SCCDIR)*2
        >= TEXTLEN
    )
        cmderror(msg, NULL);
    sprintf(cmd, LDCMD, O_outfile, SCCDIR);
    k = strlen(cmd);
    for (i=0; i<Nf; i++)
        k = concat(k, cmd, Files[i]);
    concat(k, cmd, SCCLIBC);
    concat(k, cmd, SYSLIBC);
    sprintf(cmd2, cmd, SCCDIR);
    if (O_verbose > 1) printf("%s\n", cmd2);
    if (system(cmd2))
        cmderror("linker invocation failed", NULL);
}
```

The `usage()` and `longusage()` functions print a terse synopsis of the compiler options and a more extensive help text, respectively.

```
static void usage(void) {
    printf("Usage: scc [-h] [-d opt] [-cvST] [-o file]"
           " [-D macro[=text]] file [...]\n");
}
```

```
static void longusage(void) {
    printf("\n");
    usage();
    printf( "\n"
        "-c       compile only, do not link\n"
        "-d opt   activate debug option OPT\n"
        "-o file  write linker output to 'file'\n"
        "-t       test only, generate no code\n"
        "-v       verbose, more v's = more verbose\n"
        "-D m=v   define macro M with optional value V\n"
        "-S       compile to assembly language\n"
        "\n" );
}
```

The `nextarg()` function extracts the next argument from the command line, where *argc* and *argv* are copies of the parameters of `main()`, *pi* points to the index of the current argument and *pj* points to the index the current character of the current argument. Consider the options

```
-o file
```

where *pi* is the index of `"-o"` and *pj* the index of the `'o'` in that string. In this case `nextarg()` will return `"file"` and set *pi* to the index of `"file"` and *pj* to the index of the trailing `'e'` of `"file"`.

```
static char *nextarg(int argc, char *argv[], int *pi,
                         int *pj)
{
    char    *s;

    if (argv[*pi][*pj+1] || *pi >= argc-1) {
        usage();
        exit(EXIT_FAILURE);
    }
    s = argv[++*pi];
    *pj = strlen(s)-1;
    return s;
}
```

The `dbgopt()` function extracts a *debug option* from the command line and sets the corresponding option flag. See the `nextarg()` function (above) for a description of its parameters.

```
static int dbgopt(int argc, char *argv[], int *pi, int *pj) {
    char    *s;

    s = nextarg(argc, argv, pi, pj);
    if (!strcmp(s, "lsym")) return D_LSYM;
    if (!strcmp(s, "gsym")) return D_GSYM;
    if (!strcmp(s, "stat")) return D_STAT;
    printf( "\n"
            "scc: valid -d options are: \n\n"
            "lsym - dump local symbol tables\n"
            "gsym - dump global symbol table\n"
            "stat - print usage statistics\n"
            "\n");
    exit(EXIT_FAILURE);
}
```

The `main()` function evaluates the command line options passed to the compiler controller, compiles the given files, assembles them (unless `-S` or `-t` is specified), and links them against the C library (unless `-c`, `-S`, or `-t` is specified). When no options are given, it compiles C source files ("`.c`"), assembles assembly language source files ("`.s`"), and links relocatable object files ("`.o`"). For instance, the command

```
scc -o goo foo.c bar.s baz.o
```

would

- compile `foo.c` to `foo.s`;
- assemble `foo.s` to `foo.o`;
- assemble `bar.s` to `bar.o`;
- link `foo.o`, `bar.o`, and `baz.o`;
- generate an executable named `goo`;
- delete the file `foo.s` (because it was generated by the compiler);
- keep the file `bar.s` (because it was supplied by the user).

The `main()` function will return `EXIT_SUCCESS` when compilation succeeded and `EXIT_FAILURE` when either compilation failed or an invocation of the assembler or linker returned an exit code indicating failure.

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

```
    char    *def;

    def = NULL;
    O_verbose = 0;
    O_componly = 0;
    O_asmonly = 0;
    O_testonly = 0;
    O_outfile = "a.out";
    for (i=1; i<argc; i++) {
        if (*argv[i] != '-') break;
        if (!strcmp(argv[i], "-")) {
            compile(NULL, def);
            exit(Errors? EXIT_FAILURE: EXIT_SUCCESS);
        }
        for (j=1; argv[i][j]; j++) {
            switch (argv[i][j]) {
            case 'c':
                O_componly = 1;
                break;
            case 'd':
                O_debug |= dbgopt(argc, argv, &i, &j);
                O_testonly = 1;
                break;
            case 'h':
                longusage();
                exit(EXIT_SUCCESS);
            case 'o':
                O_outfile = nextarg(argc, argv, &i, &j);
                break;
            case 't':
                O_testonly = 1;
                break;
            case 'v':
                O_verbose++;
                break;
            case 'D':
                if (def) cmderror("too many -D's", NULL);
                def = nextarg(argc, argv, &i, &j);
                break;
            case 'S':
                O_componly = O_asmonly = 1;
                break;
            default:
```

```
                usage();
                exit(EXIT_FAILURE);
            }
        }
    }
    if (i >= argc) {
        usage();
        exit(EXIT_FAILURE);
    }
    Nf = 0;
    while (i < argc) {
        if (filetype(argv[i]) == 'c') {
            compile(argv[i], def);
            if (Errors && !O_testonly)
                cmderror("compilation stopped", NULL);
            if (!O_asmonly && !O_testonly)
                assemble(argv[i], 1);
            i++;
        }
        else if (filetype(argv[i]) == 's') {
            if (!O_testonly) assemble(argv[i++], 0);
        }
        else {
            collect(argv[i++]);
        }
    }
    if (!O_componly && !O_testonly) link();
    return EXIT_SUCCESS;
}
```

Part III

Runtime Environment

Chapter 14

The Runtime Startup Module

The C *runtime startup module* (traditionally called `crt0`) is contained in the file `crt0.s`. It is the only file of the runtime environment that has to be written in assembly language.

```
#
#       NMH's Simple C Compiler, 2011,2012
#       C runtime module
#
```

The `environ` variable (of the type `char**`) will be filled with the address of the environment array. See the `getenv()` function (pg. 265) for further details.

Note that all public names defined in this file must have the single-character prefix `PREFIX` (`defs.h`, pg. 25ff) attached, because all public names generated by the compiler also have this prefix. For example, the compiler will generate the public name `Cenviron` (with the default prefix) for the variable `environ`.

When `PREFIX` is changed, *all public names in this file will have to change as well*!

```
        .data
        .globl  Cenviron
Cenviron:
        .long   0
```

The `_start` symbol indicates the default entry point for *FreeBSD*'s ELF linker. So here the fun begins. When control is transfered to `_start` by the operating system, the stack pointer points to the structure depicted in figure 14.1.

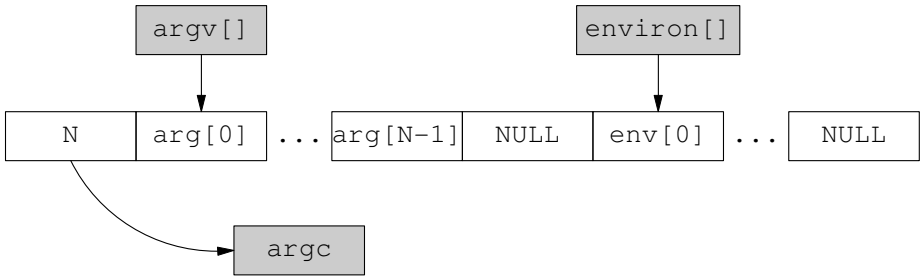


Figure 14.1: The layout of the startup parameter array

One of the first things that the below code does is to decompose this structure and store pointers to its individual parts in the `environ` variable and in the `argc` and `argv` arguments of `main()`.

After pushing `argc`, `argv[]`, and `__argc` onto the stack, the startup code calls `main()` (using the SubC calling conventions) and passes its return value on to `exit()`. In case `exit()` should return, the code performs a constant divide by zero (hoping that this will crash the program) and then keeps calling `exit()` and dividing by zero indefinitely.

The very first thing the code does is a call to `_init()`. This function will initialize the C runtime library. We will get to that in the next chapter.

```
        .text
        .globl  _start
_start: call    C_init
        leal    4(%esp),%esi    # argv
        movl    0(%esp),%ecx    # argc
        movl    %ecx,%eax       # environ = &argv[argc+1]
        incl    %eax
        shll    $2,%eax
        addl    %esi,%eax
        movl    %eax,Cenviron
        pushl   %ecx
        pushl   %esi
        pushl   $2              # __argc
        call    Cmain
        addl    $12,%esp
        pushl   %eax
```

```
        pushl   $1
x:      call    Cexit
        xorl    %ebx,%ebx
        divl    %ebx
        jmp     x
```

The internal[1] `switch` routine evaluates a *switch table* (pg. 183).

It expects the address of a switch table in `%edx` and the value of the corresponding `switch` expression in `%eax`. It checks `%eax` against each "value" entry in the table and jumps to the corresponding label if it finds a match. When no match is found, it jumps to the label pointed to by the last element of the table, which is the default case (or the end of the `switch` statement when no `default` clause was present).

The routine uses the `%esi` register to loop over the table, but it saves the register value initially and restores it before returning, so a register allocator (if SubC had one) could safely assume that the value of `%esi` is unchanged after processing a `switch` statement.

```
        .globl  switch
switch: pushl   %esi
        movl    %edx,%esi
        movl    %eax,%ebx
        lodsl
        movl    %eax,%ecx
next:   lodsl
        movl    %eax,%edx
        lodsl
        cmpl    %edx,%ebx
        jnz     no
        popl    %esi
        jmp     *%eax
no:     loop    next
        lodsl
        popl    %esi
        jmp     *%eax
```

```
int setjmp(jmp_buf env);
```

This routine implements `setjmp()`. Its argument is a two-element `int` array. It stores the stack pointer in the first slot of the array and the return

[1]The routine cannot be called from a SubC program by calling `switch`, because the compiler would turn that name into `Cswitch`

address (saved by the call to this function) in the second slot of the array. It returns 0.

```
        .globl  Csetjmp
Csetjmp:
        movl    8(%esp),%edx
        movl    %esp,(%edx)
        movl    (%esp),%eax
        movl    %eax,4(%edx)
        xorl    %eax,%eax
        ret
```

```
void longjmp(jmp_buf env, int v);
```

This routine implements `longjmp()`. Like `setjmp()` it expects a two-element `int` array as its first argument. It sets the stack pointer to the first slot of the array and then jumps to the location stored in the second slot, transferring control to the point where the corresponding `setjmp()` returned. It makes the original `setjmp()` return v.

```
        .globl  Clongjmp
Clongjmp:
        movl    8(%esp),%eax
        movl    12(%esp),%edx
        movl    (%edx),%esp
        movl    4(%edx),%edx
        jmp     *%edx
```

The remainder of this file is basically a thin translation layer that translates calls to Unix system services from SubC to traditional C *calling conventions*, i.e. the following functions reverse the order of their arguments, remove the additional `__argc` parameter, call the corresponding function in the C system library, clean up the arguments, and return the return value of the system call to the calling SubC function. A sample translation for the `read()` system call is illustrated in figure 14.2.

A function translating between two interfaces is commonly referred to as a *wrapper*. The wrappers defined in the SubC startup module provide access to all the system calls that are needed to implement the functions of the SubC library. The name of each system call is made visible to SubC programs with an underscore attached, e.g. the `read()` system call can be invoked from within SubC programs by calling `_read()`, etc.

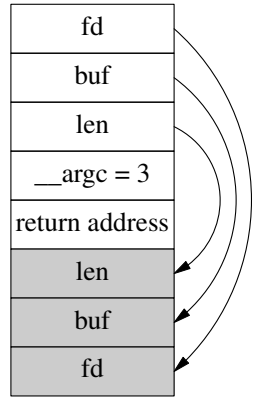


Figure 14.2: Translation of system call parameters

The system calls required by the SubC library are the following: `_exit`, `close`, `creat`, `execve`, `fork`, `lseek`, `open`, `read`, `rename`, `sbrk`, `signal`, `time`, `unlink`, `wait`, `write`.[2]

The startup module also implements the `raise()` function which is, strictly speaking, not a system call.

14.1 The System Calls

```
void _exit(int rc);
```

Terminate program execution, return *rc*`&127` to the calling process.

The `_exit()` wrapper demonstrates how the external name prefix (`PREFIX`) allows SubC functions to call system functions with the same name: SubC's `_exit()` function, which is externally named `C_exit()`, calls the system's `_exit()` function to terminate program execution. Without the prefix `_exit()` would call itself.

```
        .globl  C_exit
C_exit: pushl   8(%esp)
        call    _exit
        addl    $4,%esp
        ret
```

```
void *_sbrk(int size);
```

[2]The author is aware of the fact that `creat`, `sbrk`, and `signal` are obsolete, but he has decided to use them anyway, because they are simple and he knows them. This is kind of an underwear thing: you like what you are used to.

Increase the size of the process data segment by *size* bytes. The size may be negative, which shrinks the segment. Return a pointer to the current "break", i.e. the beginning of the memory region allocated by `_sbrk()`.

```
        .globl  C_sbrk
C_sbrk: pushl   8(%esp)
        call    sbrk
        addl    $4,%esp
        ret
```

```
int _write(int fd, void *buf, int len);
```

Write *len* bytes from the buffer *buf* to the file descriptor *fd*, return the number of bytes written.

```
        .globl  C_write
C_write:
        pushl   8(%esp)
        pushl   16(%esp)
        pushl   24(%esp)
        call    write
        addl    $12,%esp
        ret
```

```
int _read(int fd, void *buf, int len);
```

Read *len* bytes from the file descriptor *fd* to the buffer *buf*, return the number of bytes read.

```
        .globl  C_read
C_read: pushl   8(%esp)
        pushl   16(%esp)
        pushl   24(%esp)
        call    read
        addl    $12,%esp
        ret
```

```
int _lseek(int fd, int pos, int how);
```

Move the file pointer of the descriptor *fd* to the position *pos*, where *how* is the origin of the move; *how* may have the following values:

how	stdio symbol	origin
0	SEEK_SET	beginning of file
1	SEEK_CUR	current position
2	SEEK_END	end of file

When your system's implementation does not use the above *how* values, these must either be mapped in the `_lseek` function or changed in SubC's `stdio.h` file.

Note: `_lseek()` accepts an `int` position, but uses a 64-bit integer type internally on FreeBSD, so it sign-extends the value of the *pos* argument. It also returns an `int` position. Hence it can only operate safely on files whose size is not larger than `INT_MAX`.

```
        .globl  C_lseek
C_lseek:
        pushl   8(%esp)
        movl    16(%esp),%eax
        cdq
        pushl   %edx            # off_t, high word
        pushl   %eax            # off_t, low word
        pushl   28(%esp)
        call    lseek
        addl    $16,%esp
        ret
```

```
int _creat(char *path, int mode);
```

Create a new file named *path* with access mode *mode*, return a new file descriptor in read/write mode.

```
        .globl  C_creat
C_creat:
        pushl   8(%esp)
        pushl   16(%esp)
        call    creat
        addl    $8,%esp
        ret
```

```
int _open(char *path, int flags);
```

Open the existing file *path* using the given *flags* and return a new file descriptor. The following open *flags* are supported:

flags	meaning
0	open in read-only mode
1	open in write-only mode
2	open in read/write mode

When your system's `open()` call uses different flags, the above must be mapped to the values expected by your implementation.

```
        .globl  C_open
C_open: pushl   8(%esp)
        pushl   16(%esp)
        call    open
        addl    $8,%esp
        ret
```

```
int _close(int fd);
```

Close the file descriptor *fd*.

```
        .globl  C_close
C_close:
        pushl   8(%esp)
        call    close
        addl    $4,%esp
        ret
```

```
int _unlink(char *path);
```

Remove the directory entry *path*.

```
        .globl  C_unlink
C_unlink:
        pushl   8(%esp)
        call    unlink
        addl    $4,%esp
        ret
```

```
int _rename(char *old, char *new);
```

Rename the directory entry *old* to *new*.

```
        .globl  C_rename
C_rename:
```

```
        pushl   8(%esp)
        pushl   16(%esp)
        call    rename
        addl    $8,%esp
        ret
```

```
int _fork(void);
```

Create a copy of the calling process, return the process ID (PID) of the new copy (the child) to the old copy (the parent). Return 0 to the child.

```
        .globl  C_fork
C_fork: call    fork
        ret
```

```
int _wait(int *rc);
```

Wait for a forked child process to terminate. Write the return code of the terminated process to the integer pointed to by *rc*.

```
        .globl  C_wait
C_wait: pushl   8(%esp)
        call    wait
        addl    $4,%esp
        ret
```

```
int _execve(char *path, char *argv[], char *envp[]);
```

Execute a new process (terminating the caller). *Path* is the path of the program to execute, *argv[]* is the argument array passed to the process, and *env[]* is the environment array passed to the process.

```
        .globl  C_execve
C_execve:
        pushl   8(%esp)
        pushl   16(%esp)
        pushl   24(%esp)
        call    execve
        addl    $12,%esp
        ret
```

```
int _time(void);
```

Return the current system time.

```
        .globl  C_time
C_time: push    $0
        call    time
        addl    $4,%esp
        ret
```

```
int raise(int sig);
```

Send signal *sig* to the own process. See `signal()` below for possible *sig* values.

```
        .globl  Craise
Craise: call    getpid
        push    8(%esp)
        push    %eax
        call    kill
        addl    $8,%esp
        ret
```

```
int signal(int sig, int (*fn)());
```

Specify how to handle incoming signals. *Sig* is the signal to handle and *fn* is either a function to handle the signal or one of these constants:

fn	symbol	description
`(int(*)())0`	`SIG_DFL`	take default action
`(int(*)())1`	`SIG_IGN`	ignore the signal

The function returns `SIG_ERR` (`int(*)())2`) in case of an error. SubC specifies the following signals:

sig	symbol	description
2	`SIGINT`	keyboard interrupt
4	`SIGILL`	illegal instruction
6	`SIGABRT`	abort
8	`SIGFPE`	floating point exception
11	`SIGSEGV`	segmentation violation
15	`SIGTERM`	program termination

The values of the above symbolic constants may have to be changed in SubC's `signal.h` header when porting the compiler to a different operating system.

```
        .globl  Csignal
Csignal:
```

```
        pushl   8(%esp)
        pushl   16(%esp)
        call    signal
        addl    $8,%esp
        ret
```

14.2 The System Call Header

The system call interface of SubC is defined in the `syscall.h` header file (which is specific to SubC and not covered by the standard). The `signal()` and `raise()` functions are prototyped in `signal.h`. The "syscall" header is reproduced below.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  syscall.h
 */

int   _close(int fd);
int   _creat(char *path, int mode);
int   _execve(char *path, char *argv[], char *envp[]);
void  _exit(int rc);
int   _fork(void);
int   _lseek(int fd, int pos, int how);
int   _open(char *path, int flags);
int   _read(int fd, void *buf, int len);
int   _rename(char *old, char *new);
void  *_sbrk(int size);
int   _time(void);
int   _unlink(char *path);
int   _wait(int *rc);
int   _write(int fd, void *buf, int len);
```

Chapter 15

The Runtime Library

The SubC compiler—like virtually all C programs—makes heavy use of the C *standard library*. In this chapter we will have a look at those parts of the library that are required by the compiler itself, namely the standard I/O ("stdio") library, the string library, the character class ("ctype") library, and the utility ("stdlib") library.

The implementation presented in this chapter will be simple, close to the letter of the standard (TCPL2), but not very efficient, mostly because it attempts to *fail early* in order to be robust and reliable.

Note: This part of the book contains only those parts of the library that will actually be linked into the final SubC compiler. If you want to study functions that are not contained in this part, please refer to the SubC source code archive.[1]

15.1 Library Initialization

The file `init.c` contains the initialization function and the static data used by the C library. It defines the `errno` variable, the `FILE`s available to the library (see the section on the `stdio.h` header on pg. 235ff for details), and the standard streams `stdin`, `stdout`, and `stderr`.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  C runtime initialization
 */

#include <stdio.h>
```

[1] See `http://www.t3x.org`.

```
#include <errno.h>

int errno = EOK;

int   _file_fd  [FOPEN_MAX];
char  _file_iom [FOPEN_MAX];
char  _file_last[FOPEN_MAX];
char  _file_mode[FOPEN_MAX];
int   _file_ptr [FOPEN_MAX];
int   _file_end [FOPEN_MAX];
int   _file_size[FOPEN_MAX];
int   _file_ch  [FOPEN_MAX];
char *_file_buf [FOPEN_MAX];

FILE  *stdin, *stdout, *stderr;
```

The `_init()` function is called by the C runtime startup module `crt0`. It resets all `FILE`s to "closed" state and then initializes and assigns `FILE`s with the proper access modes to the *standard input*, *output*, and *error* file descriptors. It also puts the `stdout` and `stderr` streams into line buffering mode.

```
void _init(void) {
    int i;

    for (i = 0; i < FOPEN_MAX; i++)
        _file_iom[i] = _FCLOSED;
    stdin = fdopen(0, "r");
    stdout = fdopen(1, "w");
    stderr = fdopen(2, "w");
    _file_mode[1] = _IOLBF;
    _file_mode[2] = _IOLBF;
}
```

15.2 Standard I/O

The *standard I/O library* contains functions for opening, closing, reading, and writing file *streams*. All input/output functions of the C standard library are contained in this part. Its central data type is the `FILE`, but it also defines the ubiquitous `NULL` constant. Its constants, data types, and prototypes are contained in the `stdio.h` header file.

15.2.1 The stdio.h Header

We will not discuss the standard symbols contained in the library headers, such as `EOF` ("end of file") or `NULL`, except when the chosen way of implementation requires some additional explanations or invites some anecdotal comments.

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  stdio.h
 */

#define NULL    (void *)0
#define EOF     (-1)
```

The `FOPEN_MAX` constant should not be too large, because the stdio implementation described here allocates all file structures *statically*. `FILENAME_MAX` is an arbitrary value that was chosen out of thin air.

```
#define FOPEN_MAX       20
#define BUFSIZ          512
#define FILENAME_MAX    128
#define TMP_MAX         1
#define L_tmpnam        8
```

All symbols that begin with an *underscore* ("_") are internal to the library—this means that symbol names in user code should *never* begin with an underscore!

The `_IOUSR` constant is not described in TCPL2. It is used to signal to the library functions that a user-supplied (as opposed to implicitly allocated) buffer was assigned to a `FILE`.

```
#define _IONBF  0
#define _IOLBF  1
#define _IOFBF  2
#define _IOACC  3
#define _IOUSR  4
```

The `_FCLOSED`, `_FREAD`, `_FWRITE`, and `_FERROR` constants indicate the state and most recent operation of a `FILE`. They are used as follows:

Constant	In `_file_iom[]`	In `_file_last[]`
`_FCLOSED`	closed	has not yet been accessed
`_FREAD`	can be read	most recent operation was a read
`_FWRITE`	can be written	most recent operation was a write
`_FERROR`	n/a	I/O error or EOF reached

```
#define _FCLOSED        0
#define _FREAD          1
#define _FWRITE         2
#define _FERROR         4
```

Because SubC does not have `struct`s, we need to hack up a different solution for implementing `FILE`s. We cannot even use an array instead of a `struct`, because the array would have to be heterogeneous.[2]

The approach that was chosen here is to set up arrays of the required types and use an *index* (or *handle*), similar to a file descriptor, to access slots in the arrays. So a `FILE` value x would be used as in index to the various arrays which together form the underlying stream structure, e.g. `_file_ptr[x]` would access the "pointer" field of the simulated `struct`, and `_file_buf[x]` would be used to address the buffer of the same `FILE`.

The `FILE` type itself then becomes an alias of `int`. Of course, this means that the type-checker of the compiler cannot distinguish it from an "ordinary" `int`, but this approach seems to be the only viable option given the lack of `struct`s.

The arrays themselves are defined in the runtime initialization code (pg. 233). Its `extern` declarations are given below. The individual arrays serve the following purposes:

Array	Description
`_file_fd`	the file descriptor underlying the stream
`_file_iom`	the access mode of the stream
`_file_last`	the type of the most recent access (read, write)
`_file_mode`	the buffering mode (see `setvbuf()`, pg. 242)
`_file_ptr`	the offset of the next byte to transfer
`_file_end`	the offset of the last byte in the buffer
`_file_size`	the total size of the buffer
`_file_ch`	a `char` placed in the stream by `ungetc()`
`_file_buf`	the stream buffer or `NULL`

[2] It would have to contain different types.

The *access mode* of a stream is any combination of `_FREAD`, `_FWRITE` and `_FERROR`. When `_file_buf[x]` is `NULL`, then the corresponding `FILE` is *unbuffered*.

`FILE` values start at `(int*)1`, because `(int*)0` would be equal to `NULL`, which indicates failure in many stdio operations. So a `FILE` value of 1 is used to address `_file_fd[0]`, `_file_iom[0]`, etc.

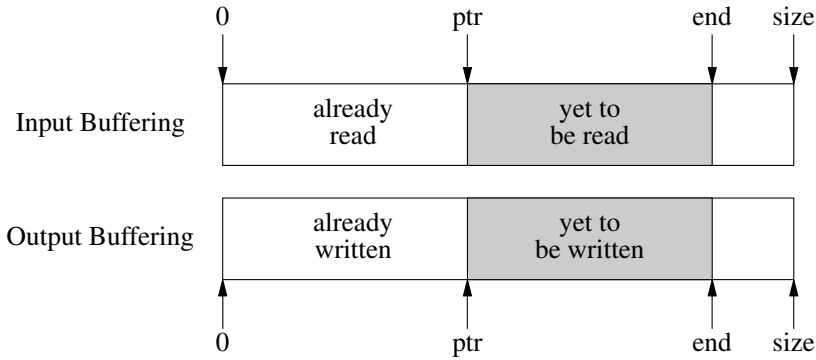


Figure 15.1: Input/Output buffering

Figure 15.1 illustrates the interaction of the `_file_ptr` (*ptr*), `_file_end` (*end*), and `_file_size` (*size*) slots of a `FILE` structure during input and output.

When reading from a fresh `FILE`, the buffer is filled by the first read on that `FILE`; *end* is set to the number of bytes read into the buffer and *ptr* is set to 0. At most *size* bytes can be read into the buffer. Reading bytes from the buffer advances *ptr* to the right. When *ptr* reaches *end*, the buffer has to be refilled and the process starts over.

When writing to a fresh `FILE`, data is first written to the buffer and each byte written to the buffer advances *end* towards *size*. When *size* is reached, the buffer has to be flushed in order to make room for new data. When writing a partial buffer, *ptr* is advanced toward *end*. This can happen when writing a line in line-buffering mode.

```
#define FILE    int

extern int       _file_fd[];
extern char      _file_iom[];
extern char      _file_last[];
extern char      _file_mode[];
extern int       _file_ptr[];
extern int       _file_end[];
```

```
extern int       _file_size[];
extern int       _file_ch[];
extern char     *_file_buf[];

extern FILE     *stdin, *stdout, *stderr;
```

SubC does not have parameterized macros, so `getc()` and `putc()` are just aliases for `fgetc()` an `fputc()`.

```
#define getc    fgetc
#define putc    fputc

#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2

void    clearerr(FILE *f);
int     fclose(FILE *f);
FILE    *fdopen(int fd, char *mode);
int     feof(FILE *f);
int     ferror(FILE *f);
int     fflush(FILE *f);
int     fgetc(FILE *f);
int     fgetpos(FILE *f, int *ppos);
char    *fgets(char *buf, int len, FILE *f);
int     fileno(FILE *f);
FILE    *fopen(char *path, char *mode);
int     fprintf(FILE *f, char *fmt, ...);
int     fputc(int c, FILE *f);
int     fputs(char *s, FILE *f);
int     fread(void *buf, int size, int count, FILE *f);
FILE    *freopen(char *path, char *mode, FILE *f);
int     fscanf(FILE *f, char *fmt, ...);
int     fseek(FILE *f, int off, int how);
int     fsetpos(FILE *f, int *ppos);
int     ftell(FILE *f);
int     fwrite(void *buf, int size, int count, FILE *f);
int     getc(FILE *f);
int     getchar(void);
char    *gets(char *buf);
void    perror(char *s);
int     printf(char *fmt, ...);
int     putc(int c, FILE *f);
```

```
int     putchar(int c);
int     puts(char *s);
int     remove(char *path);
int     rename(char *from, char *to);
void    rewind(FILE *f);
int     scanf(char *fmt, ...);
void    setbuf(FILE *f, char *buf);
int     setvbuf(FILE *f, char *buf, int mode, int size);
int     sscanf(char *s, char *fmt, ...);
int     sprintf(char *buf, char *fmt, ...);
char    *tmpnam(char *buf);
FILE    *tmpfile(void);
int     ungetc(int c, FILE *f);
```

15.2.2 Required Stdio Functions

The `remove()` function removes a file name from a directory. On a Unix system this is done by the `unlink()` system call.

```
#include <syscall.h>

int remove(char *path) {
    return _unlink(path);
}
```

The `fdopen()` function is not covered by TCPL2, but the SubC stdio library uses it to initialize `FILE`s. It is like `fopen()`, but expects an already-open file descriptor (*fd*) in the place of a file name. It allocates and initializes a `FILE` structure and returns its handle. It delegates the allocation of a stream buffer to `setvbuf()`.

When there are no more free `FILE`s, `fdopen()` sets `errno` to `ENFILE`. When the open mode (*mode*) is not a valid mode, it sets `errno` to `EINVAL`. In either case it returns `NULL`.

When allocation and initialization succeeds, `fdopen()` returns the handle of the allocated `FILE`.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int _openmode(char *mode) {
```

```
    int  n;

    if (strchr(mode, '+')) return _FREAD | _FWRITE;
    else if ('r' == *mode) return _FREAD;
    else if ('w' == *mode) return _FWRITE;
    else if ('a' == *mode) return _FWRITE;
    return _FCLOSED;
}

FILE *fdopen(int fd, char *mode) {
    int  i;

    for (i = 0; i < FOPEN_MAX; i++)
        if (_FCLOSED == _file_iom[i])
            break;
    if (i >= FOPEN_MAX) {
        errno = ENFILE;
        return NULL;
    }
    _file_buf[i]  = NULL;
    _file_fd[i]   = fd;
    _file_iom[i]  = _openmode(mode);
    _file_last[i] = _FCLOSED;
    _file_mode[i] = _IOFBF;
    _file_ptr[i]  = 0;
    _file_end[i]  = 0;
    _file_size[i] = 0;
    _file_ch[i]   = EOF;
    if (_FCLOSED == _file_iom[i]) {
        fclose((FILE *) i+1);
        errno = EINVAL;
        return NULL;
    }
    if (setvbuf((FILE *) (i+1), NULL, _IOFBF, 0) == EOF) {
        fclose((FILE *) i+1);
        return NULL;
    }
    return (FILE *) (i+1);
}
```

The `fopen()` function opens the file *path* in mode *mode*. The following modes are supported:

"r"	Open file for reading. When the file does not exist, set errno=ENOENT and return NULL.
"r+"	Like "r", but open file for reading and writing.
"w"	Create file for writing. Truncate existing file. When the file cannot be opened, set errno=EACCESS and return NULL.
"w+"	Like "w", but open for writing and reading.
"a"	Open file for writing. Create file if it does not exist. Move file pointer to EOF, so that data written to the stream will be appended. When the file cannot be opened, set errno=EACCESS and return NULL.
"a+"	Like "a", but open for writing and reading.

Note: this implementation ignores the "b" part of modes, because it does not make any sense on Unix systems. It also ignores all other characters that may follow after the above mode strings.

```
#include <stdio.h>
#include <string.h>
#include <syscall.h>
#include <errno.h>

int _openfd(char *path, char *mode) {
    int  i, plus, fd = -1;

    plus = strchr(mode, '+')? 1: 0;
    if ('w' == *mode) {
        fd = _creat(path, 0666);
        if (fd < 0) errno = EACCESS;
        if (fd >= 0 && plus) {
            _close(fd);
            fd = _open(path, 2);
        }
    }
    else if ('r' == *mode) {
        fd = _open(path, plus? 2: 0);
        if (fd < 0) errno = ENOENT;
    }
    else if ('a' == *mode) {
        fd = _open(path, plus? 2: 1);
        if (fd < 0) {
            fd = _creat(path, 0644);
            if (fd < 0) errno = EACCESS;
            _close(fd);
```

```
            fd = _open(path, plus? 2: 1);
        }
        _lseek(fd, 0, SEEK_END);
    }
    else {
        errno = EINVAL;
    }
    return fd;
}

FILE *fopen(char *path, char *mode) {
    int     fd;
    FILE    *f;

    fd = _openfd(path, mode);
    if (fd < 0) return NULL;
    return fdopen(fd, mode);
}
```

The `setvbuf()` function sets up the buffer and buffering mode of the stream f. *Buf* is a user-supplied buffer, *mode* is the desired buffering mode, and *size* is the size of the user-supplied buffer (or the size of the automatically allocated buffer when *buf* is `NULL`).

`setvbuf()` deallocates the current buffer of f, if it is not `NULL` and not user-supplied.

When *mode* is `_IONBF`, buffering is disabled for the stream f. When called with *buf=NULL* and a *mode* other than `_IONBF`, then a buffer of the requested size will be allocated. When *size* is 0, a default size (`BUFSIZ`) will be used.

When the requested buffer could not be established, `setvbuf()` returns `EOF` (and probably sets `errno` to `ENOMEM`). When the request could be fulfilled, it returns 0.

```
#include <stdio.h>
#include <stdlib.h>

static void freebuf(int fi) {
    if (_file_buf[fi] != NULL && (_file_mode[fi] & _IOUSR)
        == 0
    )
        free(_file_buf[fi]);
```

```
}

int setvbuf(FILE *f, char *buf, int mode, int size) {
    int  fi;

    fi = (int) f-1;
    if (0 == size) size = BUFSIZ;
    if (buf) {
        freebuf(fi);
        _file_mode[fi] = mode | _IOUSR;
    }
    else {
        if (_IONBF != mode)
            if ((buf = malloc(size)) == NULL)
                return EOF;
        freebuf(fi);
        _file_mode[fi] = mode;
    }
    _file_buf[fi] = buf;
    _file_size[fi] = size;
    return 0;
}
```

The `fclose()` function closes the stream f and makes its underlying structure available to `fopen()` again. `fclose(NULL)` is a null-operation. The `fclose()` function flushes the stream buffer when the stream has recently been written to (its `_file_last` slot equals `_FWRITE`). It then closes the underlying file descriptor and releases the `FILE` structure by setting its I/O mode (`_file_iom`) to `_FCLOSED`. The `fclose()` function fails when f is already closed or its buffer cannot be flushed. It returns `EOF` on failure and 0 on success.

```
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>

int fclose(FILE *f) {
    int  fi;

    if (NULL == f) return EOF;
    fi = (int) f-1;
    if (_FCLOSED == _file_iom[fi]) return EOF;
```

```
    if (_FWRITE == _file_last[fi] && fflush(f))
        return EOF;
    _close(_file_fd[fi]);
    if (_file_buf[fi] != NULL && (_file_mode[fi] & _IOUSR)
        == 0
    )
        free(_file_buf[fi]);
    _file_iom[fi] = _FCLOSED;
    return 0;
}
```

The `fflush()` function flushes the stream f. When f equals `NULL`, it flushes all open streams. Flushing a stream in output mode means to write the buffer to the underlying file descriptor. Flushing a stream in input mode means to discard all pending input from the buffer. When flushing a stream fails, `fflush()` sets `errno` to `EIO` and returns `EOF`. Otherwise it returns 0.

```
#include <stdio.h>
#include <syscall.h>
#include <errno.h>

static int _fflush(FILE *f) {
    int  fi, p, e;

    fi = (int) f-1;
    if (_file_iom[fi] & _FERROR) return 0;
    _file_ch[fi] = EOF;
    if (_file_last[fi] != _FWRITE) {
        _file_ptr[fi] = _file_end[fi] = 0;
        return 0;
    }
    if ((_file_mode[fi] & _IOACC) == _IONBF) return 0;
    p = _file_ptr[fi];
    e = _file_end[fi];
    _file_ptr[fi] = _file_end[fi] = 0;
    if (_write(_file_fd[fi], _file_buf[fi] + p, e-p) == e-p)
        return 0;
    errno = EIO;
    return -1;
}
```

```
int fflush(FILE *f) {
    int  i, rc = 0;

    if (f != NULL) return _fflush(f);
    for (i = 0; i < FOPEN_MAX; i++)
        if (!_fflush((FILE *) (i+1))) rc = -1;
    return rc;
}
```

Reading a buffered stream is a rather complex operation. It is performed by `fread()` and its helpers. The `fread()` function delegates its work to `_fread()`, which is the principal input function of the stdio library. The *size* and *count* arguments of `fread()` are simply multiplied, because meaningful file operations can only be performed in the `int` range anyway in SubC—due to the lack of a larger integer type.

The `_refill()` function re-fills a buffer when it has run dry, i.e. when its *ptr* value has reached its *end* value. The function reads a new chunk, sets *end* to the number of bytes read, and resets *ptr* to zero. When an I/O error occurs or no further bytes could be read, it sets the `_FERROR` flag of the stream f. In case of an error, it also sets `errno` to `EIO`. It returns a flag indicating whether the buffer could be re-filled.

The `_fread()` function is similar to `fread()`, but accepts just a *size* parameter instead of a (*count*,*size*) pair. The function returns immediately when the stream f is not readable or its `_FERROR` flag is set. It sets the "last operation" slot to `_FREAD` and then proceeds to fulfill the read request.

The first input source it checks is the *ungetc buffer* of the input stream (the `_file_ch[]` slot). When there is a byte in this buffer, the function transfers it to the destination buffer.

When the input stream is in non-buffering mode, it performs a single `_read()` to fulfill a read request. For buffered streams, it reads bytes from the input buffer and when the number of bytes in the buffer is not sufficient, it proceeds to read directly from the underlying file until the number of bytes still to read drops below the size of a buffer. It then fills one additional buffer and transfers the last chunk of the request from the input buffer to the destination buffer. Surplus characters may remain in the buffer.

There is quite a bit of byte shuffling involved and the code expresses the details probably better than any prose that could be invented at this point.

```
#include <stdio.h>
#include <string.h>
```

```
#include <syscall.h>
#include <errno.h>

int _refill(FILE *f) {
    int  fi, k;

    fi = (int) f-1;
    if (_file_ptr[fi] >= _file_end[fi]) {
        _file_ptr[fi] = 0;
        k = _read(_file_fd[fi], _file_buf[fi],
                    _file_size[fi]);
        if (k <= 0) {
            _file_end[fi] = 0;
            _file_iom[fi] |= _FERROR;
            if (k < 0) errno = EIO;
            return 0;
        }
        _file_end[fi] = k;
    }
    return 1;
}

int _fread(void *p, int size, FILE *f) {
    int  fi, k, len, end, ptr, total;

    fi = (int) f-1;
    if ((_file_iom[fi] & _FREAD) == 0) return 0;
    if (_file_iom[fi] & _FERROR) return 0;
    _file_last[fi] = _FREAD;
    total = size;
    if (_file_ch[fi] != EOF) {
        *(char *)p++ = _file_ch[fi];
        _file_ch[fi] = EOF;
        size--;
    }
    if ((_file_mode[fi] & _IOACC) == _IONBF) {
        if ((total = _read(_file_fd[fi], p, size)) != size) {
            _file_iom[fi] |= _FERROR;
            errno = EIO;
        }
        return total;
    }
    if (!_refill(f)) return 0;
```

```
    end = _file_end[fi];
    ptr = _file_ptr[fi];
    k = end - ptr;
    if (size < k) k = size;
    memcpy(p, _file_buf[fi] + ptr, k);
    _file_ptr[fi] += k;
    p += k;
    size -= k;
    len = _file_size[fi];
    while (size > len) {
        if (_read(_file_fd[fi], p, len) != len) {
            _file_iom[fi] |= _FERROR;
            errno = EIO;
            return total-size;
        }
        p += len;
        size -= len;
    }
    if (size != 0) {
        if (!_refill(f)) return total-size;
        memcpy(p, _file_buf[fi], size);
        _file_ptr[fi] = size;
    }
    return total;
}

int fread(void *p, int size, int count, FILE *f) {
    int  k, fi;

    if ((k = _fread(p, size * count, f)) < 0)
        return -1;
    return k / size;
}
```

The `feof()` function returns a flag indicating whether the *EOF* ("end of file") was reached while reading an input stream. The `_FERROR` flag is used to indicate the EOF on input streams.

```
#include <stdio.h>

int feof(FILE *f) {
    return (_file_iom[(int) f-1] & _FERROR) != 0;
```

```
}
```

The `fgetc()` function reads a single character from the input stream f and returns it. Like `fread()` it attempts to fail early, i.e. it returns failure (`EOF`) immediately when a program tries to read a non-readable stream. The alternative would be to let the code fail at the next attempt to read the underlying file descriptor.

`fgetc()` does not call `_fread()`, but duplicates some of its code in order to limit the damage done to the performance of the library.

```
#include <stdio.h>
#include <syscall.h>
#include <errno.h>

int _refill(FILE *f);

int fgetc(FILE *f) {
    int     fi;
    char    c, b[1];

    fi = (int) f-1;
    if ((_file_iom[fi] & _FREAD) == 0)
        return EOF;
    if (_file_iom[fi] & _FERROR)
        return EOF;
    _file_last[fi] = _FREAD;
    if (_file_ch[fi] != EOF) {
        c = _file_ch[fi];
        _file_ch[fi] = EOF;
        return c;
    }
    if ((_file_mode[fi] & _IOACC) == _IONBF)
        if (_read(_file_fd[fi], b, 1) == 1)
            return *b;
        else {
            errno = EIO;
            return EOF;
        }
    if (_file_ptr[fi] >= _file_end[fi])
        if (!_refill(f))
            return EOF;
    return _file_buf[fi][_file_ptr[fi]++];
```

```
}
```

The `fgets()` function reads a newline-delimited string of a length of up to $len-1$ characters into the buffer s. It does so by scanning the input buffer for a *newline character* and refilling it repeatedly until either a newline character is found or the maximum number of characters has been transferred to the destination buffer s. It attaches a trailing NUL character and returns a pointer to s.

When the stream f is in non-buffering mode, `fgets()` calls the `fgets_raw()` function, which reads every single character with a separate `_read()` call. Full-blown C libraries probably do this in a cleverer way.

```
#include <stdio.h>
#include <string.h>
#include <syscall.h>
#include <errno.h>

int _refill(FILE *f);

static char *fgets_raw(char *s, int len, int fi) {
    char    *p;

    p = s;
    while (len-- > 1) {
        if (_read(_file_fd[fi], p, 1) != 1) {
            errno = EIO;
            return NULL;
        }
        if ('\n' == *p++) break;
    }
    *p = 0;
    return s;
}

char *fgets(char *s, int len, FILE *f) {
    int     fi, *pptr, *pend, k;
    char    *p, *buf, *pn;

    fi = (int) f-1;
    if ((_file_iom[fi] & _FREAD) == 0) return NULL;
    if (_file_iom[fi] & _FERROR) return NULL;
    _file_last[fi] = _FREAD;
```

```
    p = s;
    if (_file_ch[fi] != EOF) {
        *p++ = _file_ch[fi];
        _file_ch[fi] = EOF;
        len--;
    }
    if ((_file_mode[fi] & _IOACC) == _IONBF)
        return fgets_raw(s, len, fi);
    pptr = _file_ptr + fi;
    pend = _file_end + fi;
    buf = _file_buf[fi];
    pn = NULL;
    while (len > 1 && pn == NULL) {
        if (!_refill(f))
            return NULL;
        if ((pn = memchr(buf + *pptr, '\n', *pend - *pptr))
            != NULL
        )
            k = pn - buf - *pptr + 1;
        else
            k = *pend - *pptr;
        if (len-1 < k) k = len-1;
        memcpy(p, buf + *pptr, k);
        *pptr += k;
        p += k;
        len -= k;
    }
    *p = 0;
    return s;
}
```

Writing to a buffered stream is about as complex as reading from one, and understanding the logic of one will help to understand the logic of the other. The `fwrite()`/`_fwrite()` pair is to writing what `fread()`/`_fread()` is to reading.

The `_fsync()` function writes a buffer to the underlying file descriptor when the buffer becomes full (its *end* reaches its *size*). Syncing an output buffer is further complicated by the fact that the output stream may be in *line buffering mode*, where only full lines are written at once, and output containing a newline character must be flushed immediately. So `_fsync()` flushes the buffer when it is full and also when the stream f is in line-buffering mode *and* the buffer contains a newline character. In the latter

case, all characters up to and including the *last* newline character in the buffer are written and then the *ptr* of *f* is advanced to the character after the last newline. In all other cases syncing a buffer simply means to write all characters in the buffer between *ptr* and *end*.

Writing to a buffered stream involves the following steps: Write to the buffer until the buffer overflows and then flush the buffer. Write subsequent output directly to the underlying file descriptor until the amount of bytes still to write drops below the size of the buffer. The remaining characters are then written to the buffer for later output. When the output stream is in line-buffering mode, it will be synced at the end of the write request, resulting in complete lines to be emitted.

```
#include <stdio.h>
#include <string.h>
#include <syscall.h>
#include <errno.h>

int _fsync(FILE *f) {
    int     fi, ptr, end, k, pn, p;
    char    *buf;

    fi = (int) f-1;
    if (_file_end[fi] >= _file_size[fi]) {
        ptr = _file_ptr[fi];
        end = _file_end[fi];
        _file_end[fi] = _file_ptr[fi] = 0;
        k = _write(_file_fd[fi], _file_buf[fi] + ptr,
                   end-ptr);
        if (k != end-ptr) {
            _file_iom[fi] |= _FERROR;
            errno = EIO;
            return 0;
        }
    }
    else if ((_file_mode[fi] & _IOACC) == _IOLBF) {
        ptr = _file_ptr[fi];
        end = _file_end[fi];
        buf = _file_buf[fi];
        pn = -1;
        for (p = ptr; p < end; p++)
            if ('\n' == buf[p]) pn = p+1;
        if (pn >= 0) {
```

```
            k = _write(_file_fd[fi], _file_buf[fi] + ptr,
                       pn-ptr);
            _file_ptr[fi] = pn;
            if (k != pn - ptr) {
                _file_iom[fi] |= _FERROR;
                errno = EIO;
                return 0;
            }
        }
    }
    return 1;
}

int _fwrite(void *p, int size, FILE *f) {
    int  fi, k, len, total;

    fi = (int) f-1;
    if ((_file_iom[fi] & _FWRITE) == 0) return 0;
    if (_file_iom[fi] & _FERROR) return 0;
    _file_last[fi] = _FWRITE;
    if ((_file_mode[fi] & _IOACC) == _IONBF) {
        if ((k = _write(_file_fd[fi], p, size)) != size) {
            _file_iom[fi] |= _FERROR;
            errno = EIO;
        }
        return k;
    }
    total = size;
    len = _file_size[fi];
    k = len - _file_end[fi];
    if (size < k) k = size;
    memcpy(_file_buf[fi] + _file_end[fi], p, k);
    _file_end[fi] += k;
    size -= k;
    p += k;
    if (!_fsync(f)) return 0;
    while (size > len) {
        if ((k = _write(_file_fd[fi], p, len)) != len) {
            _file_iom[fi] |= _FERROR;
            errno = EIO;
            return total - size;
        }
        p += len;
```

```
        size -= len;
    }
    if (size != 0) {
        memcpy(_file_buf[fi], p, size);
        _file_end[fi] = size;
    }
    if ((_file_mode[fi] & _IOACC) == _IOLBF && !_fsync(f))
        return total-size;
    return total;
}

int fwrite(void *p, int size, int count, FILE *f) {
    int  k, fi;

    if ((k = _fwrite(p, size * count, f)) < 0)
        return -1;
    return k / size;
}
```

The `fputc()` function writes a single character to the given output stream. Like `fgetc()`, it duplicates some I/O logic in order to increase performance.

```
#include <stdio.h>
#include <syscall.h>
#include <errno.h>

int _fsync(FILE *f);

int fputc(int c, FILE *f) {
    int     fi;
    char    b[1];

    fi = (int) f-1;
    if ((_file_iom[fi] & _FWRITE) == 0)
        return EOF;
    if (_file_iom[fi] & _FERROR)
        return EOF;
    _file_last[fi] = _FWRITE;
    if ((_file_mode[fi] & _IOACC) == _IONBF) {
        *b = c;
        if (_write(_file_fd[fi], b, 1) == 1)
```

```
            return c;
        else {
            errno = EIO;
            return EOF;
        }
    }
    if (_file_end[fi] >= _file_size[fi])
        if (!_fsync(f))
            return EOF;
    _file_buf[fi][_file_end[fi]++] = c;
    if ((_file_mode[fi] & _IOACC) == _IOLBF)
        if (!_fsync(f))
            return EOF;
    return c;
}
```

We use a rather cheap implementation of `putchar()`.

```
#include <stdio.h>

int putchar(int c) {
    return fputc(c, stdout);
}
```

The `_vformat()` function is the swiss army knife of output functions. It forms the basis for the `fprintf()`, `printf()`, and `sprintf()` functions and it expects the following parameters:

`int mode`	output mode, 0=string, 1=stream
`int max`	max. buffer size for string output or 0 (unlimited)
`void *dest`	destination buffer (mode 0) or stream (mode 1)
`char *fmt`	format string to control output formatting
`void **varg`	arguments to be formatted

When *mode* is 0, a `char` array of at least *max* characters must be passed in the *dest* argument. When *mode* is 1, *dest* must be an output `FILE`. The `_vformat()` function returns the number of arguments successfully written.

Its uses the following internal variables:

`FILE *outf;`	output file (in mode 1)
`int olen;`	number of characters written so far
`int limit;`	maximum number of characters to output
`char *vbuf;`	output buffer (in mode 1)
`int err;`	error flag

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int _fwrite(void *buf, int len, FILE *f);

static FILE *outf;
static int  olen;
static int  limit;
static char *vbuf;
static int  err;
```

The `itoa()` helper function converts the integer n to a string and writes its output to the buffer p. Note that p must point to the *end* of the buffer initially, because the function writes its output in reverse order. The *base* parameter specifies the numeric base (8,10,16) for the operation. The *sgnch* parameter points to the sign character to be used for positive numbers. This character is left unchanged when n is in fact positive and changed to "-" when n is negative. The sign is *not* included in the output. The `itoa()` function returns a pointer to the first character of the converted number.

```
static char *itoa(char *p, int n, int base, char *sgnch) {
    int  s = 0, a = 'a';

    if (base < 0) base = -base, a = 'A';
    if (n < 0) s = 1, n = -n;
    *--p = 0;
    while (n || 0 == *p) {
        *--p = n % base + '0';
        if (n % base > 9) *p += a-10-'0';
        n /= base;
    }
    if (s) *sgnch = '-';
    return p;
}
```

The `append()` helper function appends the string *what* to the current output string in mode 0. It does not allow the output string to grow past a size of `limit` characters, truncating it as necessary. When `limit` is zero, unlimited output can be appended.[3]

[3] Yes, this may cause a buffer overflow.

In mode 1, `append()` simply writes *what* to `outf`. In either case, it adds the number of appended characters to `olen`.

```
static void append(char *what, int len) {
    int  k;

    if (outf) {
        if (_fwrite(what, len, outf) != len)
            err = 1;
        olen += len;
    }
    else if (0 == limit) {
        memcpy(vbuf + olen, what, len);
        olen += len;
        vbuf[olen] = 0;
    }
    else {
        k = limit-olen-1;
        len = len < k? len: k;
        memcpy(vbuf + olen, what, len);
        olen += len;
        vbuf[olen] = 0;
    }
}
```

The `_vformat()` function itself, finally, is an interpreter for a subset[4] of `printf()`-style formatted output sequences as specified in TCPL2. It simply emits all characters in *fmt* except for "`%`", which is used to introduce *format specifiers* of the following form:

```
% {+,-,0,#, } {*|digits} {c,d,n,o,p,s,x,X,%}
```

The three groups of a format specifier are called the "flags", the "field width", and the "conversion character". The conversion character is mandatory, the other groups are optional. In output, format specifiers are replaced by the renditions of the objects described by them. Note that format specifiers do not really include blanks between fields (as inserted above to make the format more readable). E.g. `%5d` would be used to describe a decimal field of five characters. Figure 15.2 lists a brief summary of flags and conversions.

[4]The floating point conversions are missing.

Flags	
`+`	Print a plus sign in front of positive numbers (default: print no sign).
blank	Print a blank character in front of positive numbers (default: print no leading character).
`-`	Justify output to the left margin (default: right-justify).
`0`	Pad numeric fields with zeros instead of blanks.
`#`	Select alternative output format (prefix octal numbers with "`0`" and hexa-decimal numbers with `0x` or `0X`).

Conversions	
`c`	Print the next argument as a single character.
`d`	Print the next argument as a decimal number.
`n`	Insert the number of characters written to the output as a decimal number; do not consume an argument.
`o`	Print the next argument as an octal number.
`p`	Print the next argument as a pointer (this implementation prints pointers as hexa-decimal numbers).
`s`	Print the next argument as a string, i.e. simply copy that string to output, maybe with some padding.
`x`,`X`	Print the next argument as a hexa-decimal number; when using a capital "`X`" specifier, use upper-case letters to represent the digits 10 through 16.
`%`	Print a literal "`%`" character without consuming an argument.

Figure 15.2: Flags and conversions of `printf()`

There are a few subtle points to observe when implementing the formatting of non-decimal numbers in `_vformat()` (and hence in `printf()` and friends). For instance, any leading padding characters always precede the sign and the optional numeric base prefix (like `0x`), except when the padding characters are zeros. In this case the padding comes between the base prefix and the number itself, e.g. the format specifier `%+#10x` would print the number 0xfff as

```
____+0xFFF
```

but the specifier `%+#010x` would print it as

```
+0x0000FFF
```

See the code for details.

Note: the extra arguments of `_vformat`, which are passed to the function in the *varg* argument, are in *reverse* order, so *varg* has to be *decremented* in order to consume an argument.

```
int _vformat(int mode, int max, void *dest, char *fmt,
                void **varg)
{
    #define BUFLEN  256

    char    lbuf[BUFLEN], *p, *end;
    int     left, len, alt, k;
    char    pad[1], sgnch[2], *pfx;
    int     na = 0;

    end = &lbuf[BUFLEN];
    sgnch[1] = 0;
    if (0 == mode) {
        outf = NULL;
        vbuf = dest;
        *vbuf = 0;
    }
    else {
        outf = dest;
        err = 0;
    }
    olen = 0;
    limit = max;
    while (*fmt) {
        left = len = 0;
        *sgnch = 0;
        pfx = "";
        if ('%' == *fmt && fmt[1] != '%') {
            fmt++;
            *pad = ' ';
            alt = 0;
            while (strchr("-+0 #", *fmt)) {
                if ('-' == *fmt) {
                    left = 1, *pad = ' ';
                    fmt++;
                }
                else if ('0' == *fmt) {
                    if (!left) *pad = '0';
                    fmt++;
```

```
        }
        else if ('#' == *fmt) {
            alt = 1;
            fmt++;
        }
        else if ('+' == *fmt) {
            *sgnch = '+';
            fmt++;
        }
        else if (' ' == *fmt) {
            if (!*sgnch) *sgnch = ' ';
            fmt++;
        }
    }
    if ('*' == *fmt)
        len = (int) *varg--, fmt++;
    else
        while (isdigit(*fmt))
            len = len * 10 + *fmt++ - '0';
    switch (*fmt++) {
    case 'c':
        *pad = ' ';
        *sgnch = 0;
        lbuf[0] = (char) *varg--;
        lbuf[1] = 0;
        p = lbuf;
        na++;
        break;
    case 'd':
        p = itoa(end, (int) *varg--, 10, sgnch);
        na++;
        break;
    case 'n':
        p = itoa(end, olen, 10, sgnch);
        break;
    case 'o':
        p = itoa(end, (int) *varg--, 8, sgnch);
        if (alt) pfx = "0";
        na++;
        break;
    case 's':
        *sgnch = 0;
        *pad = ' ';
```

```
                p = *varg--;
                na++;
                break;
            case 'p':
            case 'x':
            case 'X':
                k = 'X' == fmt[-1]? -16: 16;
                p = itoa(end, (int) *varg--, k, sgnch);
                if (alt) pfx = k<0? "0X": "0x";
                na++;
                break;
            }
        }
        else {
            if ('%' == *fmt) fmt++;
            lbuf[0] = *fmt++;
            lbuf[1] = 0;
            p = lbuf;
        }
        k = strlen(p) + strlen(pfx) + strlen(sgnch);
        if ('0' == *pad) {
            if (*sgnch) append(sgnch, 1);
            append(pfx, strlen(pfx));
            pfx = "";
        }
        while (!left && len-- > k)
            append(pad, 1);
        if (*sgnch && *pad != '0') {
            append(sgnch, 1);
            append(pfx, strlen(pfx));
        }
        if (!*sgnch) append(pfx, strlen(pfx));
        append(p, strlen(p));
        while (left && len-- > k)
            append(pad, 1);
        if (outf && err) break;
    }
    return na;
}
```

The `fprintf()`, `printf()`, and `sprintf()` functions are just wrappers that call `_vformat()` with different parameters. Note that these functions have rather unusual types, e.g. `fprintf()` is defined as

```
fprintf(void *last, ...)
```

but declared[5] as

```
fprintf(FILE *f, char *fmt, ...)
```

This has to be done due to SubC's non-standard calling conventions, which require the implementation to choose a different approach for variable-argument functions. Because arguments are pushed to the stack from the left to the right, they are in the wrong order to be processed in the usual way, because the *last* argument is the argument that is located at the lowest stack-relative address in the call frame of the callee. See figure 15.3 for the stack frame of the following sample call:

```
fprintf(file, "%d %s %p\n", 123, "hello, world!", 0xdec0de);
```

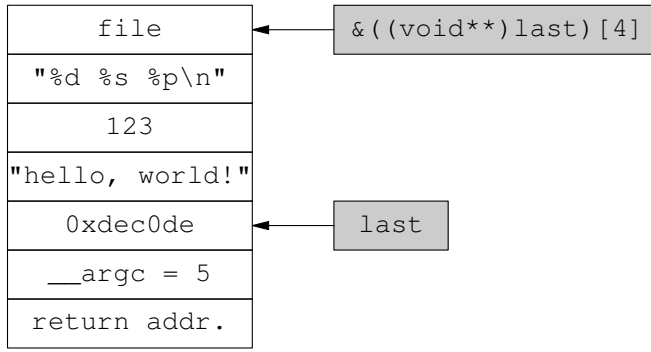


Figure 15.3: The call frame of an `fprintf()` function call

So the *last* parameter of `fprintf()` and friends is in fact the first argument passed to the function (in a traditional C call frame it would be the last). To find the address of the first argument, we need to know the total number of arguments that has been passed to the function. This is why SubC passes this extra information to each function. We can extract it with the `__argc` keyword. So the address of the first argument would be:

```
&((void**)last)[__argc-1]
```

Because arguments are passed in left-to-right order, we have to *decrement* the argument pointer in order to access the other arguments. This is

[5]SubC allows to redeclare a variadic function with a different signature in order to support this unusual approach.

basically what `fprintf()` and the related functions do: extract their own arguments (like the output `FILE` of `fprintf()` or the destination buffer of `sprintf()`) and then pass the proper parameters and the remaining arguments on to `_vformat()`.

```
#include <stdio.h>

extern int _vformat(int mode, int max, void *dest, char *fmt,
                    void **varg);

int fprintf(void *last, ...) {
    void    **args;
    FILE    *f;
    char    *fmt, *p;

    args = &last;
    args += __argc;
    f = *--args;
    fmt = *--args;
    return _vformat(1, 0, f, fmt, --args);
}
```

```
#include <stdio.h>

extern int _vformat(int mode, int max, void *dest, char *fmt,
                    void **varg);

int printf(void *last, ...) {
    void    **args;
    char    *fmt, *p;

    args = &last;
    args += __argc;
    fmt = *--args;
    return _vformat(1, 0, stdout, fmt, --args);
}
```

```
#include <stdio.h>
#include <string.h>

extern int _vformat(int mode, int max, void *dest, char *fmt,
                    void **varg);
```

```
int sprintf(void *last, ...) {
    void    **args;
    char    *buf;
    char    *fmt, *p;

    args = &last;
    args += __argc;
    buf = *--args;
    fmt = *--args;
    return _vformat(0, 0, buf, fmt, --args);
}
```

15.3 Utility Library

15.3.1 The stdlib.h Header

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  stdlib.h
 */

#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0

#define RAND_MAX    65535

extern char **environ;

void  abort(void);
int   abs(int n);
int   atexit(int (*fn)());
int   atoi(char *s);
void  *bsearch(void *key, void *array, int count, int size,
               int (*cmp)());
void  *calloc(int count, int size);
void  exit(int rc);
void  free(void *p);
char  *getenv(char *name);
void  *malloc(int size);
void  qsort(void *array, int count, int size, int (*cmp)());
int   rand(void);
```

```
void  *realloc(void *p, int size);
void  srand(int seed);
int   strtol(char *s, char *endp[], int base);
int   system(char *s);
```

15.3.2 Required Stdlib Functions

The `abort()` function is invoked by `malloc()` and `free()` when a corrupt arena is detected.

```
#include <stdlib.h>
#include <signal.h>

void abort(void) {
    raise(SIGABRT);
}
```

We could probably do `abs()` without a conditional branch, but would it be portable? E.g.: `n & INT_MAX`.

```
#include <stdlib.h>

int abs(int n) {
    return n<0? -n: n;
}
```

The `atexit()` implementation can only register one single function, because we cannot define an array of `int(*)()` in SubC. Of course, we could work around it with `void` pointers, but is it worth the effort?

```
#include <stdlib.h>
#include <errno.h>

int (*_exitfn)() = 0;

int atexit(int (*fn)()) {
    if (_exitfn) {
        errno = ENOMEM;
        return -1;
    }
    _exitfn = fn;
```

```
    return 0;
}
```

The `exit()` function invokes the function registered by `atexit()`, closes all `FILE` streams, and exits.

```
#include <stdlib.h>
#include <stdio.h>
#include <syscall.h>

extern int  (*_exitfn)();

void exit(int rc) {
    int  i;

    if (_exitfn) _exitfn();
    for (i = 0; i < FOPEN_MAX; i++)
        fclose((FILE *) (i+1));
    _exit(rc);
}
```

The `getenv()` function is not really required by the compiler, but it is included here to illustrate access to the environment variables of the process. The function uses the `environ` variable, which is declared `extern` in `stdlib.h` and defined in the C startup module (pg. 221ff).

The `environ` pointer points to an array of strings of the form

```
"name=value"
```

The `getenv()` function searches the array for a string whose `name` part matches its *name* parameter and returns the corresponding `value` part. When no match can be found, it returns `NULL`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *getenv(char *name) {
    char    **p;
    int     k;
```

```
    k = strlen(name);
    for (p = environ; *p; p++) {
        if (strlen(*p) > k &&
            '=' == (*p)[k] &&
            !strncmp(*p, name, k-1)
        ) {
            return (*p) + k+1;
        }
    }
    return NULL;
}
```

The SubC version of `malloc()` is a simple yet robust general-purpose memory allocator. Like most simple allocators it exhibits exponential allocation time, but with a rather small coefficient (see figure 15.4).[6]

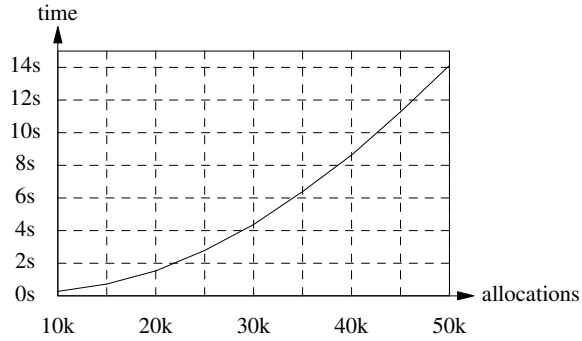


Figure 15.4: The time complexity of `malloc()`

The allocator assumes that you will either allocate a lot of small blocks (below `THRESHOLD`) or only a few large blocks. When allocating large blocks, it will move the break (i.e. allocate storage from the operating system) on each allocation, which is slow. When allocating small regions, it will preallocate 50 times the requested size in order to speed up future allocations.

The time performance of `malloc()` could easily be improved by omitting defragmentation, but at the cost of increased memory usage.[7]

Defragmentation in this case means to connect subsequent segments of free memory to form a single segment. For instance, there may be three consecutive free segments in the pool, each with a size of 3K bytes. Neither of these segment could be used to fulfill a request of 5K bytes, though. So

[6] Measured on a 600MHz CPU, allocating random-sized blocks of up to 128 bytes, i.e. below the default `THRESHOLD`.

[7] The author is an old fart, though, and hence values space more than time.

the first thing that `malloc()` does when it runs out of free space is to call the "defragger", which will connect the three segments to form one single piece of free memory with a size of 9K bytes. It will then promptly re-fragment this piece into a 5K and a 4K segment and returns the 5K segment to fulfill the request. Defragmentation is what the below `defrag()` function does.

The static variables `_arena` and `_asize` hold a pointer to and the size of `malloc`'s *arena*, which is just an archaic term for its memory pool. The `freep` variable points to the most recently allocated segment.

Without memorizing the point of the last allocation in `freep`, the time complexity of `malloc()` would be $O(\frac{n^2}{2})$ when allocating large sequences of small regions. This simple trick avoids performance degradation in this not-so-special case.

```
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>
#include <errno.h>

#define THRESHOLD   128

int         *_arena = 0;
int         _asize;
static int  *freep;

static void defrag(void) {
    int  *p, *q, *end;

    end = _arena + _asize;
    for (p = _arena; p < end; p += abs(*p)) {
        if (*p > 0) {
            for (q = p; q < end && *q > 0; q += *q)
                ;
            *p = q - p;
        }
    }
}
```

Note that the arena is an array of `int`. This approach has a few advantages:

- All allocated items are naturally aligned at machine word boundary, so we do not have to adjust them later.

- We can store the sizes of free and used segments in the `int`s of the arena itself.

When `malloc()` is first called, the arena is empty (`NULL`), so it allocates one by calling `_sbrk()`, inserts its size in its first `int`, and sets up `freep`.

Allocation is done in three passes. In each pass the allocator scans all segments of the arena and attempt to find one that begins with a value that is larger than the requested size. Each segment consists of an `int` containing the size of the segment (called the *segment descriptor*) and the corresponding number of subsequent `int`s (minus one for the descriptor itself). When a suitable segment is found, `malloc()` marks it as used by negating the sign of its segment descriptor. It also splits the segment up when it is larger then the requested size plus 1 (again, because one `int` is needed for the descriptor itself).

`malloc()` starts scanning the arena at `freep` and a pass ends when it ends up at `freep` again. The scan wraps around at the end of the arena. When no free segment was found in the first pass, the arena is defragmented and another pass is made. When it also fails, more memory is requested from the operating system.

When `malloc()` cannot fulfill a request, it sets `errno` to `ENOMEM` and returns `NULL`. When the segment pointer points somewhere outside of the arena at any time during a scan, `malloc()` prints a message and calls `abort()`, assuming that a segment descriptor has been overwritten. It also aborts when it finds a segment descriptor with a value of 0 (which would cause the algorithm to cycle indefinitely).

```
void *malloc(int size) {
    int  *p, *end;
    int  k, n, tries;

    size = (size + sizeof(int) - 1) / sizeof(int);
    if (NULL == _arena) {
        if (size >= THRESHOLD)
            _asize = size + 1;
        else
            _asize = size * 50;
        _arena = _sbrk(_asize * sizeof(int));
        if (_arena == (int *)-1) {
            errno = ENOMEM;
            return NULL;
        }
```

```
        _arena[0] = _asize;
        freep = _arena;
    }
    for (tries = 0; tries < 3; tries++) {
        end = _arena + _asize;
        p = freep;
        do {
            if (*p > size) {
                if (size + 1 == *p) {
                    *p = -*p;
                }
                else {
                    k = *p;
                    *p = -(size+1);
                    p[size+1] = k - size - 1;
                }
                freep = p;
                return p+1;
            }
            p += abs(*p);
            if (p == end) p = _arena;
            if (p < _arena || p >= end || 0 == *p) {
                _write(2, "malloc(): corrupt arena\n", 24);
                abort();
            }
        } while (p != freep);
        if (0 == tries)
            defrag();
        else {
            if (size >= THRESHOLD)
                n = size + 1;
            else
                n = size * 50;
            if (_sbrk(n * sizeof(int)) == (void *)-1) {
                errno = ENOMEM;
                return NULL;
            }
            k = _asize;
            _asize += n;
            *end = _asize - k;
        }
    }
    errno = ENOMEM;
```

```
    return NULL;
}
```

The `free()` function frees a memory region that has previously been allocated by `malloc()`. Freeing `NULL` is a null-operation.

```
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>

extern int  *_arena, _asize;

void free(void *p) {
    int  *pi;

    if (NULL == p) return;
    pi = p;
    if (pi < _arena || _arena + _asize < pi || pi[-1] >= 0) {
        _write(2, "bad pointer in free()\n", 22);
        abort();
    }
    --pi;
    *pi = -*pi;
}
```

The `system()` function is used by the compiler controller to invoke external commands like the system *assembler* and *linker*. It uses the classic `fork()`/`execve()`/`wait()` method and passes the command *cmd* to `/bin/sh` for execution. When `NULL` is passed to `system`, it checks the availability of the shell by letting it execute `exit`. This implementation is full of *Unixisms*; the author does not believe that it can be implemented in a portable way.

```
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>

int system(char *cmd) {
    int     pid, rc;
    char    *argv[4];
```

```
    if ((pid = _fork()) == -1) {
        return -1;
    }
    else if (pid) {
        _wait(&rc);
        return cmd? rc: !rc;
    }
    else {
        argv[0] = "/bin/sh";
        argv[1] = "-c";
        argv[2] = cmd? cmd: "exit";
        argv[3] = NULL;
        _execve(argv[0], argv, environ);
        exit(cmd? -1: 0);
    }
}
```

15.4 String Library

15.4.1 The string.h Header

```
/*
 *  NMH's Simple C Compiler, 2011,2012
 *  string.h
 */

extern char *sys_errlist[];
extern int  sys_nerr;

void  *memchr(void *p, int c, int len);
int   memcmp(void *p1, void *p2, int len);
void  *memcpy(void *dest, void *src, int len);
void  *memmove(void *dest, void *src, int len);
void  *memset(void *p, int c, int len);
char  *strcat(char *s, char *a);
char  *strchr(char *s, int c);
int   strcmp(char *s1, char *s2);
char  *strcpy(char *dest, char *src);
int   strcspn(char *s, char *set);
char  *strdup(char *s);
char  *strerror(int err);
int   strlen(char *s);
```

```
char  *strncat(char *s, char *a, int len);
int   strncmp(char *s1, char *s2, int len);
char  *strncpy(char *dest, char *src, int len);
char  *strpbrk(char *s, char *set);
char  *strrchr(char *s, int c);
int   strspn(char *s, char *set);
char  *strtok(char *s, char *sep);
```

15.4.2 Required String Functions

The `memchr()` function returns a pointer to the first occurrence of the byte c in the memory region p of the length n. It returns `NULL` when the byte is not contained in the region.

The string and memory functions should really be implemented in assembly language. Their portable implementations would benefit largely from an *optimizer* or even from a simple *code synthesizer*, as described in part IV (pg. 281ff) of this book.

```
#include <stdio.h>
#include <string.h>

void *memchr(void *p, int c, int n) {
    while (n--) {
        if (*(char *)p == c) return p;
        p++;
    }
    return NULL;
}
```

The `memcpy()` function copies n bytes from the s ("source") to the d ("destination") region. The expression `*s++` casts s below, because SubC refuses to compile code that dereferences `void` pointers.

```
#include <string.h>

void *memcpy(void *d, void *s, int n) {
    char    *p;

    p = d;
    while (n--) *p++ = *(char *)s++;
```

```
    return d;
}
```

The `memset()` function fills n bytes starting at p with the byte c.

```
#include <string.h>

void *memset(void *p, int c, int n) {
    char    *q;

    q = p;
    while (n--) *(char *)p++ = c;
    return q;
}
```

The `strcat()` function appends the string a ("appendix") to the string d ("destination") and returns d.

```
#include <string.h>

char *strcat(char *d, char *a) {
    char    *p;

    for (p = d; *p; p++)
        ;
    while (*a) *p++ = *a++;
    *p = 0;
    return d;
}
```

The `strchr()` function returns a pointer to the first occurrence of the character c in the string s, or `NULL` when there is no c in s.

This implementation returns a pointer to the terminating NUL character when $c = 0$. Note that this is not mandated by TCPL2, but it seems to be common practice.

```
#include <stdio.h>
#include <string.h>

char *strchr(char *s, int c) {
```

```
    while (*s && *s != c)
        s++;
    return *s || !c? s: NULL;
}
```

The `strcmp()` function compares the strings $s1$ and $s2$ and returns the difference between their first differing characters (or 0 when they do not differ).

```
#include <string.h>

int strcmp(char *s1, char *s2) {
    while (*s1 && *s1 == *s2)
        s1++, s2++;
    return *s1 - *s2;
}
```

The `strcpy()` function copies the string s to d and returns d.

```
#include <string.h>

char *strcpy(char *d, char *s) {
    char    *p;

    p = d;
    while (*s) *p++ = *s++;
    *p = 0;
    return d;
}
```

The non-standard (but common) `strdup()` function creates a copy of s and returns it. When no memory can be allocated for a copy of s, it returns `NULL`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *strdup(char *s) {
    char    *p;
```

```
    if ((p = malloc(strlen(s)+1)) == NULL)
        return NULL;
    strcpy(p, s);
    return p;
}
```

The `strlen()` function returns the number of characters in the string s (excluding the delimiting NUL).

```
#include <string.h>

int strlen(char *s) {
    char    *p;

    for (p = s; *p; p++)
        ;
    return p - s;
}
```

The `strncpy()` function copies at most n characters of the string s to d and returns d. Note that `strncpy()` does *not* terminate d after copying *exactly* n characters (assuming insufficient space in d).

```
#include <string.h>

char *strncpy(char *d, char *s, int n) {
    char    *p;

    for (p = d; n && *s; n--)
        *p++ = *s++;
    if (n) *p = 0;
    return d;
}
```

15.5 Character Types

15.5.1 The ctype.h Header

```
/*
 *  NMH's Simple C Compiler, 2011
```

```
 *  ctype.h
 */

int  isalnum(int c);
int  isalpha(int c);
int  iscntrl(int c);
int  isdigit(int c);
int  isgraph(int c);
int  islower(int c);
int  isprint(int c);
int  ispunct(int c);
int  isspace(int c);
int  isupper(int c);
int  isxdigit(int c);
int  tolower(int c);
int  toupper(int c);
```

15.5.2 The Ctype Functions

The *character type* ("*ctype*") functions are implemented by looking up characters in the static `ctypes[]` array. These functions really should be macros.

Each entry of the `ctypes[]` array contains a flag for each property of the corresponding character. For instance, `ctypes[65]` contains the flags `G` ("graph"), `U` ("upper case"), and `X` ("hex letter"), because the character "`A`" (whose ASCII code is 65) is an upper case graphical (printable) character that can represent a hexa-decimal digit.

```
#include <ctype.h>

enum {
    U = 0x01,   /* upper case */
    L = 0x02,   /* lower case */
    C = 0x04,   /* control */
    D = 0x08,   /* decimal */
    X = 0x10,   /* hex letter */
    P = 0x20,   /* punctuation */
    S = 0x40,   /* space */
    G = 0x80    /* graph */
};

static char ctypes[] = {
```

```
    C,      C,      C,      C,      C,      C,      C,      C,
    C,      C|S,    C|S,    C|S,    C|S,    C|S,    C,      C,
    C,      C,      C,      C,      C,      C,      C,      C,
    C,      C,      C,      C,      C,      C,      C,      C,
    S,      G|P,    G|P,    G|P,    G|P,    G|P,    G|P,    G|P,
    G|P,    G|P,    G|P,    G|P,    G|P,    G|P,    G|P,    G|P,
    G|D,    G|D,    G|D,    G|D,    G|D,    G|D,    G|D,    G|D,
    G|D,    G|D,    G|P,    G|P,    G|P,    G|P,    G|P,    G|P,
    G|P,    G|U|X,  G|U|X,  G|U|X,  G|U|X,  G|U|X,  G|U|X,  G|U,
    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,
    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,    G|U,
    G|U,    G|U,    G|U,    G|P,    G|P,    G|P,    G|P,    G|P,
    G|P,    G|L|X,  G|L|X,  G|L|X,  G|L|X,  G|L|X,  G|L|X,  G|L,
    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,
    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,    G|L,
    G|L,    G|L,    G|L,    G|P,    G|P,    G|P,    G|P,    C,
    };

int isalnum(int c) { return 0 != (ctypes[c&127] & (D|U|L));}
int isalpha(int c) { return 0 != (ctypes[c&127] & (U|L)); }
int iscntrl(int c) { return 0 != (ctypes[c&127] & C); }
int isdigit(int c) { return 0 != (ctypes[c&127] & D); }
int isgraph(int c) { return 0 != (ctypes[c&127] & G); }
int islower(int c) { return 0 != (ctypes[c&127] & L); }
int isprint(int c) { return c == ' ' || isgraph(c); }
int ispunct(int c) { return 0 != (ctypes[c&127] & P); }
int isspace(int c) { return 0 != (ctypes[c&127] & S); }
int isupper(int c) { return 0 != (ctypes[c&127] & U); }
int isxdigit(int c){ return 0 != (ctypes[c&127] & (D|X)); }

int tolower(int c) {
    return isupper(c)? c - 'A' + 'a': c;
}

int toupper(int c) {
    return islower(c)? c - 'a' + 'A': c;
}
```

15.6 The errno.h Header

The `errno.h` header contains the declaration of the global `errno` variable (defined in `init.c`, pg. 233), which is set by various library functions to

identify the cause of an error. It also defines various constants for the values that `errno` may assume.

The SubC implementation of the `errno` mechanism is rather crude and does not take the more specific codes generated by the underlying *operating system* into consideration; the `errno` variable of SubC is completely independent from the same variable in the system's C library. Some SubC library functions set `errno` to one of the below values to indicate the cause of failure.

EOK	no error
ENOENT	no such file
EACCESS	access denied
EIO	input/output error
ENFILE	too many open files
EINVAL	invalid argument
ENOMEM	out of memory

This implementation is more a code sample than a helpful mechanism. Caveat utilitor.

```
/*
 *  NMH's Simple C Compiler, 2012
 *  errno.h
 */

exern int errno;

#define EOK     0
#define ENOENT  1
#define EACCESS 2
#define EIO     3
#define ENFILE  4
#define EINVAL  5
#define ENOMEM  6
```

Part IV

Beyond SubC

Chapter 16

Code Synthesis

The SubC *back-end* discussed in part II of this textbook was a rather simple one that just generated fixed code fragments for various operations. While such a back-end is easy to implement, easy to port, and its correctness is easy to verify, it has, of course, one major drawback: the quality of the emitted code is rather low. This is because it imposes a simple, stack-based model of computation on a non-stack-based processor. The vast majority of modern processors are RISC or CISC machines with a variety of different addressing modes. The SubC back-end does not make use of these modes, hence its code quality suffers, both in size and run time.

To improve code quality, we have to do *code synthesis*. Code synthesis is the process that generates the proper instructions for each individual context instead of using a pre-defined fragment. For instance, the expression `a+b` would generate the following output in SubC:

```
movl    Ca,%eax
pushl   %eax
movl    Cb,%eax
popl    %ecx
addl    %ecx,%eax
```

while any programmer in their right mind would probably code it as

```
movl    Ca,%eax
addl    Cb,%eax
```

However, writing such code mechanically requires to choose the proper instruction for adding the value of a global variable to the primary register. The back-end must select the "add global" instruction based on context

information like the type of `b`. This is basically what a synthesizing code generator does.

Of course the number of fragments to be chosen from explodes in this case. The SubC generator has a set of load instructions for loading literal, indirect, local, local static, and public/static objects. It also has a set of store instructions for the same storage classes, giving a total of eight different load/store instructions. When synthesizing machine code, we will have to combine all the above addressing modes with each instruction that we intend to emit:

- Add literal, indirect, local, local static, global operands;
- Subtract literal, indirect, local, local static, global operands;
- Multiply literal, indirect, local, local static, global operands;
- Left-shift literal, indirect, local, local static, global operands;
- Compare literal, indirect, local, local static, global operands;
- etc...

Of course we also need to synthesize instructions for performing these operations on bytes and we need to synthesize instructions for doing pointer arithmetics. We already have seen how complex this process can become in the discussion of code generation for the increment operators (pg. 179ff).

In the next section we will study a simple approach that increases code quality significantly. Not all of the above combinations will be covered, though. This part of the book describes code synthesis at a rather abstract level and only outlines the involved algorithms and data structures. Filling in the details will be left to the reader.

16.1 Instruction Queuing

Code quality can be enhanced massively by postponing code generation by one single instruction. Instead of emitting the fragment linked to an operation immediately, we store it in a one-element *instruction queue* and emit it only when the next instruction arrives. This process expands our view of the instruction stream from one single instruction to two subsequent instructions. Let us see how this works.

The following listing reproduces a tiny compiler for a tiny subset of C expressions, covering just the following operators: `* / + -` (including both the unary and binary "-"). Factors are limited to single-letter variables, where lower-case letters indicate locals and upper-case letters denote globals.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char    *P;
int     A;

void emit(int i, int x) {
        if ('l' == i || 'g' == i) {
                if (A) emit('p', 0);
                A = 1;
        }
        switch (i) {
        case '_': printf("negr A\n");
                  break;
        case '+': printf("pop  X\n");
                  printf("addr X,A\n");
                  break;
        case '*': printf("pop  X\n");
                  printf("mulr X,A\n");
                  break;
        case '-': printf("pop  X\n");
                  printf("swap A,X\n");
                  printf("subr X,A\n");
                  break;
        case '/': printf("pop  X\n");
                  printf("swap A,X\n");
                  printf("divr X,A\n");
                  break;
        case 'g': printf("lg   _%c,A\n", x);
                  break;
        case 'l': printf("ll   _%c,A\n", x);
                  break;
        case 'p': printf("push A\n");
                  break;
        }
}

void skip(void) {
        while (isspace(*P)) P++;
}
```

```
void factor(void) {
        skip();
        if ('-' == *P) {
                P++;
                factor();
                emit('_', 0);
        }
        else if (isupper(*P))
                emit('g', *P++);
        else
                emit('l', *P++);
}

void term(void) {
        skip();
        factor();
        for (;;) {
                skip();
                switch (*P) {
                case '*': P++;
                          factor();
                          emit('*', 0);
                          break;
                case '/': P++;
                          factor();
                          emit('/', 0);
                          break;
                default:  return;
                }
        }
}

void sum(void) {
        skip();
        term();
        for (;;) {
                skip();
                switch (*P) {
                case '+': P++;
                          term();
                          emit('+', 0);
                          break;
                case '-': P++;
```

```
                        term();
                        emit('-', 0);
                        break;
                default:  return;
                }
        }
}

void expr(char *s) {
        P = s;
        A = 0;
        sum();
}

int main(int argc, char **argv) {
        expr(argc>1? argv[1]: "");
        return EXIT_SUCCESS;
}
```

The code implements a simple, traditional *recursive descent parser*, which is rather uninteresting and just included for the sake of completeness. The only function that concerns us is the `emit()` function of the program. It implements a simple, fragment-based code generator, exactly like the one that SubC actually uses. Instead of 386 code, though, it emits some symbolic code using the following instructions (where s = source, d = destination, and M = addressing mode):

Instruction	Description
`lM s,d`	load s to d
`swap s,d`	swap s and d
`push s`	push s
`pop d`	pop top of stack into d
`addM s,d`	add s to d
`subM s,d`	subtract s from d
`mulM s,d`	multiply s by d, result in d
`divM s,d`	divide d by s, result in d
`neg d`	negate d

All instructions with an M in their name support the addressing modes r (register), l (local), and g (global), e.g. the `ll` instruction would load a local variable to a register, `subg` would subtract a global variable, and `mulr` would multiply two registers.

We will assume for now that the target has two registers, A and X. The

compiler will generate code that is similar to that emitted by SubC, e.g. the expression `x+-Y/z` will translate to the following code (leading underscores denote symbols).

```
ll   _x,A
push A
lg   _Y,A
neg  A
push A
ll   _z,A
pop  X
swap A,X
divr X,A
pop  X
addr X,A
```

The goal of a synthesizing code generator would be to generate instructions that make use of the addressing modes available on the target platform. For instance, it should generate

```
ll   _a,A
addl _b,A
```

instead of

```
ll   _a,A
push A
ll   _b,A
pop  X
addr X,A
```

This can be done by looking at two subsequent symbols passed to the emitter. When parsing the expression `a+b`, the emitter will see the following input: *(l,a) (l,b) (+,0)*, denoting the operations *load local a*, *load local b*, and *add*. When the first *load local* instruction arrives, we put it into a single-element queue and generate no code. When the second *load local* arrives, the queue is full, so we have to flush it first, resulting in the emission of a *load* instruction that loads a to the primary register. Now we can put the (l, b) instruction into the queue. The *add* instruction arrives next. This is where the actual code synthesis starts.

When the *add* arrives, a *load local* instruction with the argument b is in the queue. So the synthesizer generates an *add* instruction with the same addressing mode as the *load* operation in the queue, thereby rendering the *load* itself redundant. It simply generates *addl* $\_b,A$ and clears the queue.

The following listing implements the emitter described above. It lists only the differences to the previous listing to avoid unnecessary redundancy.[1]

First we add some generator functions to emit synthesized instructions, `gen()` for fixed fragments and `gen2()` for parameterized fragments.

```
int     Qi = 0, Qx;

void gen(char *s) {
        puts(s);
}

void gen2(char *s, int a) {
        printf(s, a);
        putchar('\n');
}
```

The `synth()` function performs the actual code synthesis. When the operand in the queue (`Qx`) is upper case, it synthesizes an instruction with a global operand, when `Qx` is lower case, it generates an instruction with a local operand, and when the queue is empty (`Qi==0`), it fetches the operand from the stack and generates a register-register instruction. In this case it also emits code to flip the operands of the "subtract" and "divide" instructions.

Note that `synth()` has to provide a lot of templates for generating all the different kinds of instructions, even if we support only four instructions and three addressing modes. If we would generate code for the full SubC language, we would have 16 binary operators, five addressing modes (literal, local, local static, public/static, stack[2]), two operand sizes, and pointer arithmetics for the sum operators. Even if we ignore pointer arithmetics for now, that would make $16 \times 5 \times 2 = 160$ different templates only for the basic binary operators (still excluding unary operators, assignment operators, short-circuit operators, etc).

[1] Sic!

[2] For operands that have been spilled earlier.

```
void synth(int i) {
        int     g;

        g = isupper(Qx);
        if (!Qi) gen("pop  X");
        switch (i) {
        case '+': if (!Qi)
                        gen("addr X,A");
                  else if (g)
                        gen2("addg _%c,A", Qx);
                  else
                        gen2("addl _%c,A", Qx);
                  break;
        case '*': if (!Qi)
                        gen("mulr X,A");
                  else if (g)
                        gen2("mulg _%c,A", Qx);
                  else
                        gen2("mull _%c,A", Qx);
                  break;
        case '-': if (!Qi) {
                        gen("swap A,X");
                        gen("subr X,A");
                  }
                  else if (g)
                        gen2("subg _%c,A", Qx);
                  else
                        gen2("subl _%c,A", Qx);
                  break;
        case '/': if (!Qi) {
                        gen("swap A,X");
                        gen("divr X,A");
                  }
                  else if (g)
                        gen2("divg _%c,A", Qx);
                  else
                        gen2("divl _%c,A", Qx);
                  break;
        }
        Qi = 0;
}
```

The `commit()` function commits the instruction in the queue by generating a *load* instruction. It also sets the accumulator flag (`A`) to indicate that the value in the primary register is important. Like the SubC back-end it spills the accumulator to the stack when two values have to be loaded in succession. After committing an instruction the queue is clear again.

```
void commit(void) {
        if (A) gen("push A");
        switch (Qi) {
        case 'l': gen2("ll   _%c,A", Qx);
                  break;
        case 'g': gen2("lg   _%c,A", Qx);
                  break;
        }
        Qi = 0;
        A = 1;
}
```

The `queue()` function just queues an instruction, committing the one in the queue first if the queue is not empty.

```
void queue(int i, int x) {
        if (Qi) commit();
        Qi = i;
        Qx = x;
}
```

What is most interesting about the improved back-end is that we can just keep the old interface. The new `emit()` function takes exactly the same parameters as the one of the non-synthesizing back-end. It just works in a different way internally: load operations are queued, unary operations are committed immediately, and binary operations are synthesized.

```
void emit(int i, int x) {
        switch (i) {
        case 'l':
        case 'g': queue(i, x);
                  break;
        case '_': commit();
                  gen("neg  A");
                  break;
```

```
        default:  synth(i);
                  break;
        }
}
```

The algorithm used in the synthesizing back-end is not much more complex than the actual SubC back-end. The only point that has to be taken care of is where to commit the instructions in the queue. For instance, when calling a procedure, the last argument may still be in the queue when the call is generated, so the it must be committed first. However, the author is not going to spoil the fun at this point by giving too many hints!

The real work involved in the creation of a synthesizing back-end is to differentiate all the instructions for the various contexts and to write the corresponding templates. For real-world synthesizing generators, though, the approach illustrated here will soon become unmanageable, and the templates would have to move to a data structure, or some logic would have to be invented to create them on demand.

The code quality of synthesizing generators is much better than that of the simple-minded SubC generator as the following comparison demonstrates:

```
SubC              Synthesizing        Input Expression:
Generator         Generator
---------         ------------        a * b - c * d
ll   _a,A         ll   _a,A
push A            mull _b,A
ll   _b,A
pop  X
mulr X,A
push A            push A
ll   _c,A         ll   _c,A
push A
ll   _d,A         mull _d,A
pop  X
mulr X,A
pop  X            pop  X
swap A,X          swap A,X
subr X,A          subr X,A
```

16.1.1 CISC versus RISC

The code synthesis algorithm outlined here works best for *CISC* ("Complex Instruction Set Computer") architectures, and even then it produces the best results on mostly accumulator-based machines, like the *8086*, where there are virtually no general purpose registers except for the accumulator.[3]

On a *RISC* ("Reduced Instruction Set Computer') machine, though, things would look completely different. There is normally a large set of general purpose registers available with few—if any—limitations on their use. On the other hand, all the fancy addressing modes are typically limited to the *load/store* operations, while everything else is performed in register-register operations, which makes sense given the large amount of available registers. So the difference of code quality between the naïve back-end and the synthesizing back-end appears not to be as large on a RISC as on a CISC architecture:

```
Simple          CISC            RISC
Generator       Synthesis       Synthesis
---------       ---------       ---------
ll   _a,A       ll   _a,A       ll   _a,A
push A          mull _b,A       ll   _b,X
ll   _b,A                       mull X,A
pop  X
mulr X,A
push A          push A          push A
ll   _c,A       ll   _c,A       ll   _c,A
push A          mull _d,A       ll   _d,X
ll   _d,A                       mull X,A
pop  X
mulr X,A
pop  X          pop  X          pop  X
swap X,A        swap X,A        swap X,A
subr X,A        subr X,A        subr X,A
```

However, just counting the instruction does not do justice to the synthesizer. Even on a RISC machine, where operands have to be loaded in separate instructions, the generator avoids four *push* and *pop* instructions, which are register-memory operations and hence particularly slow. This is especially true on RISC machines which typically have very fast

[3]The 8086 cannot even perform indirection through AX, CX, or DX, so the compiler loses another general-purpose register for indirection (one of BX, SI, DI).

register-register instructions, so the difference between register-register and register-memory operations weights even heavier.

We can improve the synthesizing generator a bit further to make better use of existing registers, so it will generate much better code for RISC machines and slightly better code for CISC machines. We will get to that after the break.

16.1.2 Comparisons and Conditional Jumps

Another area where code synthesis can improve the emitted code significantly is the generation of conditional branches. For the expression `1==2`, the SubC generator will emit the following code:

```
        movl    $1,%eax
        pushl   %eax
        movl    $2,%eax
        xorl    %edx,%edx
        popl    %ecx
        cmpl    %eax,%ecx
        jne     L2
        inc     %edx
L2:     movl    %edx,%eax
```

Of course a synthesizing back-end would eliminate the *push*/*pop* instructions and render the following code:

```
        movl    $1,%eax
        xorl    %edx,%edx
        cmpl    $2,%eax
        jne     L2
        inc     %edx
L2:     movl    %edx,%eax
```

However, there is still a problem: the code always generates a truth value indicating the result of the comparison and stores it in the accumulator, but this result is not always needed. Consider the following code for the statement `if (1==2) 3;`

```
        movl    $1,%eax
        xorl    %edx,%edx
        cmpl    $2,%eax
```

```
        jne     L2
        inc     %edx
L2:     movl    %edx,%eax
        or      %eax,%eax
        jnz     L4
        jmp     L3
L4:     movl    $3,%eax
L3:
```

Because the result of the comparison is only used for a conditional jump there is no need to generate a normalized truth value first and then testing that normalized value in order to perform a conditional jump. An assembly language programmer would rather write:[4]

```
        movl    $1,%eax
        cmpl    $2,%eax
        je      L4
        jmp     L3
L4:     movl    $3,%eax
L3:
```

So the entire normalize-and-move part can theoretically be omitted. A synthesizing code generator can do this, too. Instead of committing a compare instruction immediately, it would move it to the instruction queue and wait for the subsequent instruction. If that instruction is a branch, the normalize-and-move part does not have to be generated, and a branch can follow directly. Of course in this case the code generator would have to synthesize the *branch-on-condition* instructions for all comparison operators that exist in the C language. Normalization still has to be done when the instruction following a comparison is not a branch, but then statements of the form

```
a = b == c;
```

are rather rare in C, so the above approach should find ample opportunity for code improvement. Note that this approach would also improve the code of all loop constructs as well as the code of the short-circuit operations `&&` and `||` and the code of the `?:` operator.

It would not make much sense on some RISC machines,[5] though, whose comparison instructions store normalized truth values in registers anyway.

[4] Assuming that the distance between L4 and L3 cannot be covered by a short jump.
[5] Like the AXP 21164.

16.2 Register Allocation

The SubC back-end takes great care of using no more registers than actually needed:

`%eax`	Primary register (accumulator)
`%ecx`	Auxiliary register, used for stack operands and shifting
`%edx`	Auxiliary register, used for indirection in post-increments and for truth value normalization

We cannot unify `%ecx` and `%edx` because they interact in some code fragments[6] (q.v. pg. 292). This leaves us with three registers that would be free for general-purpose use: `%ebx`, `%esi`, and `%edi`. The SubC back-end and low-level runtime support code make sure that none of these registers is used otherwise, so everything is in fact ready for using these registers in a more useful way than just letting them sit there.

The process of assigning registers to temporary values is called *register allocation*. Unlike memory allocation, register allocation has to be performed at compile time, so the compiler has to know which registers are available at each point of a computation. The simplest approach to keeping track of used and free registers is to treat them like a stack (a *register stack*).

In fact our naïve accumulator-based generator with its stack-based model of computation already performs some very crude kind of register allocation by *caching* the *TOS* in the accumulator. When the `A` variable of SubC's code generator is zero, the accumulator is clear and otherwise it is used. When the accumulator is allocated, it holds the top-most element of the stack. We can easily extend this model to cache the topmost n elements of the stack.

The following listing is once again a modification of the synthesizer shown initially in this part of the book. It modifies the second version, the first synthesizing generator. This model replaces the accumulator A with a set of N general purpose registers named $R1$ through RN. The variable `R` points to the register that is currently used as the accumulator; `R=0` means no register.

```
int     R = 0;
int     N = 4;
```

[6] This does not invalidate our general model of computation, because we could spill one of the X registers to memory; we just use an additional register because it is there.

The new generator functions generate the name of the current accumulator from the `R` variable. The `genr()` function applies an operation to the current accumulator and the "previous" accumulator, i.e. the second element on the stack.

```
void gen2(char *s, int a) {
        printf(s, a, R);
        putchar('\n');
}

void genr(char *s) {
        printf(s, R, R-1);
        putchar('\n');
}
```

Pushing a value now just means to select the next free register. When we run out of registers: game over for now. We will get back to this later.

```
void push(void) {
        if (++R > N) {
                fprintf(stderr, "out of registers\n");
                exit(1);
        }
}
```

Popping a value means to select the previous register.

```
void pop(void) {
        R--;
}
```

The synthesizer has to be modified a bit to generate instructions that use the new register names instead of A. It also calls `pop()` at the end, because the second stack element is already cached in $R-1$, so we do not actually need to pop it off the stack before using it.

```
void synth(int i) {
        int     g;

        g = isupper(Qx);
```

```
        switch (i) {
        case '+': if (!Qi)
                        genr("addr R%d,R%d");
                  else if (g)
                        gen2("addg _%c,R%d", Qx);
                  else
                        gen2("addl _%c,R%d", Qx);
                  break;
        case '*': if (!Qi)
                        genr("mulr R%d,R%d");
                  else if (g)
                        gen2("mulg _%c,R%d", Qx);
                  else
                        gen2("mull _%c,R%d", Qx);
                  break;
        case '-': if (!Qi) {
                        genr("swap R%d,R%d");
                        genr("subr R%d,R%d");
                  }
                  else if (g)
                        gen2("subg _%c,R%d", Qx);
                  else
                        gen2("subl _%c,R%d", Qx);
                  break;
        case '/': if (!Qi) {
                        genr("swap R%d,R%d");
                        genr("divr R%d,R%d");
                  }
                  else if (g)
                        gen2("divg _%c,R%d", Qx);
                  else
                        gen2("divl _%c,R%d", Qx);
                  break;
        }
        if (!Qi) pop();
        Qi = 0;
}
```

Loading a value always involves a "push" operation now, because we have to select a new register. We may visualize this as a register selector that is "pushed" toward higher register numbers by `push()` and "popped"[7]

[7] This is probably the point where the metaphor breaks down.

toward lower register numbers by `pop()`. This will make more sense in the next version of the code generator.

```
void load(void) {
        push();
        switch (Qi) {
        case 'l': gen2("ll   _%c,R%d", Qx);
                  break;
        case 'g': gen2("lg   _%c,R%d", Qx);
                  break;
        }
        Qi = 0;
}
```

Let us see how the register-allocating generator performs:

```
Generated Code      Input expression:
--------------
ll   _a,R1          a * b - c * d
mull _b,R1
ll   _c,R2
mull _d,R2
swap R2,R1
subr R2,R1
```

This is probably as close to hand-coded assembly language as you can get, and the above code uses only two registers. Unfortunately, further experiments are limited because our toy compiler supports only two levels of precedence. Let us fix this by adding expression grouping to the parser:

```
void sum(void);

void factor(void) {
        skip();
        if ('(' == *P) {
                P++;
                sum();
                P++;
        }
        else if ('-' == *P) {
                P++;
```

```
                factor();
                emit('_', 0);
        }
        else if (isupper(*P))
                emit('g', *P++);
        else
                emit('l', *P++);
}
```

We can now group expression to the right to tickle the register allocator a bit more:

```
Generated Code      Input expression:
--------------
ll   _a,R1          a + (b + (c + (d + e)))
ll   _b,R2
ll   _c,R3
ll   _d,R4
addl _e,R4
addr R4,R3
addr R3,R2
addr R2,R1
```

Looks good, but one more level of parentheses and the code generator will break because it ran out of registers. This is, of course, not acceptable for a real-world compiler back-end. Even if N is much larger than 4 on modern RISC machines, and even if we could spill to memory by inventing pseudo-registers, all these solutions are not really satisfying. The final back-end will combine the techniques of both synthesizing generators to create a model that will generate the same code as the present generator as long as sufficient registers are present and recycle registers when they are running short.

16.2.1 Cyclic Register Allocation

The idea behind *cyclic register allocation* is to combine the register stack with the hardware stack. Registers are allocated on the register stack and when no more registers are available, one of them is spilled to the hardware stack and thereby made available to the allocator again. Because the (ideal) stack is unbounded, this allocator cannot run out of registers. Effectively, this approach turns the register stack into a *cyclic queue* or a *ring*: when

the last register has been allocated, the first one is pushed to the stack and R is reset to the first register in the queue. The changes to the code are limited to the `push()` and `pop()` functions. We also introduce a new variable, S, which keeps track of the depth of the hardware stack.

```
int     S = 0;
```

Pushing a value now works as follows:

- When there are no more free registers ($R \geq N$), push $R1$, select $R1$, and increase the stack counter.
- When any registers have been spilled to the stack ($S \neq 0$), push the next register, select it, and increase the stack counter.
- Otherwise just select the next free register.

```
void push(void) {
        if (R >= N) {
                R = 1;
                gen("push R%d");
                S++;
        }
        else if (S) {
                R++;
                gen("push R%d");
                S++;
        }
        else {
                R++;
        }
}
```

Popping a value works accordingly:

- When there is no previous register to select ($R < 2$), pop the current register, select RN, and decrease the stack counter.
- When any registers have been spilled to the stack ($S \neq 0$), pop the current register, decrease the stack counter, and select the previous register.
- Otherwise just select the previous register.

```
void pop(void) {
        if (R < 2) {
                gen("pop  R%d");
                R = N;
                S--;
        }
        else if (S) {
                gen("pop  R%d");
                R--;
                S--;
        }
        else {
                R--;
        }
}
```

Sample output for inputs with a deep nesting of right-associative operations is given below:

```
a+(b+(c+(d+(e+f))))
ll   _a,R1
ll   _b,R2
ll   _c,R3
ll   _d,R4
push R1
ll   _e,R1
addl _f,R1
addr R1,R4
pop  R1
addr R4,R3
addr R3,R2
addr R2,R1
```

```
a+(b+(c+(d+(e+(f+(g+h))))))
ll   _a,R1
ll   _b,R2
ll   _c,R3
ll   _d,R4
push R1
ll   _e,R1
push R2
ll   _f,R2
push R3
ll   _g,R3
addl _h,R3
addr R3,R2
pop  R3
addr R2,R1
pop  R2
addr R1,R4
pop  R1
addr R4,R3
addr R3,R2
addr R2,R1
```

Of course in the special case of the "+" operator we could re-arrange the input expression, making use of the commutative nature of the operation, but this would lead us to the realm of optimization, which we will explore in the next chapter.

The code synthesizers introduced in this chapter generate quite good code with little effort. They are easy to comprehend and implement and, because they are susceptible to table-based approaches for code templates, even easy to re-target. The 80/20 rule applies at this point: with little (20%) effort we got a generator that generates pretty good (80%) code. To get another 20% of improvement, we will need to invest another 80% of work (and the 80/20 rule applies recursively). From this point on, code generation becomes increasingly complex and the involved techniques are far beyond the scope of this book.

Chapter 17

Optimization

17.1 Peephole Optimization

Peephole optimization is called so because it takes a very limited view on the program to optimize, typically at instruction-level after generating machine code or assembly language. Code generators often emit code fragments with parameters that render an instruction redundant. For example, they may emit the fragment `addl $%d,%eax` with a parameter of zero, yielding an instruction that adds zero to the primary register.[1] A peephole optimizer would find such an instruction and remove it.

The strength of a peephole optimizer is that it works at the level of actual instructions, so while its view is limited, it is also very concrete. Modern architectures, especially CISCy ones, offer plenty of opportunities to replace sequences of simple instructions with fewer, more complex, more powerful ones. This is why peephole optimizers are often table-driven and use pattern matching to find and replace sequences of instructions.

The simplest case that is handled by peephole optimization is the removal of *redundant instructions* as outlined above. Some typical candidates for removal would be all kinds of instructions with *neutral operands*, like sums with a constant zero operand, logical "or"s with zero, logical "and"s with `~0`, etc:

Pattern	Replacement
`addl $0,R`	
`subl $0,R`	
`orl  $0,R`	
`andl $0xffffffff,R`	

[1] This may happen, for instance, when computing `&a[0]`.

In the above *optimization templates*, R denotes any register, so no matter what register is involved in the operation, a peephole optimizer would always be able to identify the operation and eliminate it. Of course redundant sequences may span multiple instructions and a peephole optimizer should be able to identify these, too. For example:

Pattern	Replacement
`movl $1,R` `mull R`	
`movl $1,R` `idivl R`	
`movl $0,%ecx` `shll %cl,R`	

In the first two cases in the above table it is important to notice that R must always denote the same register in a multi-instruction pattern, so

```
movl $1,R     matches     movl $1,%ecx     but not     movl $1,%ecx
mull R                    mull %ecx                    mull %esi
```

The next step would be to replace instructions for which alternatives with fewer bytes and/or fewer instruction cycles exist, like replacing an *add* instruction by a constant factor of 1 by an increment instruction. Here are a few examples:

Pattern	Replacement
`addl $1,R`	`incl R`
`subl $1,R`	`decl R`
`addl $-1,R`	`decl R`
`movl $0,R`	`xorl R,R`
`movl $1,R`	`xorl R,R` `incr R`
`xorl $0xffffffff,R`	`notl R`

Even when replacing sequences, both the pattern and the associated replacement may span multiple instructions:

Pattern	Replacement
`cmpl $0,R` `jz L`	`orl R,R` `jz L`
`movl $4,R` `mull R`	`shll $2,R`
`movl $8,R` `idivl R`	`sarl $3,R`

An L in a pattern indicates a *label*, i.e. a destination address for a

jump instruction. Some optimization rules may add additional constraints, like the following ones that replace a conditional jump-around-jump by a conditional short jump:

Pattern	Replacement	Constraint

`L1: jz L2`
`jmp L3`
`L2:`
	`jnz L3`	$-128 < abs(\$L1 - \$L3) < 127$

`L1: jl L2`
`jmp L3`
`L2:`
	`jge L3`	$-128 < abs(\$L1 - \$L3) < 127$

On the 386 architecture the replacement of a conditional short jump around a long jump by a complementary conditional short jump is only possible if the distance between the jump instruction and its destination fits in a single byte.

Finally, the peephole optimizer could replace some typical sequences generated to access array elements with single *move* instructions using some of the more sophisticated addressing modes of the 386:

Pattern	Replacement
`addl N,R` `movl (R),R`	`movl N(R),R`

`shll $2,R2`
`addl R2,R1`
`movl (R1),R1`
	`movl (R1,R2,4),R1`

The last of the above replacement instructions would load the value stored at the address $R1 + R2 * 4$ into $R1$, doing all the scaling and adding internally. Again, the $R1$'s and $R2$'s must each refer to the same register in the pattern.

Sometimes it makes sense to perform multiple passes of peephole optimization and add additional optimization rules to match already-processed input from prior passes. For instance, the first instruction of the sequence

```
movl $0,%ecx
shll %cl,%eax
```

might get matched by the following rule (the left side is still the "pattern" and the right one the "replacement"):

`movl $0,R`	`xorl R,R`

resulting in the sequence

```
xorl %ecx,%ecx
shll %cl,%eax
```

which no longer matches

`movl $0,%ecx` `shll %cl,R`	

So the optimizer would miss the chance to eliminate a redundant sequence completely. In many cases this can be avoided by ordering the rules carefully. Other cases, though, require to make multiple passes. For example, elimination and simplification typically shorten instruction sequences, which may create further opportunities for the replacement of jump-around-jumps by short jumps. So a peephole optimizer should perform at least two passes over the program.

BTW, a peephole optimizer could also solve the issue introduced by the `geninc()` function, which may emit duplicate instructions in some cases (pg. 182). A rule of the form

`movl %eax,%edx` `movl %eax,%edx`	`movl %eax,%edx`

would take care of the problem without having to invent any additional logic in the portable part of the code generator.

17.2 Expression Rewriting

Some of the more abstract opportunities for optimization are hard to catch at the instruction level. For instance, the expression `a+b*0` is obviously equal to `a` because $b \times 0 = 0$ for any value of b and $a + 0 = a$ for any value of a. To find such opportunities, a compiler needs to build a suitable *abstract program* notation internally, such as an *abstract syntax tree* (*AST*). The SubC compiler does not employ an AST, so we will modify the toy parser from the chapter on code synthesis to build one.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

The `NODES` variable specifies the maximum number of nodes in a tree. A tree consists of *nodes*, and each node has the following fields:

Field	Description
`Type`	the operation denoted by the node
`Value`	values of numbers, names of variables
`Left`	the index of the left child node
`Right`	the index of the right child node

The `Left` and `Right` slots contain offsets into the individual arrays that together comprise the node pool. The principle employed here is the same as used in the symbol table of the SubC compiler or the `FILE` structure.

`Next` is the next node to allocate. It is initialized with 1, because 0 is used to indicate "no child".

```
#define NODES   1024

int Type [NODES],
    Value[NODES],
    Left [NODES],
    Right[NODES];
int Next = 1;

char    *P;
```

n	1	2	3
Type[n]	+	v	c
Value[n]	0	a	1
Left[n]	2	0	0
Right[n]	3	0	0

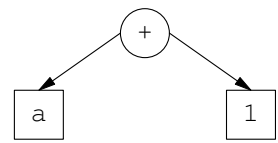


Figure 17.1: An abstract syntax tree (AST)

Figure 17.1 shows a sample syntax tree. The "`+`" node at the top of the tree is represented by slot 1 of the node pool arrays:

```
Type[1]  = '+'
Value[1] = 0
Left[1]  = 2
Right[1] = 3
```

A type of "`v`" is used to represent variables. Variables carry their single-character name in the value field. Type "`c`" indicates a constant with its value in the value field. All other types indicate operations, e.g. "`+`"

indicates the C operator of the same name. Operator nodes have no specific values.

Because the "+" node in the sample tree is stored in slot 1 of the node arrays, we can refer to it simply by this number. `Left[1]` and `Right[1]` contain the slot numbers of the left and right child of the top node in the tree.

A node that has no children (both its left and right child fields are zero) is called an *outer node* or a *leaf*. A node that has at least one child is called an *inner node* or a *vertex*. In the following, vertexes will represent operations and leaves will represent variables and constants. Operations will be indicated by circles, operands by square boxes, just like in figure 17.1.

The `node()` function allocates the next node from the pool and initializes it with the given parameters.

```
int node(int t, int v, int l, int r) {
    if (Next >= NODES) {
        fprintf(stderr, "out of nodes\n");
        exit(EXIT_FAILURE);
    }
    Type[Next] = t;
    Value[Next] = v;
    Left[Next] = l;
    Right[Next] = r;
    return Next++;
}
```

The parser of the toy compiler is almost the same as in the previous version but it also accepts numeric constants, and instead of emitting code directly, it generates a tree structure representing the input program.

```
void skip(void) {
    while (isspace(*P)) P++;
}

int sum(void);

int factor(void) {
    int  n = 0, v;

    skip();
```

```
    if ('(' == *P) {
        P++;
        n = sum();
        P++;
    }
    else if ('-' == *P) {
        P++;
        n = node('-', 0, factor(), 0);
    }
    else if (isdigit(*P)) {
        for (v=0; isdigit(*P); P++)
            v = v*10 + *P-'0';
        n = node('c', v, 0, 0);
    }
    else if (isalpha(*P)) {
        n = node('v', *P++, 0, 0);
    }
    return n;
}
```

Tree generation works as follows: When a parser function is entered, generation of a tree is first delegated to a function handling a higher level of precedence. For example, the below `term()` function calls `factor()`, which will return "constant" leaves, "variable" leaves, or subtrees representing either negation operations or expressions grouped by parentheses.

`term()` will then collect term operators. When it finds one, it recurses once more into `factor()`, which will deliver another tree representing the second argument. The `term()` function then connects the two factors with a vertex that represents the operator matched by it.

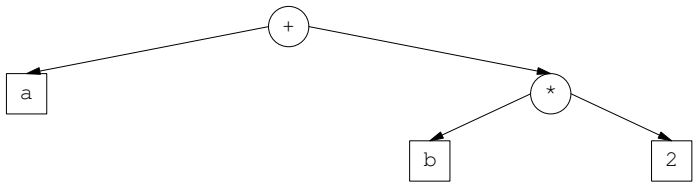
```
int term(void) {
    int  n, n2;

    skip();
    n = factor();
    for (;;) {
        skip();
        switch (*P) {
        case '*': P++;
                  n2 = factor();
                  n = node('*', 0, n, n2);
                  break;
```

```
        case '/': P++;
                  n2 = factor();
                  n = node('/', 0, n, n2);
                  break;
        default:  return n;
        }
    }
}

int sum(void) {
    int  n, n2;

    skip();
    n = term();
    for (;;) {
        skip();
        switch (*P) {
        case '+': P++;
                  n2 = term();
                  n = node('+', 0, n, n2);
                  break;
        case '-': P++;
                  n2 = term();
                  n = node('-', 0, n, n2);
                  break;
        default:  return n;
        }
    }
}

int expr(char *s) {
    P = s;
    return sum();
}
```

The `dump()` function dumps a syntax tree on `stdout` in a somewhat crude format, but it may be suitable for some simple experiments. Given the input expression `a+b*2`, the function will generate the following output (`v(x)` indicates the variable x and `c(n)` the literal number n):

```
+
  v(a)
  *
    v(b)
    c(2)
```

The `dump()` function traverses the tree n in *depth-first* order and emits vertexes before descending into subtrees. This means that it first prints the parent node, then the *complete* left subtree, and then the complete right subtree. However, while you are reading this book, you will be able to enjoy a representation that is more pleasing to the eye:




```
void dump(int n, int k) {
    int  i;

    if (!n) return;
    for (i=0; i<k; i++)
        printf("  ");
    putchar(Type[n]);
    if ('c' == Type[n])
        printf("(%d)", Value[n]);
    else if ('v' == Type[n])
        printf("(%c)", Value[n]);
    putchar('\n');
    dump(Left[n], k+1);
    dump(Right[n], k+1);
}
```

The remainder of this chapter will deal with various algorithms that will help to transform trees in such a way that the code generator can generate better code from them. In case you are not familiar with *recursive tree traversal*, be advised to have a thorough look at the `dump()` function, because it illustrates this key concept of the following sections.

Basically a tree is traversed by visiting its node and then traversing its left child and then its right child. Traversal ends whenever the processing function encounters a null node.

17.2.1 Constant Expression Folding

Constant expression folding is the process that replaces operations performed on constant factors by their results, e.g. it would replace the term `5*7+3` by `38`. The below `fold()` function finds constant expressions in the entire subtree n and replaces them with their final values. Constant expressions cannot always be avoid by the programmer. For example, they may be created in statements like

```
for (i=0; i<MAX-1; i++) f(i);
```

where `MAX` is a constant or a macro expanding to a numeric value. Constant expression folding reduces operations like `MAX-1` in the above example to their final values. It reduces all unary operations with a one constant operand and all binary operations with two constant operands. Furthermore, it recognizes the following patterns and reduces them to their corresponding values:

Pattern	Value

`0 + x`
`1 * x`
`x + 0`
`x - 0`
`x * 1`
`x / 1`
	`x`

`0 * x` `x * 0`	`0`

The `fold()` function evaluates c_1/c_2 only if $c_2 \neq 0$ and otherwise lets the program fail at runtime. In a real-word compiler, we would probably want to report the attempted division by constant zero instead.

```
int fold(int n) {
    int  cl, cr, vl, vr;

    if (!n) return 0;
    Left[n] = fold(Left[n]);
    Right[n] = fold(Right[n]);
    cl = Left[n] && 'c' == Type[Left[n]];
    cr = Right[n] && 'c' == Type[Right[n]];
    if (cl && 0 == Right[n]) {
        if ('-' == Type[n])
            return node('c', -Value[Left[n]], 0, 0);
```

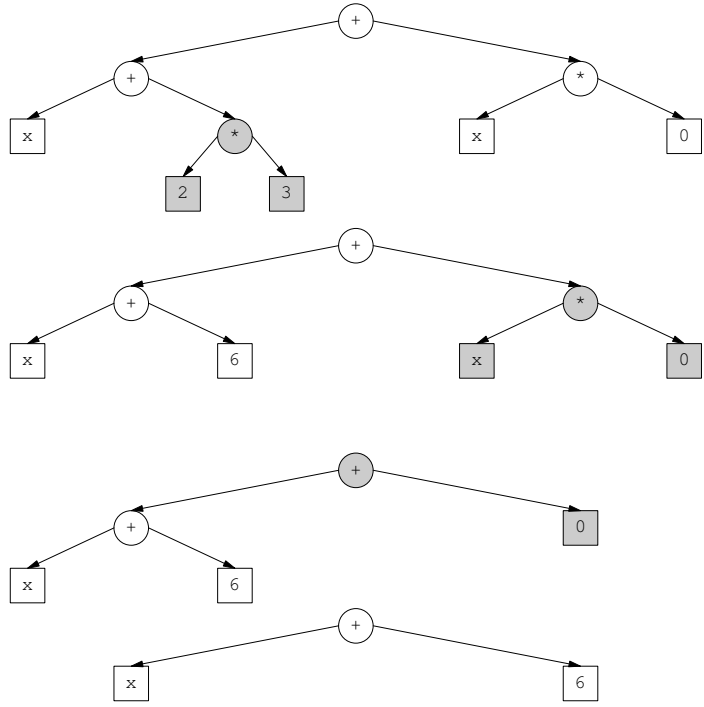


Figure 17.2: Folding the expression `x+2*3+x*0`

```
    }
    vl = Left[n]? Value[Left[n]]: 0;
    vr = Right[n]? Value[Right[n]]: 0;
    if (cl && cr) {
        switch (Type[n]) {
        case '+': return node('c', vl+vr, 0, 0);
        case '-': return node('c', vl-vr, 0, 0);
        case '*': return node('c', vl*vr, 0, 0);
        case '/': if (vr) return node('c', vl/vr, 0, 0);
        }
    }
    if (cl && 0 == vl && '+' == Type[n]) {
        return Right[n];
    }
    if (cl && 1 == vl && '*' == Type[n]) {
        return Right[n];
    }
    if (cr && 0 == vr && ('+' == Type[n] || '-' == Type[n])) {
        return Left[n];
    }
```

```
    if (cr && 1 == vr && ('*' == Type[n] || '/' == Type[n])) {
        return Left[n];
    }
    if ((cr && 0 == vr || cl && 0 == vl) && '*' == Type[n]) {
        return node('c', 0, 0, 0);
    }
    return n;
}
```

Note that because `fold()` *first* folds the left and right operand of an operation and *then* checks the operation represented by the current node itself, it will fold more complex expressions recursively, e.g. it will fold `x+2*3+x*0` to `x+6` as illustrated in figure 17.2 (gray nodes will be folded in the next step).

17.2.2 Strength Reduction

While constant folding eliminates redundant operations and reduces those with a constant result, *strength reduction* attempts to make operations cheaper (i.e. faster and/or taking up less space).

For instance, it could replace the operation `0-x` by `-x`, because negation is faster than subtracting a value from constant zero. Like constant folding, strength reduction works by tree-matching and re-arranging the *AST* of a program, so it works at a rather abstract level. Hence it cannot do some optimizations that the peephole optimizer could do safely, while there are other cases that the peephole optimizer cannot catch due to its limited view. To get the best performance, both approaches should be combined. Of course, this may result in the duplication of some optimizations, like removing sums with a constant zero operand, which can be done in both constant folding and peephole optimization.

Because AST-based optimization is limited to the abstract view of the program, it can only use operators known by the source language. This is why there is no abstract equivalent for peephole optimizations like

```
andl $0xffffffff,%eax
```

This optimization cannot be done at AST level, because it requires to know the *machine word size* of the target machine. If we did this optimization at AST level and later ported the compiler to a machine with a larger machine word size, like a 64-bit machine, the optimization would violate the principle of semantic preservation, because an "and" with `0xffffffff`

is not a null-operation on a 64-bit word.

When implementing an optimizer, it it better to be safe than to be sorry.

The below `rewrite()` function performs the following transformations (and one more, we will get to that immediately).

Pattern	Output
`0 - a`	`-a`
`a + -b`	`a - b`
`a * 2`	`a + a`

There are lots of other transformations that would make sense at this point. We could even implement a real pattern matcher and match more complex operations, thereby enabling reductions like that of `a*b+a*c` to `a*(b+c)`. Basic high school math provides us with plenty of ideas for further improvements.

```
int leaf(int n) {
    return n && ('c' == Type[n] || 'v' == Type[n]);
}

int rewrite(int n) {
    int  t;

    if (!n) return 0;
    Left[n] = rewrite(Left[n]);
    Right[n] = rewrite(Right[n]);
    if ('+' == Type[n] || '*' == Type[n]) {
        if (leaf(Left[n]) && !leaf(Right[n])) {
            t = Left[n];
            Left[n] = Right[n];
            Right[n] = t;
        }
    }
    if ('+' == Type[n] &&
        Right[n] &&
        '-' == Type[Right[n]] &&
        !Right[Right[n]]
    ) {
        return node('-', 0, Left[n], Left[Right[n]]);
    }
    if ('-' == Type[n] &&
        Left[n] &&
        Right[n] &&
```

```
        'c' == Type[Left[n]] &&
        0 == Value[Left[n]]
    ) {
        return node('-', 0, Right[n], 0);
    }
    if ('*' == Type[n] &&
        leaf(Left[n]) &&
        Right[n] &&
        Type[Right[n]] == 'c' &&
        Value[Right[n]] == 2
    ) {
        return node('+', 0, Left[n], Left[n]);
    }
    return n;
}
```

The most interesting transformation of the `rewrite()` function, though, is to convert the ASTs of *commutative operations* to *left-skewed* trees, as displayed in figure 17.3 (page 323).

This transformation is particularly powerful, because it decreases *register pressure* in the code generator, i.e. it avoids *spilling*, especially on machines with few general-purpose registers such as the 386. As we have seen in the chapter on code synthesis, when generating code for expressions whose operator precedence increases from the left to the right, values may have to be spilled to the stack. For instance, the expression `a|b+c*d` may generate code like this:

```
        movl    Ca,%eax
        pushl   %eax
        movl    Cb,%eax
        pushl   %eax
        movl    Cc,%eax
        mull    Cd
        popl    %ecx
        addl    %ecx,%eax
        popl    %ecx
        orl     %ecx,%eax
```

From an abstract point of view, inefficient code is generated because the AST of the program is *skewed* to the right, i.e. larger subtrees accumulate to the right-hand side of the tree. The `rewrite()` function fixes this by swapping the children of each node that

- represents a commutative operation;
- has a leaf on the left side;
- has a vertex on the right side.

The figure on page 323 illustrates this process by rewriting the right-skewed tree of the expression `a+(b+(c+(d+e)))` by performing the following transformations (subexpression to be swapped in the next step are underlined):

```
a + (b + (c + (d + e)))
a + (b + ((d + e) + c))
a + (((d + e) + c) + b)
(((d + e) + c) + b) + a
```

Because the "+" operation is commutative, the order of operands does not matter. When there are different operators in an expression, like in `a|b+c*d`, the same principle applies as long as the operands of each operation stay the same. Hence `rewrite()` can rewrite this expression as follows:

```
a | b + c * d
a | c * d + b
c * d + b | a
```

The AST of the resulting expression would be skewed to the left, which would lead to the following code:

```
        movl    Cc,%eax
        mull    Cd
        addl    Cb,%eax
        orl     Ca,%eax
```

Compare that to the code on the previous page!

17.3 Common Subexpression Elimination

This chapter concludes with the discussion of a slightly more sophisticated technique. At this point we are probably moving into the 20% area of the 80/20 rule. Remember: the more bytes and instructions cycles we want to squeeze out of as program, the harder it gets.

Common subexpression elimination (*CSE*) is quite a high-level optimization, and it is very common in compilers. While the techniques discussed so far attempted to use as few registers as possible, CSE attempts to make better use of existing registers, assuming that a sufficient number of free general-purpose registers is present and that making best use of them will speed the code up. CSE is often tightly coupled to register allocation but—again—this would lead us beyond the scope of this book, so we will take a glance at CSE from a rather isolated point of view.

What CSE basically does is to identify common parts of expressions ("common subexpressions"), extract them, put their value in an extra variable, and then replace the common subexpression with that variable.[2] This way the subexpression has to be computed only once and not in every place where it appears. For example, we could write the expression

```
a*b/c + a*b/c + a*b/c
```

as

```
t=a*b/c; t+t+t
```

where t is an internal temporary location. This is exactly what common subexpression elimination does. Of course a programmer would probably use the latter expression in the first place, but common subexpressions cannot always be avoided without making the code less intelligible, as in

```
a[i*n+j] = -a[i*n+j];
```

or in

```
max(expr1, expr2)
```

which may be a macro application that expands to

```
expr1 > expr2 ? expr1 : expr2
```

After all a good compiler should allow a programmer to express her intentions without having to brood *too much* about the implications for the generated code.

Common subexpression elimination makes rather heavy use of tree traversing functions, so here come a few rather general helper functions.

[2] In a "real-world" optimizer this variable would typically be a register.

The `equal()` function checks whether two trees $n1$ and $n2$ are *equal*. Two trees are equal if they share the same structure ("look the same") and have identical nodes in identical positions. More formally: two trees $n1$ and $n2$ are equal, if

- `Type[n1] = Type[n2]`
- `Value[n1] = Value[n2]`
- `Left[n1]` equals `Left[n2]`
- `Right[n1]` equals `Right[n2]`

where "equals" indicates the recursive application of the procedure.

```
int equal(int n1, int n2) {
    if (!n1) return 0 == n2;
    if (!n2) return 0 == n1;
    if (Type[n1] != Type[n2]) return 0;
    if (Value[n1] != Value[n2]) return 0;
    return equal(Left[n1], Left[n2]) &&
           equal(Right[n1], Right[n2]);
}
```

The `find()` function finds the tree x in the larger tree n and returns 1 if x is contained in n. Otherwise it returns 0. When $x = n$, it also returns 0.

```
int find(int x, int n) {
    if (!n) return 0;
    if (x != n && equal(x, n)) return 1;
    return find(x, Left[n]) || find(x, Right[n]);
}
```

The `size()` function returns the number of nodes in the tree n.

```
int size(int n) {
    if (!n) return 0;
    return 1 + size(Left[n]) + size(Right[n]);
}
```

The `maxduptree()` and `maxdup2()` functions find the largest duplicate subtree contained in the tree n. In `maxdup2()`, t is the full tree and n is

the partial tree currently being traversed. The `find()` function will only find *duplicates*, because it only finds *strict* subtrees (it will not match a tree with itself). The global variables `Sub` holds the largest duplicate subtree found so far and `K` holds the size of `Sub`.

```
int  Sub, K;

void maxdup2(int t, int n) {
    int  k;

    if (!n) return;
    if (t != n && (k = size(n)) > 2 && find(n, t) && k > K) {
        K = k;
        Sub = n;
    }
    maxdup2(t, Left[n]);
    maxdup2(t, Right[n]);
}
```

```
int maxduptree(int n) {
    Sub = 0;
    K = 0;
    maxdup2(n, n);
    return Sub;
}
```

The `replace()` function replaces each occurrence of the subtree x in n with a "variable" node named "`@`" (`v(@)`). "`@`" is the name of the temporary storage location used internally to hold the value of a common subexpression.

```
int replace(int x, int n) {
    if (!n) return 0;
    if (equal(x, n)) return node('v', '@', 0, 0);
    return node(Type[n], Value[n],
            replace(x, Left[n]),
            replace(x, Right[n]));
}
```

Given these helpers CSE is simple. The `cse()` function first identifies the largest duplicate subtree in the expression n. We could employ alterna-

tive methods here, such as searching the largest *and* most-frequently-used subexpression, but we will keep things simple.

If a duplicate subtree S exists, `cse()` generates an assignment of that subtree to the internal symbol @ and then replaces @ for each S in n. The result of the CSE is the new expression with the assignment attached. The `cse()` function uses a node of the type ";" to connect the assignment to the reduced expression in order to indicate that the assignment must be executed *before* the expression.[3]

A sample CSE is shown in figure 17.4 (page 324).

```
int cse(int n) {
    int  csub, t;

    csub = maxduptree(n);
    if (csub) {
        n = replace(csub, n);
        t = node('v', '@', 0, 0);
        csub = node('=', 0, t, csub);
        n = node(';', 0, csub, n);
    }
    return n;
}
```

Optimization is performed by applying the various techniques in a sensible order. Constant folding is performed first because strength-reducing and CSE-ing constant expressions does not make much sense. CSE is performed last because the other phases may result in different choices of common subexpressions. CSE may even be performed another time in order to catch the second-best opportunity as well, but then this optimization is rather expensive and another pass is probably not worth the extra compile time, except when compiling very complex expressions on a machine with lots of general-purpose registers, like numeric code on RISC machines.

```
void opt(char *s) {
    int n;

    n = expr(s);
    n = fold(n);
    n = rewrite(n);
```

[3] In C-speak, the ";" indicates a "sequence point".

```
    n = cse(n);
    dump(n, 0);
}

int main(int argc, char **argv) {
    opt(argc>1? argv[1]: "");
    return EXIT_SUCCESS;
}
```

17.4 Emitting Code from an AST

The only detail left at this point is how to generate code from an AST. A recursive descent parser creates a tree by descending first into the operands of an operation and then connecting the operand nodes with an operator node. When emitting code directly, we descend into the operands and then emit the operator.

This is basically the same process: descend, descend, emit versus descend, descend, connect. So instead of descending into the parser functions, we descend into the tree and emit the operator after visiting the operand subtrees:

```
void generate(int n) {
    if (!n) return;
    generate(Left[n]);
    generate(Right[n]);
    emit(Type[n], Value[n]);
}
```

Using this interface, we can in fact connect the toy optimizer of this chapter with the toy code generator of the previous chapter. Feel free to experiment with the code. You might be surprised about the quality of the code generated by this simple optimizer.

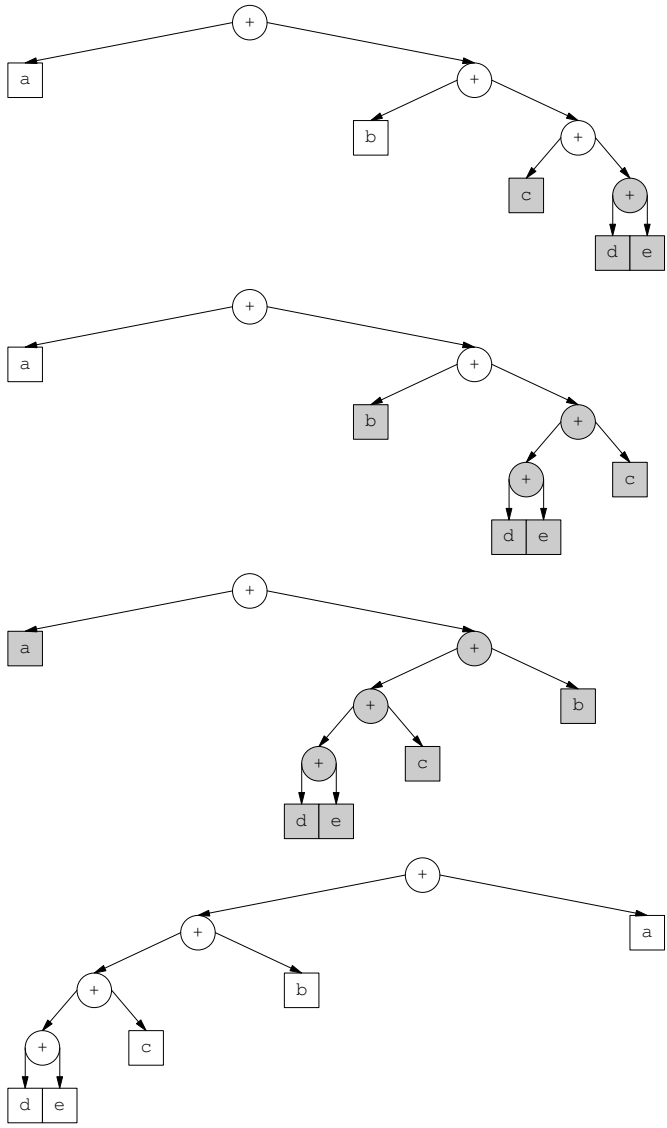


Figure 17.3: Changing the skew of right-associative trees.

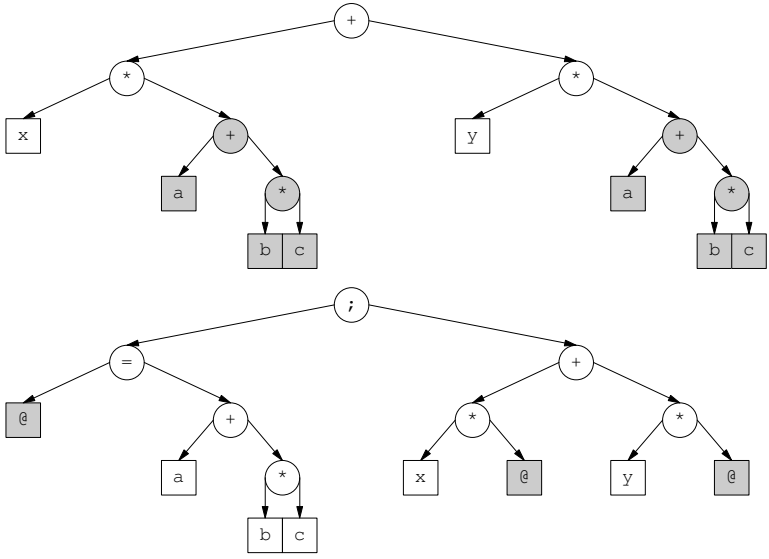


Figure 17.4: Common sub-expression elimination

Part V

Conclusion

Chapter 18

Bootstrapping a Compiler

The creation of a compiler for a new (or even an existing) language involves some *bootstrapping*. The "bootstrapping problem" in this case is to create a compiler that compiles *source language* S to *object language* O where the compiler itself is also written in S, so that it can eventually compile itself. A compiler written in S that translates S to O is called a *self-hosting compiler* because, once bootstrapped, it does not require a third-party *host compiler* any longer.

The problem is that *before* you have a compiler for S there is no way to compile the compiler. For new languages, for which no compiler exists yet, this problem is typically solved by first implementing a simple, non-optimizing compiler for S in a different language. Sometimes the initial compiler is even limited to a subset S^- of S. The subset S^- must be just powerful enough to compile the initial compiler. It may even be implemented as an *interpreter*.

For an existing language like C the initial compiler, S_0, can make use of all the features of the full S language, except when the compiler is a subset (like SubC). When the subset language S^- accepted by the final compiler is a subset of S, then the S_0 (stage-0) compiler should not make use of any sentences not in S^- or it will not be self-hosting.

18.1 Design

The first step in the creation of a compiler is to specify its source language S and object language O. In the case described in this book, this would be $S = SubC$ and $O = 386$ *assembly language*. When a full existing language is implemented, the formal specification of that language can be used to describe S, otherwise a formal specification has to be created from scratch.

In the case of a subset language, an existing specification, such as a formal grammar, can be modified to describe S^-. The object language is defined by the formal specification of the target machine and, in case the compiler will emit input for an assembler, the syntax of the corresponding assembly language.

In the case of SubC the formal grammar of the language, as presented in the "Tour" part, is a subset of the formal C89 grammar as specified in TCPL2. The output language is the AT&T-style assembly language accepted by the wide-spread GNU assembler ("GAS-386").

18.2 Implementation

The implementation phase typically begins on the input side of the compiler, adding definitions and data structures whenever they are needed. For instance, the first part of SubC that was implemented was the *scanner*. This component is straight-forward and easy to test by feeding the source code of the scanner itself to the scanner and watching for error messages. Once the scanner accepts all tokens of S and generates the proper attributes for them, implementation shifts to the parser, preprocessor, and symbol table management.

The *expression parser* is typically written next, because all the other parts of the parser will require to accept expressions, but expressions typically do not involve statements or declarations. SubC uses a separate constant expression parser, which is also covered in this step. The expression parser can be tested by feeding expressions to it. However, the parser cannot be tested thoroughly until it is completely finished, so we are pretty much in the dark during the implementation of the parser.

The *statement parser* and the *declaration parser* come next. Statements and declarations contain each other, so the order of implementation is pretty much arbitrary.[1] Once the entire parser is finished, it can be tested by feeding it the *preprocessed* compiler source code (because we are still lacking a *preprocessor*). The parser does not have to do semantic analysis at this point nor does is have to build a symbol table. The parser can be considered to be stable when it accepts the compiler source code without emitting any error messages. Error handling is considered to be part of the parsing process in this summary, so it should be mostly in place at this point.

The next step is to add either the preprocessor or the *semantic analysis*

[1] The author prefers to do the statement parser first, though, because the statement parser is more tightly coupled to the expression parser.

stage. The preprocessor is rather simple in the case of SubC and adding it will get rid of the separate preprocessing step during compiler tests, so it is a good candidate for the next step. Semantic analysis requires to build the symbol table, so symbol table management also has to be added at this point. This phase of the implementation lasts until the compiler will generate symbol table entries for all symbols encountered in its input and accept its own source code without reporting any errors. All semantic analysis should be covered before proceeding, so the compiler should now be able to detect errors like missing declarations, redefinitions, redeclarations of a different type, out-of-context keywords, etc.

The implementation process then shifts toward the back-end. Depending on the chosen type of back-end, this part of the process may cover the *code generator*, the *target description*, the abstract program *optimizer* and the machine-specific optimizer. Function calls to the back-end interface are inserted into the parser. The interface either builds an abstract program or goes straight to the *emitter* as in the case of SubC. When implementing a multi-platform compiler, particular care must be taken to keep the interface and the code generator abstract and to concentrate all machine-specific code in the target description and the machine-specific optimizer.

When the compiler will contain an optimizer, the implementation of the optimizer should be postponed until the initial implementation is finished and stable (i.e. has passed the triple test; pg. 331). Optimizer stubs can be created before proceeding. A "stub" is a function that will just return its original input and will be supplied with some more sophisticated functionality later in the process.

When the back-end generates an abstract program, that abstract program can be analyzed to make sure that the compiler *front-end* generates proper output. An AST-dumping routine, such as the one introduced in the previous chapter (pg. 311) can be helpful here. A simple back-end, such as SubC's, can be written in parallel with the target description part, and the assembler can be used to test the formal compliance of the output, at least at the syntactic level.

At all stages of the process the SubC compiler read its input from `stdin` and emitted code to `stdout`, which is good enough for initial testing. This changes only when the last part of the compiler, the *compiler controller*, is added. The compiler controller only drives the compilation process, hence the techniques used in it are rather common and not specific to compiler-writing. The complexity of the compiler controller depends on the demands of the users. SubC's controller is just powerful enough to compile simple multi-file projects. Compiler controllers often grow significantly during the

lifetime of a compiler and are quite prone to feeping creaturism.[2]

As soon as the compiler controller is capable of compiling and linking the entire compiler source code, some heavier testing can be performed. The first step is to compile all the source files of the compiler itself and pass them to the assembler in order to fix syntactically erroneous statements generated by the emitter. Once the assembler accepts the complete compiler output, the object files comprising the compiler can be *linked* for a first time.

This step, of course, requires the *runtime environment* to be in place. When writing a full C compiler with compatible calling conventions, the runtime startup module and library of an existing C compiler may be used. For subset compilers, some of the minimal C library implementations may be suitable (such as the SubC library). In the case of SubC, of course, the library had to be written from the scratch due to SubC's weird calling conventions. This is not necessarily a bad thing, though, because this process offers a great opportunity to test the compiler on small, self-contained programs. In fact a lot of bugs in the initial compiler were discovered while implementing the SubC library, so the step of implementing an own runtime library is highly recommended, even if ready-to-run alternatives should exist.

During the implementation of the runtime library it makes sense to write some small test programs, compile them with the not yet completely finished compiler, and test them. The better the compiler performs at this stage, the less spectacularly the next step will fail—and it *will* fail.

18.3 Testing

The next step is to compile the compiler with the S_0 compiler, resulting in an S_1 (stage-1) compiler and then to compile the code again with the S_1 compiler. In the vast majority of cases, this step will end in a crash, core dump, infinite loop, or some similarly spectacular event. This is to be expected. In the author's experience about 50% of the debugging takes place during library implementation and initial creation of the stage-2 compiler. Tools that are helpful in debugging the compiler are rather conventional, like inserting tactical `printf()` statements and examining emitted code fragments to make sure that they meet the specification. The goal is to isolate the part of the compiler that fails, find out why it fails, and fix it.

There is one difference to non-compiler projects, though, because the compiler can fail at two stages: the compiler logic itself can be flawed,

[2] Creeping featurism.

resulting in an immediate error, or the code emitted by the compiler can be flawed, resulting in an error that is visible in the S_1 compiler but not in the S_0 compiler.

Once the S_1 compiler successfully compiles its own code, the result will be an S_2 compiler. The S_2 compiler can in turn be used to create an S_3 compiler, etc. However, all steps beyond stage-2 can be omitted, because when the S_1 compiler is equal to the S_2 compiler, its subsequent stages will generate the same code over and over again, so all further stages will be equal, too. This three-stage test is widely known as the *triple test* and is widely accepted as a first informal proof that the compiler generates (mostly) correct code. The process is outlined in figure 18.1. Depending on the complexity of the compiler and the runtime support library, though, there may be *corner cases* that are not covered by the triple test, so writing an additional test suite it recommended at this point.

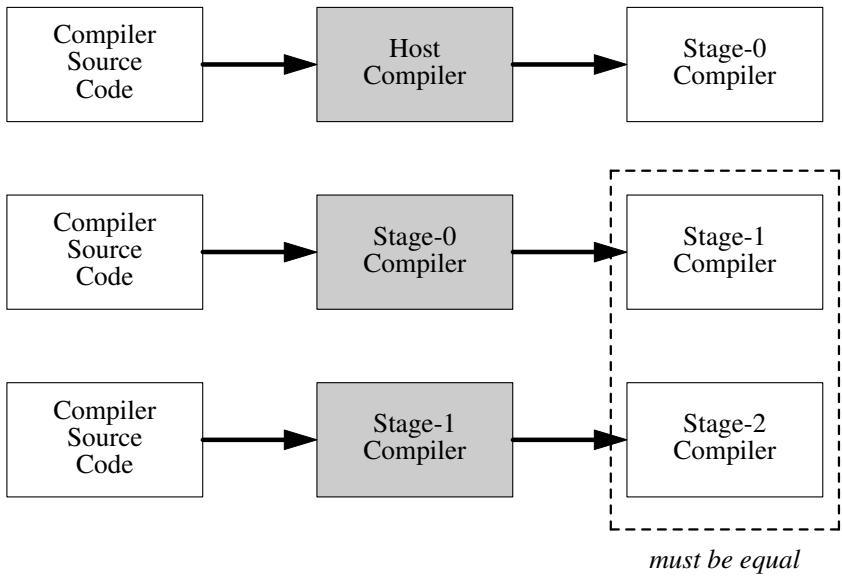


Figure 18.1: The triple test

When the triple test has been passed, it is time to party, because the compiler is now fully self-hosting. The compiler can—and *should*—be used to bootstrap itself from this point on, because subtle bugs are much more likely to manifest in stages one and beyond. Whenever *any part* of the compiler is modified after this stage of the development process, the triple test should be run as a *regression test*.

18.4 Have Some Fun

For your amusement, here are some data about the SubC compiler as presented in this textbook. First, how long does it take to bootstrap the compiler with various host compilers, e.g. gcc 4.2.1 (GNU C), pcc 0.9.9 (Portable C), and scc (SubC itself)?

	Compile S_0		Comp. S_1/S_2	
Command	Time	L/S	Time	L/S
`gcc -static -O2 -o scc-gcc *.c`	5.65s	744	0.47s	8955
`pcc -static -O -o scc-pcc *.c`	1.17s	3597	0.51s	8252
`scc -o scc-scc *.c`	1.06s	3970	1.06s	3970

SCC is a fast compiler, but PCC is definitely in the same ballpark. GCC is much slower (744 *lines per second* (l/s) versus SCC's 3970). Unsurprisingly, PCC and GCC both generate much faster code than SCC. The compilers emitted by them both compile more than 8000 l/s while, of course, the S_2 SCC compiler is exactly as fast as the S_0 compiler generated by itself.[3] An interesting side note: PCC is about as fast as SCC and generates code that is about as fast as GCC's. This is why it is the default C compiler on the author's computer system.

Let us have a look at the code size next:

File	Text	Data	BSS	File Size
scc-gcc	209421	7984	66408	236584
scc-pcc	203181	8112	66132	230076
scc-scc	66640	8044	30224	76576

Again there is not much of a difference between GCC and PCC, but the *text* and *BSS* segment sizes of the SCC output are much smaller than those of the other compilers. This is because SubC uses its own very simple library implementation. The code generated by SubC is in fact larger than that of its competitors, but in small projects this is obviously more than compensated for by using a non-bloated *C standard library*.

Finally, here is a breakdown of the source code sizes of the individual parts of the compiler. It is no surprise that the parser is the largest part in a non-optimizing compiler, but it is quite interesting that the expression parser alone accounts for almost half of the syntax and semantic analysis code.

[3] Yes, this is contradiction in terms, but let's assume it is not.

Component	LOC	%LOC
Scanner	528	12.5
Parser	1651	39.2
Expressions	665	15.8
Statements	330	7.8
Declarations	484	11.5
Emitter	910	21.6
Preprocessor	174	4.1
Symbol table	259	6.2
Controller	279	6.6
Misc.	408	9.7
Total	4209	100.0

The complete runtime support code (headers, library, startup module) weights in another 2721 lines in total, so the complete compiler, ready to compile itself, has a size of 6930 lines of code. This is the amount of C code you will have to write to get a non-optimizing compiler for a reasonable subset of C89—completely from scratch.

But even if you just followed the code throughout this textbook: congratulations! It's quite a tour! Now go hit the bar or the gym or wherever you go to hang out! You have deserved it.

Part VI

Appendix

Appendix A

Where Do We Go from Here?

In case you want to practice your new skills by extending the SubC compiler, here are some suggestions.

A.1 Piece of Cake

Make `&array` a valid expression.

Currently the "`&`" operator is limited to lvalues, but taking the address of an array is a valid exception. This should be easy to fix in a couple of minutes.

Add the `auto` and `register` storage class specifiers.

Both of these specifiers declare the subsequent identifiers as "automatic" (i.e. stack-allocated) objects, hence both keywords are limited to function bodies. The `register` keyword tells the compiler that we intend to use a variable heavily, e.g. as a loop index. Most compilers ignore `register`, and `auto` is redundant in function bodies anyway, so this extension is pretty much a no-brainer.

Add the `#error` and `#pragma` commands.

The `#pragma` preprocessor command is used to modify the behavior of a compiler in an implementation-specific way, so we may simply swallow the rest of the line and be done with it.

The `#error` command is really useful as it can be used to abort compilation with a meaningful error message. It passes the remainder of the command to the error handler and then exits.

Implement a more general `sizeof` operator.

The real `sizeof` operator accepts all kinds of primary expressions and tries to make sense of them. For instance, `sizeof(*p)` or `sizeof(a[1])` would be valid expressions.

All the information you need is present in the `LV` structure, so what are you waiting for?

Add `goto` and labels ("tagged statements").

You will need an additional name space (symbol table) for label names, an additional internal prefix for user-generated labels, and you will have to extend the parser to handle labels and `goto`s:

```
stmt :=
          ...
        | tagged_stmt
        | goto IDENT ;
        | ...

tagged_stmt :=
          IDENT : stmt
```

Semantically viewed a `goto` is just a jump to the corresponding label. Make sure that destination labels exist. Should labels be local to functions or global? Check the specs!

A.2 This May Hurt a Bit

Implement `static` prototypes.

Prototypes are implicitly `extern` in SubC, which means that you cannot express local forward declarations of the form

```
static int foo(int x);
```

so currently all mutually recursive functions have to be public. Add a new "static prototype" storage class to fix this problem. Make sure that you handle the redefinition of `static` prototypes properly. What happens when a `static` prototype is redeclared `extern` or vice versa?

Add K&R-style function definitions.

Old-school (K&R-style) C programs used function definitions of the form

```
int chrpos(s, c)
char *s;
int  c;
{
   /* ... */
}
```

instead of

```
int chrpos(char *s, int c) { /* ... */ }
```

A full C89 compiler still has to accept both of these variants. Note that each declaration between the head and a body is optional. Parameter types default to `int`. "ANSI-style" and old-style declarations may not be mixed, so this would be illegal:

```
int chrpos(char *s, c)  /* WRONG! */
int  c;
{
   /* ... */
}
```

When you add support for K&R-style function definitions properly, another one of SubC's quirks should be gone, too. Either each parameter is decorated with type information in the function signature or none of them is.

What happens when a K&R-style declaration redeclares a symbol? What if a declared symbol does not appear in the parameter list? Note that basically every old-style declaration is a *re*declaration, because the parameter first appeared in the signature.

Support dynamic local initializers.

Real C compilers accept dynamic values as initializers of local variables, for instance:

```
int puts(char *s) {
    int  k = strlen(s);
    /* ... */
}
```

This can be quite handy and make your code more readable. However, you have to generate code for initializing the local variables while their

addresses are still unknown! This can be a bit of a brain-twister. In case you want to spoil the fun, read the footnote.[1]

Add the `unsigned int` *type.*

Adding the `unsigned int` data type will require *a lot of work*, because you will have to figure out all the places where the sign makes a difference. Unsigned division is an obvious candidate. What about asymmetric comparisons? How do you find out whether a negative signed value is less or greater than an unsigned value? What is the result type of an operation involving both a signed and an unsigned operand?

I would like to predict that you will spend a lot of time reading the specs while working on this one!

Oh, and make sure that the combination `unsigned char` is invalid for now. Better yet, implement `unsigned char`s, too. How do signed and unsigned objects of different types interact?

Yes, this is obviously not not a one-nighter!

A.3 Bring 'Em On!

Implement parameterized macros.

This project is the easiest of the hard ones because is it pretty much isolated. The `#define` command has to be expanded to handle macros with parameters. Parameterized macro expansion is done in the scanner. The code does not interact much with the rest of the compiler, and you can output the expanded macros to debug your solution.

Make sure that parameter names occur only once per macro. How do you assign actual arguments to parameter names? How do you expand parameters in macro texts? Make sure that the proper number of arguments is passed to each macro.

Implement the `long` *and* `short` *modifiers.*

The good news is that either `long int` or `short int` int will be the same type as `int`. The bad news is that everything that applies to the `unsigned int` project suggested above also applies to this one, because you are adding one more data type.

[1] Just push each initial value onto the stack as you parse local declarations. Use a dummy value for variables that are not being initialized. Emit code to move the stack pointer when allocating local arrays. Note that you do not have to allocate local storage separately when you choose this approach. BTW, why would zero be a *bad* dummy value for uninitialized variables? What would be a good one? Check the specs!

Only this data type has a different size than all existing types of SubC,[2] so you will have to add new load and store operations to the code generator. Not to forget all the new increment operations, and we know that there are quite a few of them!

Take your time for this project, get a copy of TCPL2 if you do not already have one, and prepare for a lot of reading and testing.

BTW, what about `unsigned long int` and `unsigned short int`?

Add the `struct` *and* `union` *types.*

Compound data types are incredibly useful, so adding them to the language will indeed simplify quite a few things. For instance, it would allow us to rewrite the stdio library function using a proper `FILE` type.

But then the SubC symbol table is not really prepared for hosting compound data types, so you will have to use your imagination to integrate them. Spec-reading is mandatory for this project. The minimum that you will have to add are `struct` declarations, `struct` type specifiers, and the "`.`" and `->` operators. Make sure to properly integrate the new type into the type checker.

Things that you may postpone for now are `union`s (although they pretty much "fall into place" once `struct`s are there) and passing `struct`s by value. You may want to omit `struct` initialization completely.

How do you implement arrays of `struct`s and pointers to `struct`s?

Add an optimizer

All the basics are explained in the chapters on code synthesis and optimization. In principle, you can keep the entire front end and just replace the code generator. However, this is only half of the truth. The synthesizing back-end will at least need some hints regarding when to commit the instruction queue. You will encounter a lot of strange bugs. While much of the work might be rather straight forward, the individual stages of the compiler will soon start to interact in ways that you did not imagine before. The prize, of course, is a compiler that generates much more efficient code!

[2] You will most probably implement `short int`s as 16-bit integers, because doing 64-bit arithmetics on a 32-bit processor will be a bit hard, especially full 64-bit division and multiplication.

Appendix B

(Sub)C Primer

The *C programming language* is a rather small, procedural, block-structured, imperative programming language. A *procedural language* is a language that uses procedures or *functions* as a means of grouping and abstracting code. A *structured language* uses loops and conditional statements (instead of "jumps" or "goto"s) to implement flow of control. It is "block structured" when it uses blocks to organize code. A *block* is a delimited set of statements with some optional data declarations. An *imperative language* tells the computer "what to do" rather than what the values of functions are ("functional paradigm") or how parts of a program are connected logically ("declarative paradigm").

Being an imperative, procedural, block-structured language, C is in the same family of programming languages as ALGOL, Pascal, Modula, Java, C++, or even Python. However, the latter three add some additional features, like dynamic memory management or an object layer. So C is a rather low-level language, and it is sometimes referred to—or even used as—a portable assembler for an abstract machine. A description of such a machine and its compiler is the primary subject of this book.

SubC is a subset of the C programming language that differs from C89 mostly by omission; the details can be found in section 3.1 (pg. 19ff) of this book. The remainder of this appendix will describe SubC.

B.1 Data Objects and Declarations

There are two *primitive types* in SubC, the `int` (integer) and `char` (character).

```
int   i, j, max_number;
char  c, c2;
```

Both of these types are used to store integer numbers or characters. They differ only in size. A `char` is typically byte-sized, and an `int` has the size of a natural machine word, e.g. 4 bytes on the 386, giving it a range from $-2,147,483,648$ to $2,147,483,647$. The SubC `char` is unsigned, so it can store values from 0 to 255.

Any number of objects[1] can be declared in a single *declaration*. All objects are referred to using symbolic names called *identifiers*. An identifier may consist of upper and lower case letters, underscores ("_"), and decimal digits, but the first character of an identifier may not be a digit. Upper and lower case is distinguished. Identifiers beginning with an underscore are reserved for use in the library and for language extensions (such as SubC's `__argc` keyword).

Initial values can be assigned to variables in their declarations, e.g.:

```
int   i = 0;
char  c = 'x';
int   a[] = { 1, 2, 3, 4, 5 };
char  s[]  = "hello";
```

where `'x'` is the character "x", `"hello"` is an array of `char` (a.k.a. a "string") containing the characters "h", "e","l", "l", "o", and NUL. The numbers between the braces form an array of `int` containing the integers from 1 to 5. Note that the lengths of initialized arrays are computed by the compiler, so no size is specified between the brackets.

There is in fact a third type, which is declared in an `enum` statement, but to the compiler `enum`s are just read-only integers. An `enum` declarations looks like this:

```
enum { foo,
       bar = 17,
       baz;
};
```

The first identifier in an `enum` declaration gets the value 0, the next identifier gets the value 1, etc. However, an explicit value can be assigned to an

[1] An "object" here means a *data object*, and not "an instance of a class"; the latter is non-existent in C.

identifier by appending an equal sign and a constant expression. Specifying an explicit value will set the internal `enum` counter to that value, so `baz` will be set to 18 in the above example.

Primitive types (but not `enum`s) can be extended to form arrays of integers or characters or pointers to these types. An array of `int`, for example, would be a sequence of `int`s with a given length. A pointer to `int` would be a new type that refers to—or *points* to—an `int` object or an array of `int` objects. Pointers may also be grouped in arrays and pointers may point to pointers. Figure B.1 illustrates all variations of the `int` type that are valid in SubC.

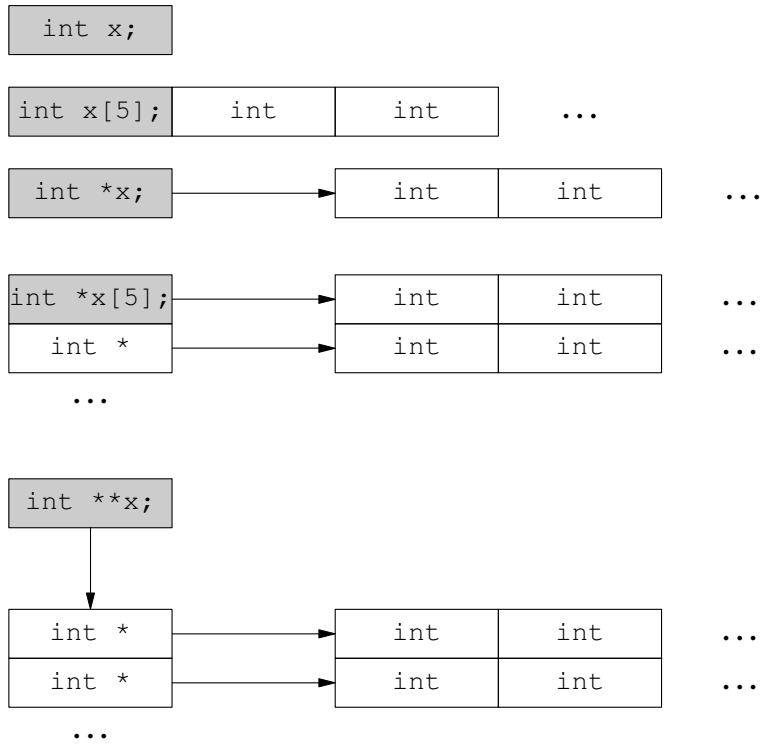


Figure B.1: Arrays and pointers

A pointer is declared by prefixing its identifier with a star and an array is declared by specifying its size in square brackets:

```
int  *pointer_to_int;
int  array_of_int[100];
```

SubC also supports the following combinations (which are just a tiny subset of those allowed by C):

```
int  **p;     /* pointer to pointer */
int  *a[10];  /* array of pointer */
```

(`/* ... */` is a *comment*.)

The difference between a pointer and an array is that an array is a cohesive sequence of objects starting at a specific location, while a pointer is a separate object that holds the address of ("points to") an array.

Different types of objects can be mixed in a single declaration as long as they share the same primitive type:

```
char  c, s[100], **args;
```

The types "pointer to `char`" and "array of `char`" are also called *strings*.

B.1.1 Void Pointers

There is a special pointer type called a "pointer to `void`" or a `void` pointer. Its is declared with the `void` keyword:

```
void   *p;
```

Note that `void` alone is not a valid type (except as a function type), but C pointers can point to `void`. Pointers to `void` pointers are also legal.

A pointer to `void` is special, because it can be assigned any other pointer type without having to use a type cast (see below). This is particularly convenient when declaring functions that should accept pointers to objects of different types (like I/O functions), e.g.:

```
int writeblock(void *block, int length);
```

Such a declaration would allow the subsequent code to pass pointers and arrays of any type to the function.

`void` pointers can be assigned to and compared to pointers of any type. However, they cannot be *dereferenced* without casting them to a more concrete type first, e.g. you would have to write `*(int*)p` instead of `*p` to access the `int` pointed to by the `void` pointer *p*.

B.2 Expressions

The C language has a whopping 46 operators at 15 different levels of precedence. They are summarized in the tables B.2 and B.3.

Prec.	Op	Description
15	`f(args)`	call function f with optional arguments
	`(expr)`	subexpression grouping
	`a[i]`	fetch the i'th element of the array a
14	`!x`	logical NOT; map $x = 0 \rightarrow 1$ and $x \neq 0 \rightarrow 0$
	`~x`	invert bits of x
	`x++`	increment x after accessing it
	`x--`	decrement x after accessing it
	`++x`	increment x before accessing it
	`--x`	decrement x before accessing it
	`+x`	positive sign (null operation)
	`-x`	negate x
	`*p`	access object pointed to by p
	`&x`	compute address of x
	`(type) x`	pretend that x has type *type*
	`sizeof(x)`	compute size of x
13	`x * y`	product of x and y
	`x / y`	integer quotient of x and y
	`x % y`	remainder of `x/y`
12	`x + y`	sum of x and y
	`x - y`	difference between x and y
11	`x << y`	x shifted to the left by y bits
	`x >> y`	x shifted to the right by y bits

Figure B.2: Operators

Notes

The `[]` operator can also be applied to pointers and the "`*`" operator to arrays, e.g. `p[5]` returns the fifth element of the array pointed to by p and `*a` returns the first object of the array a.

The "`&`" operator can be combined with `[]` to compute the addresses of array elements, e.g. `&a[7]` will deliver the address of the seventh element of a.

`(Type)x` is a so-called *type cast*. For example,

```
(int *) x;
```

will assume that x is a pointer to `int`, no matter what the type of x actually is. Caveat utilitor! Here is a summary of all valid type casts in SubC:

Prec.	Op	Description
10	`x < y`	test whether x is less than y
	`x > y`	test whether x is greater than y
	`x <= y`	test whether x is less than/equal to y
	`x >= y`	test whether x is greater than/equal to y
9	`x == y`	test whether x is equal to y
	`x != y`	test whether x is not equal to y
8	`x & y`	bitwise product (AND) of x and y
7	`x ^ y`	bitwise negative equivalence (XOR) of x and y
6	`x | y`	bitwise sum (OR) of x and y
5	`x && y`	short-circuit logical AND of x and y
4	`x || y`	short-circuit logical OR of x and y
3	`x ? y : z`	conditional operator (if x then y else z)
2	`x = y`	assign y to x
	`x *= y`	short for `x = x * y`
	`x /= y`	short for `x = x / y`
	`x %= y`	short for `x = x % y`
	`x += y`	short for `x = x + y`
	`x -= y`	short for `x = x - y`
	`x <<= y`	short for `x = x << y`
	`x >>= y`	short for `x = x >> y`
	`x &= y`	short for `x = x & y`
	`x ^= y`	short for `x = x ^ y`
	`x |= y`	short for `x = x | y`
1	`x , y`	compute x and y, return y

Figure B.3: Operators, continued

```
(int)  (char)  (int *)  (char *)  (int **)  (char **)
(int (*)())
```

(The last one will be explained in the section on functions, pg. 353.)

The comparison operators (`<`, `<=`, `>=`, `>`, `==`, `!=`) will deliver 1 if the respective condition applies and 0 otherwise.

The *short-circuit* expression `a && b` computes b only if a has a non-zero value. The expression `a || b` computes b only if a is zero. `a ? b : c` computes b only if a is non-zero, and c only if a is zero.

The *order of evaluation* of `a,b` is unspecified, which means that a may be computed before b or vice versa.

B.2.1 Pointer Arithmetics

While most arithmetic operators can only be applied to the types `int` and `char`, but not to pointers and arrays, the following operators pose an exception:

```
++   --   +   -   +=   -=
```

When these operators are applied to a pointer, they can be used to *move* pointers to different elements of the same array. E.g.:

```
p += 5      /* move to the 5th element of p */
p--         /* move to the ``previous'' element of p */
```

where the "previous" element is the element with the highest address below p. This is illustrated in figure B.4.

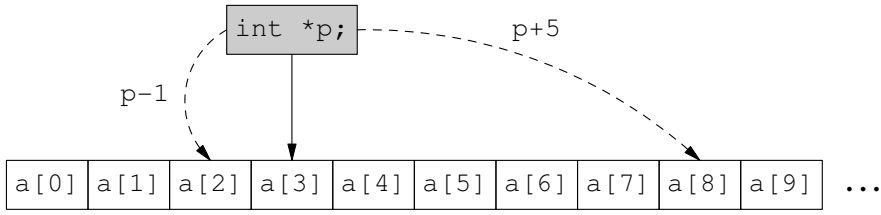


Figure B.4: Pointer arithmetics

Because each element in an array of `int` has a size that is larger than one byte (4 bytes on the 386 processor), adding 1 to an `int` pointer will in fact add 4 to its value, so that it will point to the next `int` in an array. Subtracting a value n from an `int` pointer will subtract n times the size of an `int`. The expressions `*(p+n)` and `p[n]` are equivalent.

The "`-`" operator can be used to subtract one pointer from another, giving the number of elements between the two pointers, e.g. the expression `&a[7] - &a[5]` will always give 2, no matter which (pointer/array) type a has.

The following combinations of integers and pointers are valid in pointer arithmetic operations (where p is a pointer and n an integer):

Expr.	Result type	Description

`p + n`
`n + p`
`p - n`
	p	address of the n'th element of p

`p0 - p1`	n	number of elements between $p0$ and $p1$

B.3 Statements

`{}`
`;`

These statements are null-operations.

`{ statement; ... }`

Curly braces are used to delimit a *compound statement*, i.e. a list of statements that will be executed in the given order. Compound statements can be used to include more than a single statement in the body of a loop or conditional statement, e.g.:

```
for (i=0; i<10; i++) {
    do_something();
    do_something_else();
}
```

```
if (i<17) {
   do_something();
   do_something_else();
}
```

`if (expr) stmt`
`if (expr) stmt1 else stmt2`

The `if` statement executes the statement *stmt* only if the expression *expr* has a "true" (non-zero) value. When an `else` clause is also specified, it executes *stmt1* if *expr* is true and *stmt2* if *expr* is false (zero).

`switch (expr) { statement; ... }`
`case constexpr:`
`default:`

A `switch` statement computes the given *expr* and compares it to each *constexpr* (constant expression) specified in a `case` of the `switch`. When a match is found, all *statement*s in the switch that follow that `case` will be executed (except when a `break` is found; see below). When no *constexpr* matches, the statements in the `default` section of the switch statement will be executed. When no *constexpr* matches and no default case exists, the `switch` statement has no effect. Here is an example:

```
switch (c) {
case '1': case '3': case '5':
case '7': case '9':
    printf("odd\n");
    break;
```

```
case '0': case '2': case '4':
case '6': case '8':
    printf("even\n");
    break;
default:
    printf("not a digit\n");
    break;
}
```

`break`

A `break` statement transfers control to the statement immediately following a `switch` or loop. It is invalid outside of `switch` or loop contexts.

`continue`

A `continue` statement transfers control in such a way that the next iteration of a loop is entered immediately. See the individual loop statements for details. The `continue` statement is invalid outside of loop contexts.

`do statement while(expr);`

The `do` loop repeats the given *statement* while *expr* remains true. The *statement* is executed at least once. `continue` will jump to the test of the loop.

`for (init; test; update) statement`

A `for` loop works as follows:

- the *init* expression is computed;
- the *test* expression is computed; when is is not true, the loop terminates;
- the *statement* is executed;
- the *update* expression is computed;
- the *test* expression is computed again, etc.

A `continue` statement in `for` jumps to the *update* part. The following `for` loop counts from 1 to 10:

```
for (i=1; i<=10; i++) ;
```

The *init*, *test*, and *update* parts are all optional (but the semicolons between them are not). The statement `for(;;);` loops indefinitely.

```
while (expr) statement
```

The `while` loop executes the given *statement* as long as *expr* is true. When *expr* is false initially, the *statement* is never executed. `continue` jumps to the test of the loop.

```
return;
```

```
return expression;
```

The `return` statement returns control to the caller of the current function. When an *expression* is specified, its value is returned to the caller.

B.4 Functions

A C *function* definition consists of a *head* specifying the name, *type*, and *signature* of the function and a *body* holding the local data objects and statements of the function:

```
int chrpos(char *s, int c) {
    char  *p;

    p = strchr(s, c);
    return p? p-s: -1;
}
```

In this example, `int` is the type of the function, *chrpos* is its name, and `(char*s, int c)` is its signature. The rest of the definition forms the body of the function. The variable *p* is *local* to the function. It exists only as long as the function executes. The signature contains the types and names of the function *parameters*. It is used to type-check calls to the function. Parameters are local variables whose values will be set by the caller of the function.

When a function is called, the parameters are paired up with actual arguments by position, i.e. the first parameter gets the value of the first argument, etc. The return value of a function can be used as a factor in an expression.

A function of the type `void` does not return any value and hence its call is not a valid factor. Void functions do not return any values in `return` statements.

When a function is declared with the `static` keyword in front of its type, then the function will not be exported, i.e. external modules cannot call it. C functions are "public" (non-static) by default.

When a local variable in a function is declared `static`, then its value will persist between calls. The following function returns a series of increasing values when called in succession:

```
int next(void) {
    static int v = 0;

    return v++;
}
```

A function not accepting any parameters has the signature `(void)`.

Functions may be called *indirectly* by storing their addresses in a *function pointer* and then calling that pointer instead of the function, e.g.:

```
#include <stdio.h>

main() {
    int  (*fp)();

    fp = puts;
    fp("foo!");
}
```

SubC function pointers are limited to functions of the type `int` and they may not have signatures, so no argument type checking can be performed when calling a function through a pointer.

B.5 Prototypes and External Declarations

When a function is called before its declaration, a signature for the function will be created by the compiler and the parameter types will be inferred from the argument types. This does not always work, though, so it is good practice to specify a function *prototype* prior to the first call to a function. A function prototype consists of a function head followed by a semicolon in the place of a function body, e.g.:

```
int writeblock(void *block, int length);
```

Prototypes are also used to specify the types of functions in external modules.

Data objects in external modules can be accessed after declaring them with the `extern` keyword, e.g.:

```
extern char  **env;
extern int   errno;
```

All data objects are public by default, but can be made local to a module by declaring them `static`:

```
static int  only_used_here;
```

B.6 Preprocessor

The C language is actually two languages in one: the C language itself and the language of the C *preprocessor* (also called *CPP*).

Preprocessor statements are identified by a leading "`#`" character and they are executed *before* the compiler sees the C source code in which they are embedded. Each CPP statement must be contained in a separate line.

SubC supports the following CPP commands:

`#define name text`

The `#define` statement creates a *macro* named *name* and assigns the given *text* to it as a value. *Text* may be anything, a number, a string, or even a series of tokens:

```
#define HELL      for (;;) {
#define GOTO_HELL }
```

When the name of a macro is found in the input stream, it is replaced by the text of its value, e.g.:

```
main() {
    HELL
        printf("let me out!\n");
    GOTO_HELL
}
```

would expand to

```
main() {
    for (;;) {
        printf("let me out!\n");
    }
}
```

(The sequence `\n` in the string denotes a newline character.)

`#undef name`

The `#undef` statement removes the definition of the macro with the given *name*.

`#include <file>`
`#include "file"`

The `#include` statement includes the content of the specified file in the current input program. When angle brackets are used instead of quotes around the file name, the file will be assumed to be a *header file* that belongs to the C language implementation. These files will be loaded from a special location.

`#ifdef name`
`#ifndef name`
`#else`
`#endif`

The `#ifdef` and `#ifndef` commands are used to compile regions of code *conditionally*. Only when the macro *name* has been defined by `#define`, the code between `#ifdef name` and `#endif` will be compiled. `#ifndef` compiles subsequent code only if the given *name* is *not* defined. The `#else` command reverses the current state of compilation, `#endif` terminates the conditional region. The following code would print "`:-)`" at run time:

```
#define FOO
#ifdef FOO
    printf(":-)\n");
#else
    printf(":-(\n");
#endif
```

Both `#ifdef` and `#ifndef` may be nested safely.

B.7 Library Functions

Most of the more abstract functionality of the C language is contained in its *standard library*. Constants, data types, and prototypes of standard library functions are contained in the corresponding *standard headers*, which are typically included at the beginning of a C program:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("Hello, World!\n");
    exit(EXIT_SUCCESS);
}
```

Here the `stdio.h` header provides the prototype for the `printf()` function and `stdlib.h` provides the prototype of `exit()` as well as the `EXIT_SUCCESS` macro.

See the sections on the C library (pg. 233ff) for brief descriptions of the library functions used in this textbook.

Appendix C

386 Assembly Primer

The *386 processor* (also known as 80386, 386DX, or IA-32) is a 32-bit processor that was originally manufactured by the Intel corporation in the mid-1980's. It extended the 8086, 80286, and the (lesser known) 80186[1] from a machine word size of 16 to 32 bits, added a flat (non-segmented) 32-bit addressing model, and made the instruction set more orthogonal, for example by allowing indirection through registers other than the index registers.

The 386 architecture is the common basis for all modern *x86* processors up to and including the x86-64, which is still able to execute 386 code.

C.1 Registers

The register set of the 386 contains the original 8086 register set and extends it to 32 bits. For instance, the 8086 had a 16-bit *accumulator* called *AX* and the 386 has a 32-bit accumulator called *EAX* (extended *AX*), which contains *AX* in its least significant 16 bits. So 386 programs can address both the 16-bit accumulator as *AX* and the extended accumulator as *EAX*. All other registers are mapped to 32-bit registers in the same way, e.g. *SI* becomes *ESI*, etc. The part of the register set that will be used in the textbook can be found in figure C.1. Note that *AT&T notation* will be used throughout this textbook to name registers and instructions, so *EAX* will be named *%eax*, *CL* will be *%cl*, etc.

The registers on the left-hand side of the figure can be addressed as 32-bit registers and 16-bit registers, and the 16-bit parts of them can be addressed as two separate 8-bit registers, one holding the "low byte" (e.g.

[1] Basically an 80286 without "protected mode" instructions.

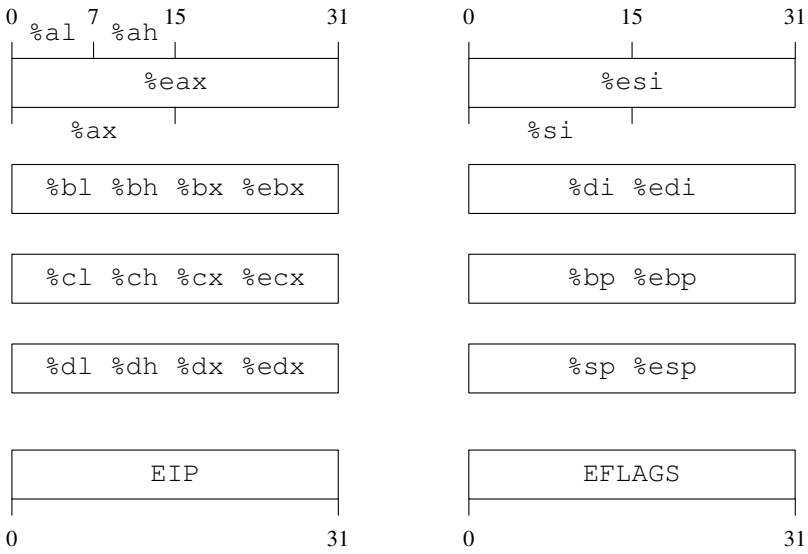


Figure C.1: The registers of the 386 processor

%al) and one holding the "high byte" (e.g. *%ah*) of the 16-bit register. The registers on the left were originally called the "accumulator" (*%eax*), the "base register" (*%ebx*), the "counter register" (*%ecx*) and the "double word register" (*%edx*). The *DX* register formed a 32-bit register in combination with *AX* in the 8086. In the 386 all of these registers are pretty much general purpose registers in the sense that most operations can now be performed on them, but some are still tied to special purposes:

- *%eax* holds the low words of 64-bit multiplication results and 64-bit dividends;
- *%cl* and *%ecx* hold the counter in bit-shift operations with a variable shift count and the iteration counters in *loop* instructions;
- *%edx* holds the high words of 64-bit multiplication results and 64-bit dividends.

The *%ebx* register was originally used as a base register in indirect addressing, but it is no longer tied to that purpose, because any general-purpose register can now be used as a base register. Hence *%ebx* is now free for general use.

The *EIP* ("extended instruction pointer") register holds the address of the next instruction to execute; it cannot be accessed directly, but is modified implicitly by "call", "jump", and "return" instructions (pg. 365f).

The *EFLAGS* ("extended flags") register holds (among other things)

some bits indicating properties of the results of arithmetic instructions, such as "zero", "negative", or "carry". It also holds the "direction" bit of the load-and-increment instructions. See your favorite 386 documentation for a detailed discussion of the flags register.

C.2 Assembler Syntax

The general format of an AT&T 386 assembly language instruction is

```
label:  instr   source,dest
```

where *label* is a symbolic name identifying an instruction or a data object, *instr* is an instruction, *source* is the source operand of the instruction, and *dest* is the destination operand. AT&T syntax places the source operand to the left of the destination operand, so `addl %ecx,%eax` would add the content of *%ecx* to *%eax* and not vice versa.

All parts except for the instruction itself are optional (if the instruction does not have any operands). There are zero-operand instructions, single-operand instructions (with either a source or destination operand), and two-operand instructions.

C.2.1 Assembler Directives

386 assembly language also defines some *pseudo instructions* (a.k.a. *assembler directives*), which may take different sets of operands than those indicated above. The pseudo instructions used in this textbook are the following:

`.text` — Text segment

Subsequent instructions and data definitions will be emitted to the *text segment*, i.e. the "executable" part of the program.

`.data` — Data segment

Subsequent instructions and data definitions will be emitted to the *data segment*, i.e. the "storage" part of the program.

`.globl name` — Global symbol

The given symbol name will be made "global", so that external modules can refer to it.

`.byte value` — Define byte

Emits a byte with the given value. The value may be specified as a small integer number or a C-style character constant.

`.long value` — Define 32-bit machine word

Emits a machine word with the given value. The value is specified as an integer number.

`.lcomm name,size` — Define local "common" block

"Common" blocks are blocks that are shared among modules, so a "local common" block is in fact a misnomer. An "lcomm" block is just a block of local storage with a length of *size* bytes that is identified by the symbol *name*. It is local by definition, but can be made global with a *.globl* directive. Note that the name of an "lcomm" block is specified as an operand and not as a label.

C.2.2 Sample Program

Here is a sample 386 assembly program that will compile on FreeBSD-386 systems (and, as the author suspects, also on some Linux-386 systems):

```
        .text
        .globl  _start
_start: pushl   $13         ; write(1,hello,13)
        pushl   $hello
        pushl   $1
        call    write
        addl    $12,%esp    ; remove arguments
        pushl   $0          ; _exit(0)
        call    _exit
        .data
hello:  .byte   'H',  'e',  'l',  'l',  'o',  ','
        .byte   ' ',  'w',  'o',  'r',  'l',  'd'
        .byte   10
```

The program pushes the number 1, the address of the bytes marked by the label *hello*, and the number 13 onto the stack (in reverse order, because `write` expects them that way), calls `write`, removes the arguments by incrementing the stack pointer, and then exits via `_exit`.

Note that you can call `write` and `_exit` just like that because these are *system calls* that do not require any prior initialization. If you would call `puts()` or `printf()` instead, the program would very probably crash due

to lack of initialization of the stdio part of the C library. Code generated by SubC can do this, because it is linked against code that initializes the runtime library.

The global `_start` symbol marks the default entry point for the ELF linker. It is optional, but its absence could cause a warning at link time.

You can compile, link, and run the program with the following commands (after saving it to the file `hello.s`):

```
as -o hello.o hello.s
ld hello.o /usr/lib/libc.a
./a.out
```

C.3 Addressing Modes

C.3.1 Register

Instructions involving one or multiple registers use the *register addressing mode*. Examples:

```
subl    %edx,%eax   /* subtract %edx from %eax */
incl    %edi        /* increment %edi */
```

C.3.2 Immediate

In *immediate addressing mode* the value of one operand is contained in the instruction. For instance:

```
subl    $32,%ebp    /* subtract 32 from %ebp */
pushl   $0xfff      /* push 4095 to the stack */
movl    $L,%eax     /* load the address of L into %eax */
```

Immediate operands are prefixed with a "`$`" sign. All C-style numeric representations may be used. When the "`$`" is followed by a label, the value of the immediate operand will be the address marked by that label. "Destination" operands may not be immediate, e.g. `incl $1` is illegal.

C.3.3 Memory

The *memory addressing mode* uses an operand that is stored in memory. Symbolic names are used to refer to memory locations. E.g.:

```
decl    foo         /* decrement 'foo' */
addl    $15,bar     /* add 15 to 'bar' */
```

C.3.4 Indirect

The 386 CPU has quite a few *indirect addressing mode*s. An indirect operand is an operand whose address is stored in a register. Access is performed indirectly through the register. The simplest form of indirect access uses just one register. The instruction

```
movl    (%eax),%ebx
```

would load the *%ebx* register with the content of the memory location whose address is stored in the *%eax* register. In C we may envision this as `ebx = *eax` or `ebx = eax[0]`. However, the 386 also has a few more complex indirect addressing modes:

386 Example	C Equivalent
`movl 16(%eax),x`	`x = *(eax + 16)`
`movl 20(%eax,%ebx),x`	`x = *(eax + ebx + 20)`
`movl 12(%eax,%ebx,4),x`	`x = *(eax + ebx * 4 + 12)`

The full format of an indirect operand is

```
displacement(register, base-register, scaling-factor)
```

where only the "register" part is mandatory. The "displacement" and "base-register" parts default to 0, and the "scaling-factor" defaults to 1. The "register" and "base-register" part may be any general-purpose register, but if one of them is either *%esp* or *%ebp*, then the other must not be either of them, e.g.:

```
movl    (%eax,%ebp),x    /* OK */
movl    (%esp,%ebx),x    /* OK */
movl    (%ebp,%esp),x    /* WRONG! */
```

When a "scaling factor" is specified, it must be either 1, 2, 4, or 8. The address referred to by an indirect operand is computed as follows:

register + *base-register* ∗ *scaling-factor* + *displacement*

C.4 Instruction Summary

Instructions in this summary accept the following argument types:

a	address (label)
d	destination (register/memory/immediate/indirect)
n	immediate
r	register
s	source (register/memory/immediate/indirect)
%cl	the *%cl* register

Note: when either *s* or *d* is a memory or indirect operand in a two-operand instruction, then the other operand *must* be a register.

When the operand size (8-bit byte, 16-bit word, 32-bit longword) cannot be inferred from a register, the type must be appended to the instruction as a single-character suffix, e.g.:

```
incb    x       /* increment byte */
negw    (%eax)  /* negate word */
lodsl           /* load long word and increment by 4 */
```

C.4.1 Move Instructions

`cld` – Clear direction bit

The "direction" bit specifies whether the *lods* instructions will decrement (direction bit set) or increment (direction bit clear) the *%esi* register after loading an operand.

`lea s,r` – Load effective address of *s* into *r*

This instruction loads the address rather than the content of a memory or indirect operand (*s*) to the register *r*. It computes the address of *s* in the same way as *mov*, but does not perform the final indirection.

`lods` – Load string / load and increment

The *lods* instruction loads the operand pointed to by the *%esi* register into the accumulator and then increments *%esi*. This instruction must *always* have a type suffix:

`lodsb`	load byte into *%al*, increment *%esi* by 1
`lodsw`	load word into *%ax*, increment *%esi* by 2
`lodsl`	load longword into *%eax*, increment *%esi* by 4

See also: *cli*, above.

`mov s,d` – Move *s* to *d*

`pop d` – Pop top of stack into *d*

Load `(%esp)` into *d*, then add 4 to *%esp*.

`push s` – Push *s* onto stack

Subtract 4 from *%esp*, then move *s* to `(%esp)`.

`xchg s,d` – Exchange *s* and *d*

Neither *s* nor *d* may be immediate.

C.4.2 Arithmetic Instructions

`add s,d` – Add *s* to *d*

`and s,d` – Bitwise "and" *s* to *d*

`cdq` – Convert double (long) word to quad word

Sign-extend the 32-bit value in *%eax* to a 64-bit value ("quad word") in *%edx*/*%eax*.

`dec d` – Decrement *d*

`div s` – Divide (unsigned) *%edx*/*%eax* by *s*

This is a 64-bit/32-bit division. The high-word of the dividend is stored in *%edx*. To perform a 32-bit division, set *%edx* to zero.

`idiv s` – Divide (signed) *%edx*/*%eax* by *s*

This is a 64-bit/32-bit division. The high-word of the dividend is stored in *%edx*. To perform a 32-bit division, sign-extend *%eax* using the *cdq* instruction.

`imul s` – Multiply *%eax* by *s*

The 64-bit result will be stored in *%edx*/*%eax*.

`inc d` – Increment *d*

`neg d` – Negate *d*

`not d` – Invert *d* (bitwise "not")

`or s,d` – Bitwise "or" *s* to *d*

`sar n,d` – Arithmetically shift d right by n bits

`sar %cl,d` – Arithmetically shift d right by $\%cl$ bits

An "arithmetic" right shift will preserve the sign bit of the operand, i.e. leave the most significant bit as it is.

`sbb s,d` – Subtract with borrow s from d

Set $d = d - s - b$ where b is the "borrow" (a.k.a. "carry") flag. The carry flag is set when an overflow occurs in an add/subtract operation.

`shl n,d` – Shift d left by n bits

`shl %cl,d` – Shift d left by $\%cl$ bits

`shr n,d` – Shift d right by n bits

`shr %cl,d` – Shift d right by $\%cl$ bits

Unlike in the *sar* instruction the most significant bit is set to zero.

`sub s,d` – Subtract s from d

`xor s,d` – Bitwise "xor" s to d

C.4.3 Branch/Call Instructions

`call a` – Call subprogram at a

`call *r` – Call subprogram with address in r

A "call" pushes the *EIP* register to the stack and then loads it with a (or r).

`cmp s,d` – Compare s to d

The *cmp* instruction sets various flags in the *EFLAGS* register. These flags are tested by the conditional jump instructions below.

`je a` – Jump on equal to a

`jg a` – Jump on greater to a

`jge a` – Jump on greater/equal to a

`jl a` – Jump on less to a

`jle a` – Jump on less/equal to a

`jne a` – Jump on not-equal to a

`jnz a` – Jump on not-zero to a

`jz a` – Jump on zero to a

These instructions typically follow a *cmp* instruction. For instance, the sequence

```
cmpl    $5,%eax
jl      L
```

will jump to L, if *%eax* is less than 5.[2]

Note that the conditional branch instruction cannot cover distances that do not fit in a *byte* (±127 bytes).

Also note that `je` is exactly the same as `jz` (the latter indicating that the difference between two operands is 0) and `jne` is the same as `jnz`.

The flags tested by the conditional jump instructions are also set by other instructions. For example, `or s,d` will set the "zero" flag when both *s* and *d* are zero; hence

```
        orl     %eax,%eax
        jz      L
```

will branch to *L* when *%eax* is zero.

`jmp a` – Jump to *a*

`loop a` – Loop to *a*

This instruction is used to implement counting loops. It decrements the *%ecx* ("counter") register and then jumps to *a* when *%ecx* is not zero.

`ret` – Return from subprogram.

The *ret* instruction simply pops *EIP* off the stack.

[2]Yes, the *source,destination* order of operands is confusing as hell in *cmp* instructions!

List of Figures

Index