INTRODUCTION TO COMPILER CONSTRUCTION WITH UNIX

Axel T. Schreiner
H. George Friedman, Jr.

Introduction to Compiler Construction with UNIX*

Axel T. Schreiner
Sektion Informatik
University of Ulm, West Germany

H. George Friedman, Jr.
Department of Computer Science
University of Illinois at Urbana-Champaign

Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

Library of Congress Catalog Card Number: 84-631)39

Editorial/production supervision:
Sophia Papanikolaou/Barbara Palumbo
Cover design: Lundgren Graphics
Manufacturing buyer: Gordon Osbourne

Prentice-Hall Software Series, Brian W. Kernighan, advisor.

Printed in the United States of America

10 9 8

ISBN 0-13-474396-2 01

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

To Carol and Claudia,
who put up with us.

Contents

Introduction

Better user interfaces, especially to the many micro computers which are becoming so popular, often require recognition and processing of rather elaborate command languages. Language recognition thus has many applications. However, it can be downright unpleasant if done *ad-hack*. Building a compiler illustrates one application of language recognition. It also illustrates how to design and implement a large program with tools and succ essive extensions of a basic design. Understanding a simple compiler is interesting in its own right: it helps one to master and better utilize programming systems in general.

This book is a. case study: how to create a compiler using generators such as *yacc* (LALR(1) parser generator) and *lex* (regular expression based lexical analyzer generator), two very powerful yet reasonably easy to use tools available under the UNIX' system.

A very simple subset of C, called *sampleC*, is defined and used as an example for compiler development. The resulting implementation of *sampleC* is not intended as an end in itself, and is therefore allowed to produce less than optimal object code. Suggestions for improvements to the code and extensions to the language are given as problems for the reader in several chapters.

The text largely avoids theoretical details, such as detailed discussion of grammars, or explanations of the internal workings of the generators, but it does suggest readings. It explains at least the simpler aspects of using the generators.

As a result, on one level we present a tutorial on how to use the generators to get a simple, easily modifiable implementation done quickly and reliably. On another level, the reader learns practical details about the components of a compiler and the customary interfaces between them.

As such, the text is intended both as a short exposition preceding detailed algorithm studies in compiler construction, and as a description of how to productively employ the generators described. It is not intended to be a comprehensive treatment of the subject of compiler design. Neither does it discuss all aspects of using the generators; once the text has been read, the original descriptions of the generators [Joh78] and [Les78b] are accessible, and they are intended to be used as references to accompany the text. Since a compiler is a large program, the text demonstrates how such a program can be structured and designed with attention to debugging, extension, and maintenance issues.

The reader is expected to have a working knowledge of an editor, of the host machine's file system manipulations, and — of course — of C, in which all the examples are written. Knowledge of other languages, such as Pascal, Fortran, or Basic, would also be useful to the reader. An understanding of block structured languages such as C and their scope rules and of data structures such as stacks, is important beginning with chapter 5. Some experience with a. pattern matching editor is assumed in chapter 2.

t UNIX is a trademark of Bell Laboratories.

A few words about terminology. We make a careful distinction between the *declaration* of a variable, function, label, or other program element, and the *definition* of that element. A declaration is merely a statement that a. program element exists and has certain characteristics, such as the type of a variable, the types of the parameters and return value of a function, etc. A definition gives substance to a declared program element, by providing a value for a variable, a body of code for a function, a location for a label, etc.

In discussing grammars, we have departed slightly from traditional terminology. We prefer to avoid the word *production* and the terms *right hand side* and *left hand side*. Instead, we refer to a *rule,* by which we mean *all* of the productions (and the alternative right hand sides) of a given non-terminal symbol, and to a *formulation*, by which we mean *one* of the alternatives on the right hand side of a rule.

Machine readable source files for the examples in this book can be obtained from the second author, e.g., over ARPANET, BITNET, CSNET, or USENET.

We would like to thank the reviewer for his constructive suggestions. We also gratefully acknowledge the use of the computer facilities of the Computer Science Department of the University of Illinois at Urbana-Champaign, which are supported in part by NSF grant MCS 81-05896, and of the Sektion Informatik of the University of **Ulm,** with which this book was developed, composed, and typeset.

ATS
HGF
Urbana, Illinois

Chapter 1
Language Definition

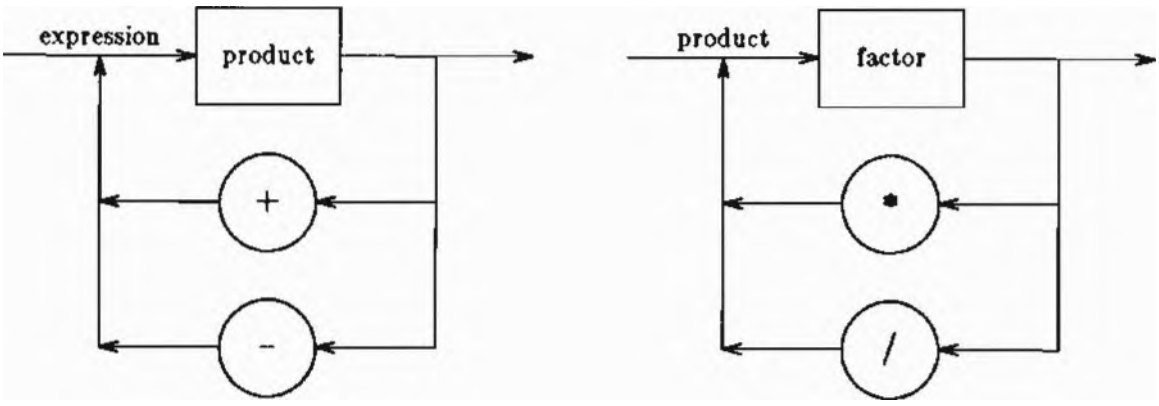
A compiler accepts *a source program* — a program written in some *source language* — and constructs an equivalent *object program* written in an *object language*, such as assembler language or binary machine language. In other words, a compiler must recognize an input source program and check it for consistency of grammar and meaning, then compose an equivalent object program.

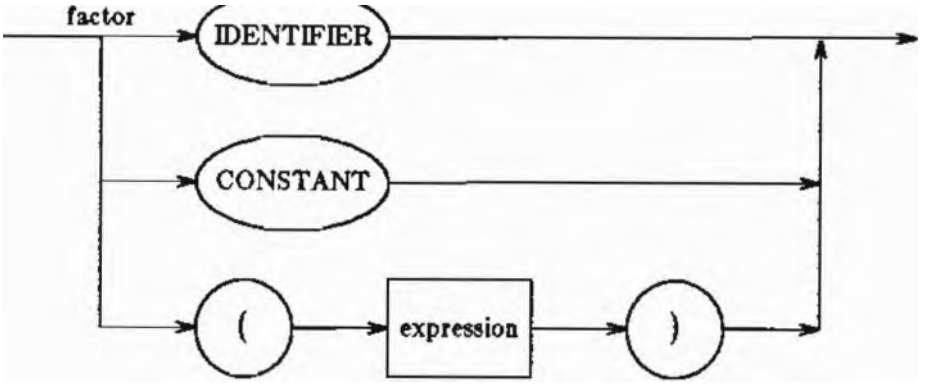
Before we can build a compiler, we need to discuss the mechanics of language definition. The present chapter will describe methods of specifying the grammar of a source language, and of checking the correctness of that grammar. The next chapter will show how the individual "words" of the source language can be recognized. In chapter 3, we will combine these two aspects of a compiler so that it can recognize a correct source program, and in chapter 4, we will extend it so that it can do something "reasonable" even with an incorrect source program. Subsequent chapters will show how to save information needed during compilation and how to use this information to finally generate an object program.

1.1 Purpose

Two aspects constitute a language definition: *syntax* and *semantics*. The syntax deals with the mechanical aspects, namely whether or not a sequence of words (or letters) is a *sentence* in the language. What the sentence means — and sometimes whether it is legitimate on that account — is determined by the semantics of the language.

Formal notations exist for both parts of the language definition. The syntax is usually explained through a sequence of models describing the parts of a sentence. *Syntax graphs*, pioneered by Wirth in the definition of Pascal, are drawn like flowcharts:





Backus Naur Form (BNF) is ***a*** language in which the syntax of a language (even of BNF itself) can be specified:

```
expression
        : expression '+' product
        I expression '-' product
        | product

product
        : product '*' factor
        I product '/' factor
        I factor

factor
        : IDENTIFIER
        I CONSTANT
        I 'C' expression ')'
```

We will discuss BNF in more detail later.

Describing the semantics of a language ***is*** much harder. If a formalism is employed, it essentially simulates execution of a sentence on a more or less well-defined theoretical machine model, and the resulting machine states — legitimate or not define the meaning of the sentence, or rule the sentence out as meaningless.

We can use plain English to describe the meaning of a sentence or of a part thereof. In the absence of a formal notation, simplicity and precision become extremely important — Wirth's Pascal or Modula-2 definitions, [Jen75] and [Wir82], are very good examples, and just about any PL/I or Fortran language reference manual tends to be verbose to the point of destruction.

There is a tradeoff between syntactic and semantic description of limitations imposed on a sentence. Consider, e.g., Basic, where an identifier can be defined as

```
id
        : LETTER DIGIT
        I LETTER

identifier
        : id '$'
        I id
```

If the operator + is used for addition between numerical values and for concatenation between strings, the semantic description must state that + may not be used to "add" a string to a numerical value or vice versa.

Alternatively, we can define

```
real_id
        : LETTER DIGIT
        | LETTER

string_id
        : real id
```

In this fashion we can distinguish string and numerical computations in a purely syntactic fashion throughout the language definition.

A Basic identifier is certainly a borderline case between what can (and should) be specified syntactically, and what can (and should) be specified semantically. In general in defining a language, we should not do in English what can be done (sensibly) in a more formal way.

1.2 Mechanical aspects

BNF is a formalism to describe the syntax of a language. It was pioneered by Peter Naur in the Algol 60 Report [Nau63] and has since been used to describe numerous languages. It has also been extended and overloaded, perhaps to a point where the resulting description is no longer easily grasped [Wij75].

A ***grammar,*** the BNF description of a language, consists of a sequence of ***rules.*** A rule **consists** of a ***left-hand side*** and a ***right-hand side,*** separated by a colon. The left-hand side consists of a single, unique ***non-terminal symbol'.*** The right-hand side consists of a sequence of one or more ***formulations,*** separated from one another by a bar. Each formulation consists of a sequence of zero or more non-terminal and terminal symbols. Only one formulation in a rule may be empty; since this can be the only formulation in the rule, the entire right-hand side may be empty. An example of a grammar was shown in section 1.1:

```
expression
        : expression '+' product
        | expression '-' product
        | product

product
        : product '*' factor
        | product '/' factor
        | factor

factor
        : IDENTIFIER
        | CONSTANT
        | '(' expression ')'
```

[1] **We only discuss context-free language definitions in a very informal manner. For a comprehensive treatment consult, e.g., [Aho77].**

A grammar defines a language by explaining which sentences may be formed. The non-terminal symbol on the left-hand side of the first rule is termed the *start symbol*. Here the start symbol is expression. The first rule lists all possible formulations for the start symbol. Each formulation may introduce new non-terminal symbols, such as product in this example.

For each non-terminal, a rule must exist, and any one of the formulations from this rule can be substituted for the non-terminal. Substitution continues until a sequence of terminal symbols, a *sentence, is produced* from the start symbol of the grammar. The terminal symbols are not further explained in the grammar; they are the alphabet in which are written the sentences of the language which the grammar describes.

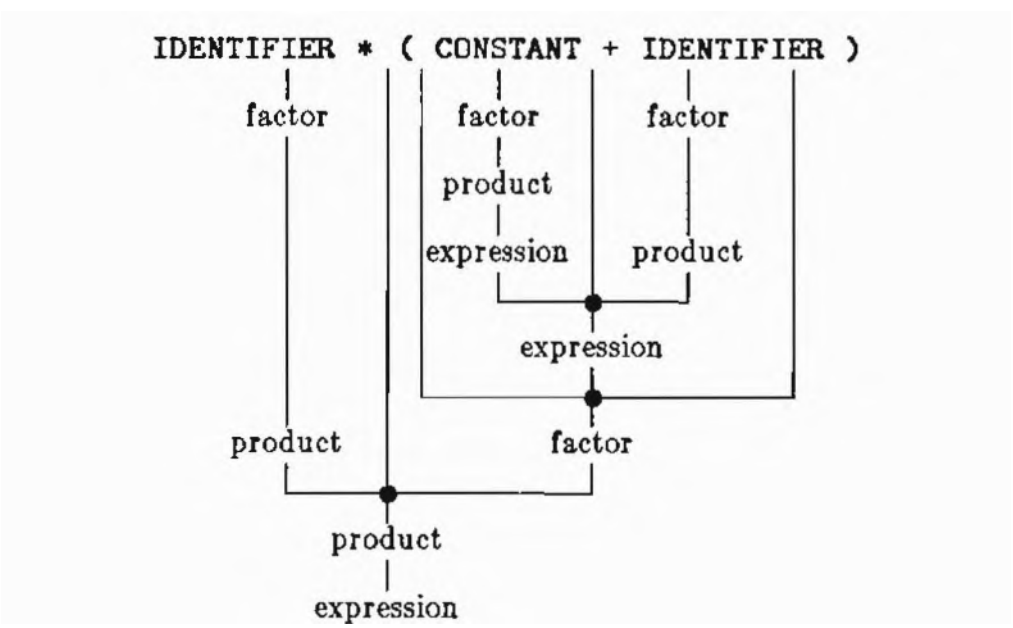
Once the rules of the grammar involve non-terminal symbols in a recursive fashion, infinitely many — and infinitely long — sentences are possible. In our example we have, e.g., the following sentence:

```
IDENTIFIER
```

by choosing the formulations product for expression, factor for product, and IDENTIFIER for factor. Another example is

```
IDENTIFIER * ( CONSTANT + IDENTIFIER )
```

Here, we must use number of intermediate formulations before we can conclude that this is in fact a sentence described by the grammar. It is customary to arrange these formulations as a parse *tree*. The root of this ordered tree is labeled with the start symbol of the grammar. The leaves are, in order, labeled with the terminal symbols of a sentence. Each non-terminal node is labeled with a non-terminal symbol, and the branches from a node lead to nodes which, in order, are labeled with the symbols from a formulation of this non-terminal.



BNF is itself a language and can be described in BNF:

```
grammar
        : grammar rule
        | rule

rule
        : rule '|' formulation
        | NONTERMINAL ':' formulation
        | NONTERMINAL

formulation
        : formulation symbol
        | symbol

symbol
        : NONTERMINAL
        | TERMINAL
```

This description is slightly more restrictive than the informal definition given above: here an empty formulation must be the first one in a rule. (Why?!)

As the examples show, recursion plays a major role in BNF. Rules are usually written in *a left-recursive* fashion — the non-terminal to be formulated appears again at the beginning of one of its own formulations, thus giving rise to an infinitely long sequence of like phrases. This technique tends to obscure simple situations, and especially language reference manuals therefore extend BNF with iterative constructs such as brackets [and] to enclose optional items and braces { and } to enclose items which may appear zero or more times. Sometimes parentheses are also employed, to introduce precedence and factor the selection operation (bar) and normal concatenation of symbols in a sequence. Extended BNF (or one variant thereof) can be included in our description of BNF. We merely need to replace the rule for symbol by the following:

```
symbol
        : NONTERMINAL
        | TERMINAL
        | '{' formulation '}'
        | '[' formulation ']'
```

Our grammar for arithmetic expressions can then be modified:

```
expression
        : product { '+' product }
        | product { '-' product }

product
        : factor { '*' factor }
        | factor { '/' factor }
```

factor remains as above.

Extended BNF can describe itself:

```
grammar
          rule { rule }

rule
        : NONTERMINAL     [ formulation ] { 'I' formulation }

formulation
        : symbol { symbol }

symbol
          NONTERMINAL
        f TERMINAL
        | '{' formulation '}'
        : '[' formulation ']'
```

Our discussion has been quite informal. Still, a word on representation is perhaps in order: *non-terminal symbols* are specified like identifiers in a programming language — they consist of letters and possibly digits and underscores; they start with a letter. *terminal symbols* tend to be spelled in upper case; if they are single special characters, we enclose them in single quotes. White space (spaces and tabs) is usually insignificant, and merely serves to delimit other symbols.

1.3 Convenient grammars

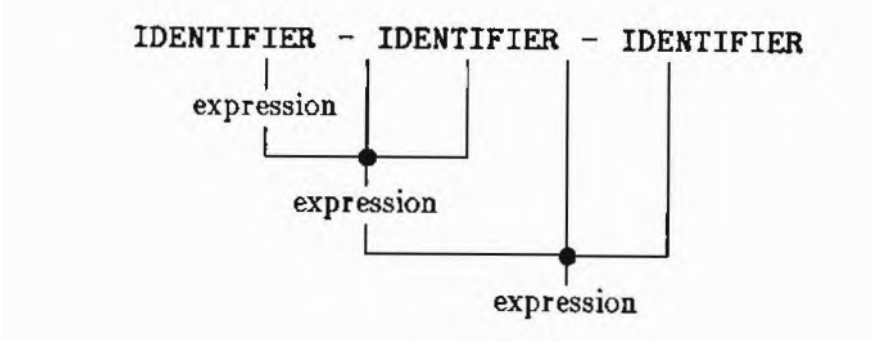
If we define a language to be a set of sentences, i.e., of sequences of terminal symbols, there are usually many ways to define a grammar describing the language. While a language need not be finite, a grammar is by definition required to be finite. This restriction alone, however, is not sufficient. Consider the following two grammars describing very simple arithmetic expressions:

```
expression
        : expression '-' IDENTIFIER
        | IDENTIFIER
```

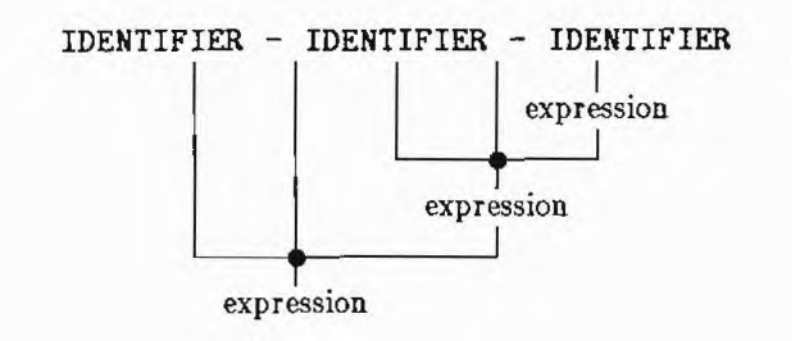
and

```
expression
        : IDENTIFIER '-' expression
        I IDENTIFIER
```

For both grammars, the language consists of all alternating sequences of IDENTIFIER and −. There is an important difference, which we can see if we compare the parse trees for an expression:



The left-recursive grammar collects terminal symbols beginning on the left; the right-recursive grammar builds (in this case) a mirror image of the first parse tree:



In the context of arithmetic expressions, the parse tree is *interpreted* to also describe precedence and associativity of operators, i.e., to define the order in which the terms of an arithmetic expression need to be combined for evaluation. This example introduces only one operator and there is therefore no precedence problem. Depending on the grammar, however, we would introduce different ways to implicitly parenthesize: the left-recursive grammar implies that the operator – is left-associative, i.e., is implicitly parenthesized from the left, and the right-recursive grammar implies the insertion of parentheses from the right. Most operators are left-associative and must therefore be introduced using left recursion.

Unfortunately, this is not the only problem. Consider the following proposal for a grammar, still for the same language of arithmetic expressions:

```
expression
        : expression '-' expression
        | IDENTIFIER
```

With this rule, we can build *either* parse tree shown above. The grammar is therefore called *ambiguous:* there exists a sentence in the language for which two different parse trees may be built. Ambiguity is often a property of the grammar, not usually of the language. As the previous examples indicate, there exist non-ambiguous grammars for the same language. We can make the proposal unambiguous by insisting that the parse tree must be built up starting on the left — this condition is known as a *disambiguating rule'.*

As a first task, a compiler must recognize the language it is to translate, i.e., for every sequence of terminal symbols, it must be able to determine efficiently whether or not it is a sentence of the language. The problem can be dealt with by attempting to build a parse tree. If we have an unambiguous grammar for the language, we can begin with its start symbol, try each formulation in turn, substitute for each non-terminal, and attempt to arrive at a parse tree for a proposed sentence. In an organized fashion, this amounts to backtracking and is thus inherently inefficient. Additionally, if the compiler produces output while it constructs the parse tree, backtracking cannot even be readily accomplished, since it would involve undoing some of the output!

A convenient grammar for language recognition must not only be unambiguous, it must be deterministic: the as yet unused rest of the input, i.e., the tail of the sequence

of terminal symbols proposed as a sentence, together with the partially constructed parse tree, must enable us to uniquely decide which rule and formulation to use in order to complete the parse tree if it exists, or to discover that no tree can be constructed to accommodate the next input symbol.

Different notions of convenience exist, depending largely on the ingenuity of the compiler writer and the power of his tools. A simple sufficient condition is, for example, that each formulation start in a unique terminal symbol. In this case the next, single input symbol determines what needs to be done.

The grammars for modern programming languages tend to follow a less restrictive pattern: they usually possess the LL(1) or LR(1) property, or variants thereof. In the first case, a parse tree can be built top-down without backtracking; in the second case, the parse tree can be built bottom-up without backtracking using certain tables, for which construction programs exist. In each case we only need to know the *next* input symbol at all times, i.e., we require *one symbol look-ahead*.

The LL(1) property is simply this: whenever there is a question as to which rule or which formulation to use, the next input symbol must enable us to uniquely decide what to do. The question arises when we need to choose one of several formulations in a rule. If none of the formulations is empty, each starts either in a terminal or in a non-terminal symbol. The next input symbol can be one of these terminal symbols; alternatively, considering recursively all the formulations for all the (first) non-terminal symbols, we arrive at more terminal symbols at the beginning of formulations. The next input symbol *must* be *exactly one* of these terminal symbols, which therefore must all be different. If there is an empty formulation, we need to additionally consider the initial terminal symbols of all the formulations for non-terminal symbols which can *follow* in the present situation.

Deciding the LL(1) property is not really difficult, since it can be phrased in terms of relations such as *a terminal symbol starts a formulation for a non-terminal symbol* and *two symbols follow each other* in *a formulation*. Such relations can be expressed as Boolean matrices [Grind, and more complicated relations can be composed and computed. Using the LL(1) property is even easier: once a grammar is LL(1), a recursive recognizes for its language can be built in a very straightforward manner; see [Wir77].

Even if a grammar is definitely not LL(1), certain semantic tricks can be used to make language recognition deterministic. Consider a simplified excerpt for Pascal:

```
statement
        : IDENTIFIER ':=' expression
        I IDENTIFIER
```

The first formulation describes an assignment, the second one a procedure call. Both versions of IDENTIFIER being equal, a name as next input symbol would not decide which formulation is to be used. The problem is usually circumvented by recognizing that only a procedure IDENTIFIER can start a procedure call; such an IDENTIFIER can, however, not start an assignment. Since in Pascal names need to be declared before they can be used, we can solve our syntactic problem with a semantic trick.

1.4 Checking a grammar

A compiler must decide whether or not a sequence of terminal symbols is a sentence of a language. A grammar describes a language. A natural question is whether we can use a grammar more or less directly in the recognition process.

First of all, however, the grammar should be checked: there must be rules for all non-terminals, all non-terminals must be reachable from the start symbol, and the grammar should satisfy a property such as LL(1) so that it is suitable for recognition and not ambiguous.

Johnson's *yacc*, a powerful utility in the UNIX system [John], can be used to check a grammar in this fashion. *yacc* accepts a grammar specified in BNF and (for the present discussion) will indicate any problems which make the grammar unsuitable for language recognition. As an example, we prepare the following input file grammar:

```
        first example of a yacc grammar
*/

%token IDENTIFIER



expression
          expression '-' IDENTIFIER
        I IDENTIFIER
```

and test it with the command

```
yacc grammar
```

Nothing happens, and this is as it should be — the grammar is acceptable according to *yacc*. Actually, there will be a new file *y.tab.c*, which will be discussed in chapter 3.

As the example shows, input to *yacc* uses the representation for BNF discussed earlier. Since terminal symbols and non-terminal symbols look alike, *yacc* requires that the terminal symbols be defined using %token specifications prior to the actual grammar. A line containing %% must separate the two parts of the input filet. White space and C-style comments are ignored.

On a more technical level, let us look at how *yacc* checks a grammar. (The computational aspects of this are discussed by Horning in chapter 2.C. in [Bau76] or in [Aho74].) *yacc* has what amounts to an option for debugging a grammar; if we issue the command

```
yacc -v grammar
```

we obtain the following file *y.output:*

```
state 0
        $accept : _expression tend
```

2 We will introduce the relevant aspects of the input language for *yacc* as we go along. As a reference, the reader is referred to [Joh78].

```
        IDENTIFIER shift 2
           error

        expression goto 1

state 1
        $accept : expression Send
        expression : expression- IDENTIFIER

        Send accept
        - shift 3
        • error


state 2
        expression : IDENTIFIER      (2)

        . reduce 2


state 3
        expression : expression - IDENTIFIER

        IDENTIFIER shift 4
        . error


state 4
        expression : expression - IDENTIFIER      (1)

        . reduce 1


4/127 terminals, 1/175 nonterminals
3/350 grammar rules, 5/550 states
0 shift/reduce, 0 reduce/reduce conflicts reported
3/225 working sets used
memory: states,etc. 26/4500, parser 0/3500
3/400 distinct lookahead sets
0 extra closures
3 shift entries, 1 exceptions
1 goto entries
0 entries saved by goto default
Optimizer space used: input 9/4500, output 4/3500
4 table entries, 0 zero
maximum spread: 257, maximum offset: 257
```

We show the entire *y.output* file here, but only the first few lines of each state are of interest now. A more complete explanation of *y.output* will be provided in section 3.1.

yacc adds the rule

$accept : expression $end

to the grammar and places its position marker, indicated by an underscore in the rule, just before the start symbol, i.e., expression in this case. This is termed state 0. *yacc* then tries to move the position marker across the start symbol.

Whenever the position marker is located before a non-terminal, the position marker is placed before all of its formulations in parallel. A *state* is a set of formulations with position marked, which is derived in this fashion. A formulation together with the position mark is called a *configuration.*

A new state is computed as follows: for each configuration in the old state in turn, the position marker is moved across the next symbol. Within the old state, this is done for all configurations with the same next symbol in parallel. For each possible symbol, we thus reach a new set of configurations, termed a new state.

As described above, if the position marker in a configuration precedes a non-terminal symbol, all formulations for that symbol must be added to the state, with the position marker at the beginning of each new formulation.

The procedure must terminate, since there is by definition a finite number of rules and non-terminals, of formulations, and of terminals in a grammar, and since the formulations must be of finite length. The set of states is thus finite.

In the example, state 0 is constructed implicitly by *yacc.* Since the position marker precedes the non-terminal expression, we must add all formulations for it with the position marker at the beginning of each formulation:

```
expression : _expression - IDENTIFIER
expression : _IDENTIFIER
```

This is omitted *in y.output,* since it is obvious from the point of view of the *yacc* algorithm.

In the state, the position marker precedes expression and IDENTIFIER. Moving it across expression in all possible configurations, we obtain state 1; moving it across IDENTIFIER, we obtain state 2. In neither state the position marker precedes a non-terminal, and no new configurations result. State 2 only contains a *complete configuration,* where the position marker has reached the end of a formulation.

In state 1 we can move the position marker across the fictitious Send symbol added by *yacc,* and across the terminal - symbol in the other configuration. The former operation is essentially ignored; the latter produces state 3.

Again, state 3 cannot be extended, and we can only move the position marker across IDENTIFIER to reach state 4. This is the last possible state, since its only configuration is complete.

The algorithm just described is, of course, the simulation of using all parts of the grammar for input recognition. Moving the position marker across a terminal, a *shift* action, means to accept the terminal at a certain position in an input sequence; moving across a non-terminal symbol implies that a formulation for the non-terminal symbol has been completed elsewhere in the simulation. (This is discussed in more detail in chapter 3.)

The comments following the configurations in each state in *y.output* outline what actions would be taken during recognition for each possible input symbol. A period stands for any *other symbol* and reduce indicates the fact that a complete configuration, i.e., a certain formulation, would be used to collect a number of accepted symbols and to replace them by the appropriate non-terminal. reduce is accompanied by the number of the formulation to be used. shift indicates the next state which will

be entered upon accepting the respective symbol.

1.5 Common pitfalls

It looks like nothing can go wrong in this analysis, but this is not the case. Forgetting to define terminal symbols, forgetting to define a rule for a non-terminal, or adding non-terminals and rules inaccessible from the start symbol are obvious errors, which ***yacc*** will discover and report immediately.

Problems with the grammar are more subtle and need individual attention. Consider the following excerpt dealing with the usual `if` statement:

```
/*
        dangling else
 */

%token IF...THEN
%token ELSE

%%

statement
        : '.'               /* empty */
        | IF...THEN statement
        | IF...THEN statement ELSE statement
```

yacc will complain about ***a shift/reduce conflict*** and if we look at ***y.output,*** we find the following:

```
4: shift/reduce conflict (shift 6, red's 2) on ELSE
state 4
        statement : IF...THEN statement        (2)
        statement : IF...THEN statement ELSE statement

        ELSE shift 6
        . reduce 2


state 6
        statement : IF...THEN statement ELSE statement

        IF...THEN shift 3
           shift 2
        . error

        statement goto 6
```

The problem is quite common. It concerns the question to which `if` the else belongs in the following program fragment:

```
if (condition)
        if (condition)
                                /* empty statement */
        else
```

In terms of ***pace, in*** state 4 we could still accept (shift) the terminal symbol ELSE and move on in the second configuration, or we could consider the first, complete configuration and substitute the non-terminal (reduce) statement. Shifting means to extend the innermost if statement, i.e., to connect each else to the innermost if, and this is what languages like C and Pascal require.

Technically, state 4 exhibits a shift/reduce conflict, a mistake in the proposed grammar, which makes it ambiguous and thus unsuitable for language recognition. *yacc*, however, intentionally permits such conflicts, and (as the comments indicate) it will provide for the longest possible input sequence, i.e., it will do just what is normally required.

The next problem is more serious. The setting is an excerpt from a Basic dialect, where the result of a comparison can be assigned to a variable or used in a decision:

```
/*
        multiple clauses
 */

%token IDENTIFIER, IF, THEN, SUM

%%

statement
        : IDENTIFIER '=' expression
        | IF condition THEN statement

condition
        : expression
        | SUM '<' SUM

expression
        : SUM '<' SUM
        | SUM
```

yacc will note a ***reduce/reduce conflict,*** and investigation of ***y.output*** reveals the following:

```
state 14
        condition : SUM < SUM_      (4)
        expression : SUM < SUM_     (6)

        . reduce 4
```

State 14 contains more than one complete configuration. In this case, ***pace*** would use the formulation introduced first in the grammar, but this tends to be risky **if** the grammar is later modified or rearranged. While we usually tolerate shift/reduce conflicts such as the dangling else problem, we always attempt to rewrite the grammar to eliminate formulation combinations which provoke reduce/reduce conflicts. In the present case this is easy — admittedly, the example is too simple to be realistic:

```
/*
        multiple clauses, no conflicts
 */
```

```
%token IDENTIFIER, IF, THEN, SUM

%%

statement
        : IDENTIFIER '=' expression
        I IF expression THEN statement

expression
          SUM '<' SUM
```

vice has a very permissive syntax for its own input. For typographical reasons, we have even omitted the semicolon which may follow each rule. The inadvertent addition of an empty formulation can provoke a rather startling number of conflicts!

Another ambiguity problem is, in fact, a virtue. Consider the following grammar for arithmetic expressions:

```
expression
        : expression '+' expression
        | expression '-' expression
        | expression '*' expression
        | expression '/' expression
        | IDENTIFIER
```

While the description is short and devoid of extraneous non-terminal symbols, it does not convey any notions of precedence and associativity of the operators. *yacc* permits such a situation, as long as precedence and associativity are specified explicitly in the first part of the input file as follows:

```
%token IDENTIFIER

%left   '+'
%left   '*' '/'
```

`left` defines a list of terminal symbols to be left-associative and to have equal precedence among each other. Precedence then increases in the order of appearance of successive left, right, and nonassoc lines in the input file. right, of course, indicates right-associativity, and nonassoc lists terminal symbols of equal precedence and does not permit them to associate with themselves.

This variety of ambiguity and explicit disambiguating rules is preferred — it introduces fewer non-terminals and thus streamlines a grammar.

1.6 Example

This book will show a complete compiler for *sampleC*, a. rudimentary subset of C. The various modules for this compiler will be described in an "example" section near the end of each chapter.

Here is our informal definition of *sampleC* in extended BNF, where we enclose terminal symbols composed from special characters in quotes and specify reserved words in capital letters. Identifier and Constant are additional terminal symbols; unlike

the others, they stand for classes of words and not for themselves. The definition is slightly verbose to simplify extensions left as exercises.

```
program
        : definition { definition }

definition
        : data_definition
        | function_definition

data_definition
        : INT declarator { ',' declarator } ';'

declarator
        : Identifier

function_definition
        : [ INT ] function_header function_body

function_header
        : declarator parameter_list

parameter_list
        : '(' [ Identifier_list ] ')' { parameter_declaration }

Identifier_list
        : Identifier { ',' Identifier }

parameter_declaration
        : INT declarator { ',' declarator } ';'

function_body
        : '{' { data_definition } { statement } '}'

statement
        : [ expression ] ';'
        | '{' { data_definition } { statement } '}'
        | IF '(' expression ')' statement [ ELSE statement ]
        | WHILE '(' expression ')' statement
        | BREAK ';'
        | CONTINUE ';'
        | RETURN [ expression ] ';'

expression
        : binary { ',' binary }

binary
        : Identifier '=' binary                 from right
        | Identifier '+=' binary
        | Identifier '-=' binary
        | Identifier '*=' binary
        | Identifier '/=' binary
        | Identifier '%=' binary
        | binary '==' binary    precedence      from left
        | binary '!=' binary
        | binary '<' binary     precedence
```

```
        | binary '<=' binary
        | binary '>' binary
        | binary '>=' binary
        | binary '+' binary     precedence
        | binary '-' binary
        | binary '*' binary     precedence
        | binary '/' binary
        | binary '%' binary
        | unary

unary
        : '++' Identifier
        | '--' Identifier
        | primary

primary
        : Identifier
        | Constant
        | '(' expression ')'
        | Identifier '(' [ argument_list ] ')'

argument_list
        : binary { ',' binary }
```

The semantics of *sampleC* are those of C, suitably pruned. The language has only an int data type, functions with a variable and arbitrary number of parameters and with int result, global, function-local and block-local scalar variables, and the control structures `if-else` and `while`. A number of the C operators are supported. Only one file will be compiled at a time, and a `main()` function must be present. This last condition will have to be explicitly accounted for; see chapter 5.

The informal definition above was translated into BNF. This involved mostly elaborating the repetitive constructs using recursion. A list of %token definitions and precedence relationships had to be added. The following result is acceptable to *yacc*:

```
/*
 *      sample c
 *      syntax analysis
 *      (s/r conflict: one on ELSE)
 */

/*
 *      terminal symbols
 */

%token  Identifier
%token  Constant
%token  INT
%token  IF
%token  ELSE
%token  WHILE
%token  BREAK
%token  CONTINUE
%token  RETURN
%token  ';'
%token  '('
```

```
%token  ')'
%token  '{'
%token  '}'
%token  '+'
%token  '-'
%token  '*'
%token  '/'
%token  '%'
%token  '>'
%token  '<'
%token  GE      /* >= */
%token  LE      /* <= */
%token  EQ      /* == */
%token  NE      /* != */
%token  '&'
%token  '^'
%token  '|'
%token  '='
%token  PE      /* += */
%token  ME      /* -= */
%token  TE      /* *= */
%token  DE      /* /= */
%token  RE      /* %= */
%token  PP      /* ++
%token  MM      /* -- */
%token  ','

/*
 *      precedence table
 */

%right  '=' PE ME TE DE RE
%left   '|'
%left   '^'
%left   '&'
%left   EQ NE
%left   '<' '>' GE LE
%left   '+' '-'
%left   '*' '/' '%'
%right  PP MM

%%

program
        : definitions

definitions
        : definition
        | definitions definition

definition
        : function_definition
        | INT function_definition
        | declaration

function_definition
```

```
        : Identifier '(' optional_parameter_list ')'
          parameter_declarations compound_statement

optional_parameter_list
        : /* no formal parameters */
        | parameter_list

parameter_list
        : Identifier
        | parameter_list ',' Identifier

parameter_declarations
        : /* null */
        | parameter_declarations parameter_declaration

parameter_declaration
        : INT parameter_declarator_list ';'

parameter_declarator_list
        : Identifier
        | parameter_declarator_list ',' Identifier

compound_statement
        : '{' declarations statements '}'

declarations
        : /* null */
        | declarations declaration

declaration
        : INT declarator_list ';'

declarator_list
        : Identifier
        | declarator_list ',' Identifier

statements
        : /* null */
        | statements statement

statement
        : expression ';'
        | ';'   /* null statement */
        | BREAK ';'
        | CONTINUE ';'
        | RETURN ';'
        | RETURN expression ';'
        | compound_statement
        | if_prefix statement
        | if_prefix statement ELSE statement
        | loop_prefix statement

if_prefix
        : IF '(' expression ')'

loop_prefix
```

```
        : WHILE '(' expression ')'

expression
        : binary
        | expression ',' binary

binary
        : Identifier
        | Constant
        | '(' expression ')'
        | Identifier '(' optional_argument_list ')
        | PP Identifier
        | MM Identifier
        | binary '+' binary
        | binary '-' binary
        | binary '*' binary
        | binary '/' binary
        | binary '%' binary
        | binary '>' binary
        | binary '<' binary
        | binary GE binary
        | binary LE binary
        | binary EQ binary
        | binary NE binary
        | binary '&' binary
        | binary '^' binary
        | binary '|' binary
        | Identifier '=' binary
        | Identifier PE binary
        | Identifier ME binary
        | Identifier TE binary
        | Identifier DE binary
        | Identifier RE binary

optional_argument_list
        : /* no actual arguments */
        | argument_list

argument_list
        : binary
        | argument_list ',' binary
```

1.7 A note on typography

As we remarked before, *yacc* input is almost entirely free format, but certain conventions are helpful:

Each nonterminal should appear just once on the left hand side of a rule. We place the nonterminal alone on a line, at the left margin. All formulations are then listed together.

The colon between left-hand and right-hand side of *a* rule is indented one tab position, the first formulation is indented one blank past the colon, and actions — to be discussed in chapter 3 -- will be indented two tab positions.

The colon, the 1 symbol introducing an alternative formulation, and the semicolon terminating the rule are all vertically aligned.

In order to conserve space in the book, we do not terminate a rule by a semicolon. While this is acceptable to *yacc*, it is, however, a bad idea in practice to omit the semicolon, since omitting it tends to obscure typographical errors such as replacing a colon by a bar.

Rules are separated from one another by a blank line.

Empty formulations are clearly indicated by a suitable comment — this is particularly helpful for debugging.

We prefer long names, joined together by underscores; periods are also acceptable for this purpose. We also tend to use the suffix _list to indicate a comma-separated list of things, and plural to indicate a sequence of things.

1.8 Problems

1. Design a. *small* subset of Pascal, and write a BNF description of it. Add token definitions and precedence relations to the BNF description. Submit this language definition to yacc, and correct it if necessary until is acceptable to *yacc* (i.e., until *yacc* does not issue any error messages other than unavoidable shift/reduce conflicts).

2. Extend the definition of *sampleC* by adding another standard feature of C, such as arrays. Change the grammar given in section **1.6** so that the new description of *sampleC is* acceptable to *yacc*.

3. Write a grammar for EBNF in BNF. A grammar for EBNF in **EBNF** can be found in section 1.2. What changes must be made to convert this grammar to BNF? Add token definitions and whatever else is needed to make the resulting grammar acceptable to *yacc*. Hint: to remove all shift/reduce conflicts, you will want to include a semicolon to terminate each rule.

Chapter 2
Word Recognition

2.1 Purpose

We have seen in the previous chapter that a language is a set of sentences, which in turn are sequencesof terminal symbols. Terminal symbols in a programming language usually come in three varieties: operators are represented as (short) sequences of special characters, reserved words are represented as predefined sequences of letters whose meaning cannot vary, and user-defined terminal symbols encompass constants and identifiers subject to a specialized syntax definition. Then, of course, there is white space — blanks, tab characters and line separators — which in most modern programming languages merely separates terminal symbols but is otherwise insignificant. Comments, too, follow their own syntax, different for just about every programming language [Wic73] and just as insignificant as white space.

This chapter deals with *lexical analysis,* that phase of the compilation process which assembles terminal symbols from the unstructured sequence of input characters presented to the compiler. White space and comments are usually ignored, while operators and reserved words are identified and passed on using an internal representation — typically small integer constants. User-defined constants and identifiers need to be saved in an appropriate table, and a generic representation such as Identifier or Constant together with a reference to the table entry passed on.

We could attempt to solve the lexical analysis problem simultaneously with the actual language definition in the following manner:

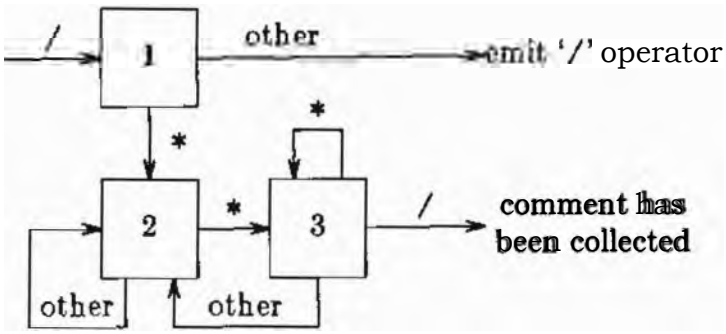
```
statement
        'I' 'F' condition 'T' 'H' 'E' 'N' statement
```

The resulting grammar, however, is bound to be loaded with conflicts and the technique is horribly inefficient.

Lexical analysis accounts for a large amount of the processing which a real compiler does. By dealing with lexical analysis mostly independently from the rest of the compilation process, we can employ more appropriate techniques and can at the same time hide within a single module all knowledge about the actual representation of our programming language in a real world character set.

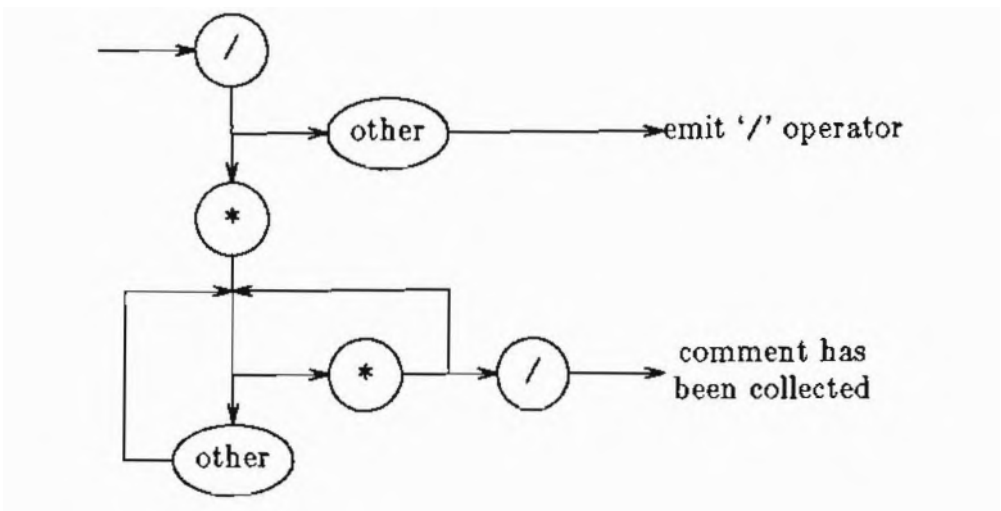
2.2 Definition tools

How *do* we assemble characters into terminal symbols? Lexical analysis is the classical application for the theory of finite state automata. A transition diagram is generally easy to devise from the lexical specification of a programming language, and an eight to sixteen hour *ad-hack* approach loosely based on the transition diagram will cope with just about any such specification. Things usually get messy when we need to dispose of strings, comments, and (floating-point) constants, in about that order. By way of illustration let us consider a C-style comment described as a transition diagram:



In a transition diagram, the states are the numbered nodes, and the transitions are the branches between the states, labeled with the input characters causing the transition. Characteristically, the exit from the diagram with the collected terminal symbol is somewhat haphazard — sometimes the following character has already been seen, and sometimes it has not.

A transition diagram corresponds quite closely to a syntax graph. The branches in the transition diagram describe the transitions; the nodes in the syntax graph contain the symbols which need to be found in order to move on. The difference is that syntax graphs regularly call one another, while these simple transition diagrams are not supposed to.



A comparison of the two types of diagrams suggests why transition diagrams are preferred in this approach to lexical analysis. The approach itself, however, is error prone, and the result tends to be quite hard to modify.

A better solution for the problem of lexical analysis can be derived directly from the theory: we need a convenient way to describe the finite state automaton corresponding to the lexical specification of our language, we need a compiler to produce appropriate tables from the description, and we need an interpreter to simulate the finite state automaton defined by the tables.

Such a compiler has, in fact, been written: it is called ***lex*** [Les78b]. *lex* accepts a table of patterns resembling editor patterns[1] and produces a table-driven C program

1 See `ed(1)` in [Ker78a].

capable of recognizing input strings satisfying the patterns. As the *substitute* command in the editor shows, patterns can be conveniently used to identify specific character sequences.

Patterns are a very convenient specification language for the finite state automaton actually constructed. With each pattern a C statement can be associated which will be executed when a string satisfying the pattern is found in the input. In simple translation applications the C statement will usually write a modified copy of the string to standard output; in compiler applications the C statement should return an appropriate encoding of the string to the caller of the lexical analysis function.

As we shall see, *lex* is a very powerful tool in its own right, and it can be used to great advantage in language recognition. However, *lex* is, in our opinion, quite unfriendly to use: error messages are short and entirely unspecific, commenting input for */ex* is cumbersome at best, and brute force uses of *lex* can create huge programs. Still, *lex* is the tool of choice, since the alternative — hand-coding the lexical analysis function — requires a significantly higher investment in manpower.

The rest of this chapter **will** introduce in reasonable detail a sufficient number of *lex* features to cope with most compiler applications. We will first describe the most frequently used operations in patterns, how patterns are specified to *lex,* and how typical language constructs such as identifiers, strings, and comments can be specified as patterns. While we explain most possibilities, the chapter is not intended as a comprehensive description.

In section 2.5 we introduce *lex* as a program generator and we show a few small but complete *lex* programs for file inclusion, file splitting, and word counting. Our *sampleC* implementation is continued in section 2.7 where the complete lexical analyzer for *sampleC* is presented.

2.3 Patterns

Users of *ed* and similar text editors are already familiar with the following constituents of patterns:

Letters, digits, and some special characters represent themselves.

Period represents any character, with the exception of line feed.

Brackets, [and 1, enclose a sequence of characters, which is termed a *character class.* The class represents any one of its constituents, or any single character *not* in the given sequence if the sequence starts with ^. Within the sequence, — between two characters denotes the inclusive range.

If * follows one of these pattern parts, it indicates that the corresponding input may appear arbitrarily often, or even not at all.

at the beginning of a pattern represents the beginning of an input line.

$ at the end of a pattern represents the end of an input line (but not the line feed character itself).

With a suitable escape convention for special characters and white space, we can write patterns for most terminal symbols of a programming language. There are two e

\ quotes a single following special character; in particular, two \ characters represent a single \. 1 may precede the letters b, n, and t; the combinations then denote backspace, line feed, and tab characters, just as in C.

A better technique, especially for sequences of characters, is to enclose one or more characters in double-quotes. The characters thus lose their special meaning.

Most special characters have a special meaning in *lex;* if a special character should represent itself, it is best quoted. Special characters need not be quoted inside a character class. White space must always be quoted or represented by a \ escape sequence. A double-quote can be introduced within double-quotes with **\"**, just as in C.

By way of example, let us look at patterns for some terminal symbols in *sample C.* The operators are simply quoted — we generally prefer double-quotes to the backslash convention:

```
"*"

"*="
```

and so on. We need a pattern to recognize white space:

```
[\t\n]
```

Constants consist of one or more digits:

```
[0-9][0-9]*
```

Identifiers follow the rules of C, i.e., they start in a letter or underscore and continue with arbitrarily many letters, digits, or underscores:

```
[A-Za-z_][A-Za-z0-9_]*
```

The following pattern attempts to deal with single-line comments:

```
"/*".*"*/"
```

A comment should start in /* and terminate with the *first* occurrence of */. It can extend over many lines.

The last two patterns exhibit certain limitations with which casual users of *ed* should be quite familiar: a pattern, by definition, represents the *longest* possible input sequence. In the case of constants and identifiers, this is really what is desired; in the case of comments, however, we need exactly the opposite! There is also a question as to what happens if we specify

```
"<"
"<="
```

i.e., two patterns which represent the same initial sequence of characters, or

```
"int"
[a-z][a-z]*
```

i.e., two patterns where one represents a subset of the possibilities of the other. We will deal with ambiguities in the next section.

In order to deal successfully with C- and Pascal-style comments and strings, we need to introduce a few more *lex* pattern features. Just as for *egrep*[2] parentheses may

[2] See **grep(1)** in [Ker78a].

grouping within a pattern, I denotes alternatives, + denotes one or more of the item preceding it, and ? denotes zero or one occurrence. The last two features simplify some of the patterns shown earlier, e.g., integer constants are recognized with

```
[0-9]+
```

A terminal symbol which is delimited by single characters, such as a Pascal-style comment enclosed in braces, is easy to handle:

```
"{"[ }]*"}"
```

Using an alternative, the technique extends easily to Pascal- or C-style strings, where the delimiter must be duplicated or escaped if it is to appear within the string:

```
\'([^'\n]|\\['\n])+\'
```

Note that a Pascal-style string as defined here cannot extend over several lines, and that it must contain at least one character. A C string can extend over several lines if the line feed is escaped using \, it can contain escaped double-quotes, and it can be empty.

If there is a sequence of characters used as a delimiter, as in C-style comments enclosed by /* and */, or in Pascal-style comments enclosed by (* and *), the necessary pattern becomes quite complicated. The basic idea is not to permit the terminating delimiter once the opening delimiter has been recognized; unfortunately, this can only be done by enumeration. Consider the following proposal for a pattern to recognize a C-style comment:

```
/*"([ */]|[ *]"/"| *"[ /])*"*/"
```

The comment begins with /* and ends with */. In between can be zero or more occurrences of one of three alternatives, namely any character but * or /, a / preceded by any character except *, or finally a * followed by any character other then /. Note that a character class using the complement operator ^ is somewhat dangerous it usually includes line feed and thus can easily swallow an entire input file!

Alas, our 'solution' for C-style comments is not quite perfect: /*/ is not recognized as the beginning of a comment, since in this pattern, / must *follow* a character in the comment, and /***/ is not recognized as a complete comment, since in this pattern * must *precede* another character in the comment, which causes the third * to be ignored as part of the delimiter. A better solution is the following pattern, which admittedly is even harder to read:

```
"/*"*"/"*([ */]|[ *]"/"|"*"[ /])*"*"*"*"*/"
```

Now zero or more slashes may immediately follow the opening delimiter, and zero or more asterisks may precede the closing delimiter. Thus the special cases are also accounted for.

There is yet more to *lex* patterns. Iteration can be limited to a specific range using a notation like

```
[a-z][a-z0-9]{0,7}
```

to denote an identifier starting in a lower case letter, which is followed by zero to seven letters or digits. A pattern may be recognized only before a certain right context, which itself is not represented by the pattern; the right context is given following a / mark:

```
-0[xX]/[0-9a-f]+
```

would recognize the sign and the base prefix of a negative hexadecimal C constant, but would not include the digits of the constant itself. Patterns can be preceded by start conditions, which essentially permit the dynamic selection of one of several sets of patterns specified together. Finally, a very able (and careful) programmer can even interact directly with the input, output, and buffer managing routines employed by the automaton generated by *lex*. The patient and imaginative reader is cheerfully referred to the original publication [Les78b].

2.4 Ambiguity as a virtue

A set of *lex* patterns tends to be highly ambiguous. Two rules are employed by *lex* to sort things out:

lex always chooses that pattern which represents the longest possible input string.

If two patterns represent the same string, the first pattern in the list presented to *lex* is chosen.

Both rules are, in fact, assets rather than liabilities. The first rule asserts that the patterns

```
int
[a-z]+
```

completely recognize integer (as an instance of the second pattern) and not only the initial int. The second rule causes the pattern `int` and not the second, more general pattern to recognize `int` from the input.

As has been demonstrated in the previous section, the first disambiguating rule can backfire if used carelessly. The second rule, however, encourages an arrangement in which the most general pattern, e.g., for an identifier, is placed last, with more selective patterns preceding it to pick off exceptions.

Consider the following recognition problem: when typing German texts on an American keyboard, *umlaut* characters are most quickly typed as ae, oe, ue, etc. Before the resulting document is presented to nroff or *trot* to be processed, e.g., with the *ms* macro package [Les78a], one should replace all these letter combinations by invocations of the *ms* string `\*:`, as for example in `\*:` a, *:o, etc. The pattern

```
ue
```

however, recognizes the letter combination ue even in contexts such as `Quelle` (German: fountain), or eventuell (German: possibly), where it does not represent the umlaut. We therefore need to pick off these special cases *before* we present the general combination:

```
[Qq]ue
ntue
ue
```

Since ambiguous pattern lists are acceptable as input to *lex*, we can simply insert more specials cases as we encounter them.

Unlike *make*, where only in the case of `reduce/reduce` conflicts does input order make a difference, it greatly matters in *lex* since conflicts in the sense of *yacc* are frequently encountered. This, unfortunately, often leads to illogical arrangements of the patterns. The order of patterns does imply something like a control structure.

2.5 "lex" programs

Input to *lex* consists of one file with three parts, separated by lines beginning in

```
first part

%%
pattern         action
        ...
%%
third part
```

The first part is optional; it can contain lines controlling the dimensions of certain tables internal to *lex*, it can contain definitions for text replacements as we shall see in section 2.7, and it can contain (global) C code preceded by a line beginning in `%{` and followed by a line beginning in `%}`. Even if the first part of the *lex* specification is empty, the separator %% between the first and second parts of the *lex* specification cannot be omitted.

The third part and the separator preceding it are optional. The third part contains C code which is used as is. As we shall see, it usually contains (local) functions which the second part uses.

The second part of the specification consists of a table of patterns and actions. This part of the specification is quite line-oriented. A pattern starts at the beginning of a line and extends to the first non-escaped white space. Following an arbitrary amount of white space, an action is specified which is thus associated with the pattern. The action is a single C statement, or several statements enclosed in a set of braces. The action may also consist of a bar I, indicating that the present pattern will use the same action as the next pattern.

lex input itself has no provisions for comments(!), but within braces, regular C comments can be written.

From the table, *lex* will construct a C function `yylex()` in the file *lex.yy.c*. If this function is linked with a program and called, standard input is read until the next, longest possible string represented by a pattern has been collected. The action associated with the pattern is then executed. If this action contains a return statement, `yylex()` will return, possibly with a. function value as dictated by the return statement.

fez adds a default pattern and action to the patterns specified by the user in such a way that all otherwise unrecognized input characters are copied to standard output.

The following *lex* program will remove all upper case letters from its input:

```
%{
/*
        remove upper case letters
 */
%}

%%

[A-Z]+
```

Assuming that the program is contained in a *file exuc.l*, the following commands produce an executable program *exuc*:

```
lex exuc.l
cc lex.yy.c -11 -0 exuc
```

-11 references the *lex* library, which contains a default main() function, which will just call `yylex()` once. This library must always be supplied when a function `yylex()` generated by *lex is* to be linked.

More useful actions need to have access to the input string recognized by the pattern with which they are associated. The char vector yytext t] contains this string, null-terminated, the int variable `yyleng` has `strlen(yytext)` *as* a value, and the int variable `yylineno` contains the number of the current input line. Let us look at a few marginally useful programs:

```
/*
        line numbering
 */

%%

        ECHO;
^.*$    printf("%d\t%s", yylineno, yytext);
```

This first program prints standard input and precedes each nonempty input line by its number and a tab character. ECHO is defined within the C program produced by *lex*; **it causes yytext** [] to be **printed.**

If we also want to number the empty lines, the following program can be used:

```
%{
/*
        line numbering
 */

%%

        printf("%d\t%s", yylineno-1, yytext);
```

Now we recognize line feed as part of the pattern. `yylineno` is already incremented to the next line when our action gets control.

```
%{
/*
 *      word count
 */

int     nchar, nword, nline;
%}

%%

\n              ++ nchar, ++ nline;
[^ \t\n]+       ++ nword, nchar += yyleng;
.               ++ nchar;

%%

main()
{
        yylex();
        printf("%d\t%d\t%d\n", nchar, nword, nline);
}
```

This example shows a use for the third part of a *lex* specification: we include our own main() function, which here prints the statistics gathered during execution of the `yylex()` function.

The next example is typical of a large class of problems, where a special action — here it is file inclusion — needs to be taken when just one pattern is recognized. Most of the input is just passed through. Similar applications include, e.g., adorning reserved words for the publication of a program, gathering a table of contents, etc.

```
%{
/*
 *      .so filename    file inclusion
 */

#include <ctype.h>
#include <stdio.h>

static include();
%}

%%

^".so".*\n      { yytext[yyleng-1] = '\0'; include(yytext+3); }

%%

static include(s)
        char * s;
{       FILE * fp;
        int i;

        while (*s && isspace(*s))
```

```
                ++ s;
        if (fp = fopen (s, "r"))
                while (0. = getc(fp)) != EOF)
                        output (1);
                fclose (fp) ;

        else
                perror(s);
```

output() is the routine which *lex* uses to write all output. The program can only handle one level of file inclusion.

The last example deals with depositing output in several files, selected by options in the input. This is a variant of the *split* utility.

```
%{
/*
        .di filename    file splitting
 */

#include <ctype.h>
#include <stdio.h>
static divert();
%}

%%

^".di".*\n      { yytext[yyleng-1] = 'W; divert(yytext+3); }

%%

static divert(s)
        char * s;

        while (*s && isspace (*s))
                s;
        if (! freopen(s, "w", stdout))
                perror(s);
}
```

This example is quite difficult to handle with tools such as *sed* or *awk*, where the number of output files is severely limited. Since *lex* can pick the file name from within a complex context, this solution is significantly easier to modify than a hand-written C program.

2.6 Testing a lexical analyzer

For use as a lexical analyzer in a compiler, `yylex()` should be designed as a function returning an int value. Upon each call, the next terminal symbol should be collected from the input, encoded, and returned as the function value. Each pattern now is designed to recognize one or more terminal symbols, and the associated action will contain a return statement to produce the function value. We must make provisions so that *all* input characters are recognized, even if they neither belong to terminal symbols, nor are to be ignored silently, since `yylex()` is supposed to let all input

disappear in this case.

Testing such a. function can be messy — the function values are numbers and as such usually not terribly mnemonic. We found a C programming trick quite helpful.[2] For debugging purposes we arrange for the *lex* program to have the following principal g

```
%{
#include <assert.h>
#define token(x)        (int) "x"
main()
{       char * p;

        assert(sizeof(int) >= sizeof(char *));
        while (p = (char *) yylex())
                printf("%s is \"%s\"\n", p, yytext);
}
%}

%%

pattern         return token(MNEMONIC);
```

As the *sampleC* example will show, it is possible to build a lexical analyzer in such a way that it can always be conditionally compiled for testing purposes in this fashion.

The caller of yylex() receives the value MNEMONIC in printable form as a result value of `yylex()`. The simple main() routine will then display the input text yytext and a decoded representation of the returned value together for debugging purposes.

By convention, `yylex()` is expected to return zero(!) as an end of file indication. *lex* generates the function `yylex()` so that this happens internally; main() is coded to terminate once `yylex()` returns zero.

One last advice: the reserved words of a programming language are particularly easy to write down as patterns. Unfortunately, a. long list of such self-representing patterns dramatically increases the size of the program generated by *lex*. It is much more efficient to collect reserved words and identifiers with the same, single pattern and to screen the results with a small C function. A standard approach to this probkm is shown in the next section.

2.7 Example

From the *yacc* specification of *sampleC* we know which terminal symbols must be found by the lexical analysis routine for this compiler: yylex() needs to find all symbols mentioned in %token statements and all single-character terminal symbols quoted directly. At present, using the debugging technique introduced in the previous section, we do not need to worry about the exact function values which must be returned as encodings of the various terminal symbols.

[2] Unfortunately, this trick will not work on machines such as the MC 68000, where pointer values do not fit into `int` variables. The library macro `assert()` is employed to guard against this possibility; it will abort the program if the given condition is not met.

The patterns pose no problems, since we will eventually run the C preprocessor prior to our own compiler. The C preprocessor will remove comments(!), and it makes the usual #define, **#include**, and conditional compilation facilities available for our *sample C* implementation.

The complete, final lexical analyzer is shown below. DEBUG must be defined when we compile the lexical analyzer for testing purposes. A few lines in the following file should therefore be ignored at present; they will be explained in chapter 3 where we integrate our language recognition program.

```
%{
/*
 *      sample c -- lexical analysis
 */

#ifdef  DEBUG           /* debugging version - if assert ok */

#       include <assert.h>

        main()
        {       char * p;

                assert(sizeof(int) >= sizeof(char *));

                while (p = (char *) yylex())
                        printf("%-10.10s is \"%s\"\n", p, yytext);
        }

        s_lookup() {}
        int yynerrs = 0;

#       define  token(x)        (int) "x"

#else   ! DEBUG         /* production version */

#       include "y.tab.h"
#       define  token(x)        x

#endif  DEBUG

#define END(v)  (v-1 + sizeof v / sizeof v[0])
static int screen();
%}

letter                          [a-zA-Z_]
digit                           [0-9]
letter_or_digit                 [a-zA-Z_0-9]
white_space                     [ \t\n]
blank                           [ \t]
other                           .

%%

^"#"{blank}*{digit}+({blank}+.*)?\n     yymark();
```

re turn tok n (GE) ;

```
"<="                            return token(LE);
"=="                            return token(EQ);
"!="                            return token(NE);
"+="                            return token(PE);
"-="                            return token(ME);
"*="                            return token(TE);
"/="                            return token(DE);
"%="                            return token(RE);
"++"                            return token(PP);
"--"                            return token(MM);

{letter}{letter_or_digit}*      return screen();

{digit}+                        { s_lookup(token(Constant));
                                  return token(Constant);
                                }

{white_space}+                  ;

{other}                         return token(yytext[0]);

%%

/*
 *      reserved word screener
 */

static struct rwtable {         /* reserved word table */
        char * rw_name;         /* representation */
        int rw_yylex;           /* yylex() value */
        } rwtable[] = {         /* sorted */
        "break",        token(BREAK),
        "continue",     token(CONTINUE),
        "else",         token(ELSE),
        "if",           token(IF),
        "int",          token(INT),
        "return",       token(RETURN),
        "while",        token(WHILE),
        };

static int screen()
{       struct rwtable * low = rwtable,
                * high = END(rwtable),
                * mid;
        int c;

        while (low <= high)
        {       mid = low + (high-low)/2;
                if ((c = strcmp(mid->rw_name, yytext)) == 0)
                        return mid->rw_yylex;
                else if (c < 0)
                        low = mid+1;
                else
                        high = mid-1;
        }
        s_lookup(token(Identifier));
```

```
        return token(Identifier);
}
```

Assuming that this text is in the file *samplec.l*, the lexical analyzer is compiled for testing purposes as follows:

```
lex samplec.l
cc -DDEBUG lex.yy.c -11 -o lex1
```

We can run some tests through this lexical analyzer to check if the proper terminal symbols are recognized. The simplest, complete program is as follows:

```
main ()
```

For this program the lexical analyzer will return the following:

```
Identifier is "main'
yytext[0] is "("
yytext [0] is ")"
yytext[0] is "{"
yytext [0]  is "}"
```

We took a very simple approach to recognizing single character operators and at the same time finding and signaling all illegal input characters: if no other pattern catches a character, it is returned itself as value of `yylex()`. In this fashion we obtain a compact lexical analyzer, and we will later deal with all input errors in a systematic way.

A few other features of the lexical analyzer are perhaps worth mentioning. *lex* has a rudimentary text replacement facility within the patterns. A line of the form

```
name      replacement
```

in the first part of the *lex* specification defines a replacement text for a name. The name can then be specified as

```
{name}
```

within a pattern, and the replacement will be substituted. We use this facility often, usually to make the patterns more transparent and mnemonic.

The first pattern deals with lines of the form

```
# linenumber filename
```

Such lines are produced by the C preprocessor as position stamps during file inclusion and selection for conditional compilation. As will be explained in chapter 3, `yymark()` is a function which we designed to record the relevant information for our own error messages.

As promised, we use *a* simple binary lookup function screen() to recognize reserved words as special cases of those terminal symbols recognized by the identifier pattern. screen() is really intended as a. blueprint solution for such a screening problem. Notice how the token technique is even used in the reserved word table!

s `lookup()` is a handle for symbol table management which we will use and explain beginning in chapter 5.

2.8 Problems

1. Write *a lex* program which will read a Pascal program and print out only (* *) comments, eliminating everything else. Hint: what modifications are needed to the *lex* pattern for C comments given in section 2.3?

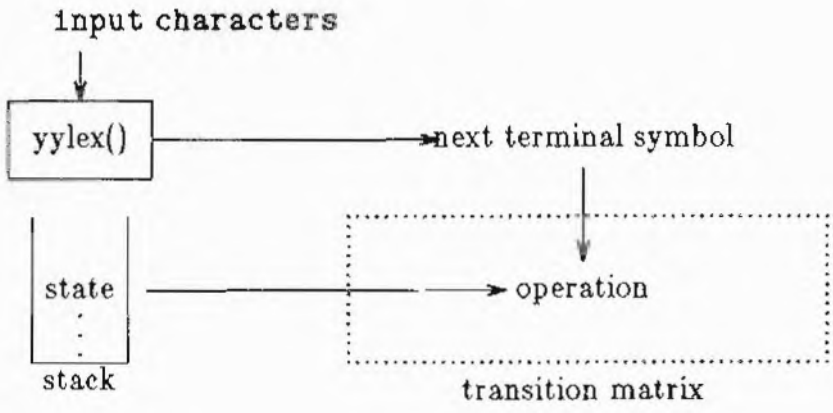
2. Write **a** *lex* pattern which will recognize a Fortran REAL constant, defined as a string of digits having either a decimal point or an E-type exponent, or both. For an additional challenge, take into account the fact that in Fortran, white space is allowed anywhere outside of quote marks, even within the characters of a constant, identifier, or keyword.

3. Using the pattern in problem 2, write a complete *lex* program which will read a Fortran program and convert all REAL constants to DOUBLE PRECISION. The program should also change all REAL variable declarations to DOUBLE PRECISION. (It is convenient that the word REAL occurs in Fortran only as a variable declaration statement, assuming that it is not foolishly used as a variable name.)

4. Write a complete *lex* program which will read a program written in your favorite programming language, recognize all of the keywords in that program, and print them in upper case letters. All other upper case letters should be converted to lower case. Hint: use a function like screen() (see section 2.7) to recognize the keywords.

5. Write a complete *lex* program which will read a program written in your favorite programming language and produce a cross-reference table of all identifiers in the program. Hint: once your *lex* program recognizes an identifier, it should use suitable C functions to compare it to a. table of language keywords, which can be ignored, and to the list of other previously recognized identifiers. New identifiers should be added to the latter list. Each list entry should be the head of a chain of cross-reference elements which record values obtained from yylineno. After `yylex()` terminates, the cross-reference table can be printed.

We can visualize the parser function as follows:



Current state, on top of the stack, and next terminal symbol, produced as needed by a call to `yylex()`, select an operation from the transition matrix. The file *_;.output* shows the contents of the transition matrix for each acceptable next terminal symbol and each state. Five types of operations will be found in the transition matrix:

accept

This operation .happens only once, namely when we have send as next terminal symbol, represented as a non-positive value of `yylex()`, and are getting ready to successfully complete recognition.

error

This operation is found as element of the transition matrix for all those next terminal symbols which must not be seen in a particular current state.

shift new *state*

This operation indicates that the next terminal symbol is acceptable in the current state. The new *state* is pushed onto the stack and becomes the current state. We have, in fact, moved on in some configuration.

reduce *formulation number*

This operation is present in the transition matrix for all those states which contain a complete configuration. The *formulation* number indicates the complete configuration; it appears following the configuration in the file *y.output*. At this point, we will pop as many states off the stack as the formulation has symbols. The uncovered state on top of the stack is the new current state. The non-terminal whose formulation was just completed is used prior to the next terminal symbol. The actual next terminal symbol will be processed following the non-terminal symbol just explained.

goto new *state*

As we just saw, a reduce operation implicitly generates a non-terminal symbol to he used prior to the next terminal symbol. goto is the shift operation for this non-terminal symbol. A. **shift** operation always uses and discards the next terminal symbol; **a** goto operation uses a non-terminal symbol and leaves the next terminal symbol for a subsequent **shift** operation. Otherwise, goto and shift

operate in the same fashion: the new *state* is pushed onto the stack and becomes the current state.

To illustrate the operations in context, let us look again at the simple grammar introduced in chapter 1:

```
expression
        : expression '-' IDENTIFIER
        | IDENTIFIER
```

In chapter 1 we introduced a file *y.output* which *yacc* can produce from this grammar. We will now once again look at this file to demonstrate that it in fact fully documents the parser. The parser starts out with state 0 as current state on the stack:

```
state 0
        $accept : _expression Send

        IDENTIFIER shift 2
           error

        expression goto 1
```

If the next terminal symbol is an IDENTIFIER, we will perform a shift operation, i.e., we will accept the symbol and push the new current state 2 onto the stack. Any other input symbol would be considered in error.

```
state 2
        expression : IDENTIFIER      (2)

           reduce 2
```

Without regard to the new next terminal symbol, we will use formulation 2 for a reduce operation, i.e., since formulation 2 consists of one symbol (IDENTIFIER), we will pop one state of the stack. Having come this far we uncover state 0.

The reduce operation has just generated expression as a non-terminal symbol, and the instructions for state C prescribe that we goto state 1 in this situation. Notice that the next terminal symbol, if any, has thus far not been considered.

```
state
        $accept :  expression_ Send
        expression : expression- IDENTIFIER

        $end accept
        - shift 3
        . error
```

In state 1 we consider the symbol following the first IDENTIFIER in the input. $end, the end of input, or a – operator are anticipated at this point. $end leads to an accept operation — our parser has recognized IDENTIFIER as a sentence! It would be instructive for the reader to follow the parser actions in *y.ouput* for a longer sentence.

One problem remains: how does yylex() know what values yyparse() expects as representations of the next terminal symbol? A natural convention is to represent single characters as terminal symbols by their value in the character set, i.e., the C

in this chapter, we will be putting things t
a lexical analyzer specification presented to *lex*
table facility, make up a language recognizer. A
nizer can be used for such things as pretty-print

To start, we need to understand the workin
from a grammar, deposits in the file *y.tab.c*, a
this background in place, we show in section 3.
ally put together.

It turns out that we must supply a main()
procedure yyerror() which will be called from
the input. In section 3.3 we take time out to pr
which allows us to optionally invoke the C prep
also construct a general yyerror() function whi
tion of an error in the input to the parser.

Unfortunately, grammars do not always im
the language designer. In section 3.4 we therefo
bugs in the recognizer. It turns out that the file
vided by *yacc* can be combined into a useful st
the grammar.

We will conclude the chapter with a form
ally implement such a program, we must still so
about terminal symbols from the lexical analy
actions enter the picture in *yacc* specifications
constructing a simple desk calculator.

3.1 Parser generation

We explained in section 1.4 how ***yacc***
amounted to traversing all rules in a highly par
terminal symbol acceptance. If we can somel
things must actually get simpler: rather than fol
lel, ***yacc*** can then move through only those s
The result must be a device for language recogn

yacc, in fact, builds a parser while analyzi
down automaton — a stack machine — consi
states, a transition matrix to derive a new st
current state and next input symbol, a table of
executed at certain points in the recognition, an
mit execution. The result is packaged as a func
on a lexical analyzer function yylex() to read s
or one to indicate whether or not a sentence was

Chapter 3
Language Recognition

together: a grammar presented to ***yacc,*** , combined with a rudimentary symbol As the example will show, such a recog- ting programs.

ngs of the parser which *yacc* constructs nd describes in the file ***y. output.*** With 2 bow a recognizer for *sampleC* ***is*** actu-

program to drive the recognizer and a n the parser if an error is discovered in resent a comfortable version of main() processor prior to our own compiler. We ich gives a clear indication of the posi-

mediately reflect the true intentions of ore discuss how one goes about finding e y. ***output*** and a debugging option pro- rategy to locate "misunderstandings" in

atter for *sampleC.* Before we can actu- lve the problem of passing information yzer to the parser. Thus, in section 3.5, , and we demonstrate their flexibility by

c analyzes a grammar. The analysis rallel fashion while simulating arbitrary how feed real input to this algorithm, llowing numerous possibilities in paral- tates which are selected by the input. nition, a ***parser*** or ***syntax analyzer.***

ing the grammar. The parser is a push- sting of a "large" stack to hold current ate for each possible combination of user-definable actions which are to be nd finally an interpreter to actually per- ction yyparse 0, which calls repeatedly tandard input, and which returns zero s presented as input file.

constant 'x' will represent the terminal symbol 'x' in the grammar presented to *yacc*. For a terminal symbol name, introduced by a %token statement in *yacc*, however, `yyparse 0` and `yylex()` must use the same integer value as a representation; this value must be distinguishable from the representation of single characters.

yacc aids in defining suitable values. The command

```
yacc -d grammar.y
```

instructs *yacc* to produce a file *y.tab.h* containing one C preprocessor #define statement for each name introduced as %token.[1] The replacement text for each name is a unique integer constant, starting at 257.

The file *y.tab.h* is already present within the file *y.tab.c*, which is also produced by *yacc*, and which contains the function `yyparse()`. The same terminal symbol representations can thus be used for `yylex()` by including *y.tab.h* with a C preprocessor #include statement in the definition of the lexical analysis function.

This requires, however, that names used in the grammar for terminal symbols and introduced through %token statements cannot be reserved words in C. The lexical analysis function `yylex()` can be written by hand, or it can be produced by (ex as discussed in chapter 2.

3.2 Example

In section 1.6, we presented *sampleC in* a form acceptable to *yacc*. Assume that this definition is in a file *samplec.y*. In section 2.7, we showed the file *samplec.l* containing a lexical analyzer for the terminal symbols of *sampleC*. We can now put both functions together to obtain a parser for *sample C*.

If we do *not* define DEBUG while compiling the lexical analyzer, the C preprocessor statements

```
#include "y.tab.h"
#define token(x) x
```

in *samplec.l* will take effect. The first causes the terminal symbol representations produced by *yacc* in *y.tab.h* to be used in compiling `yylex()`. The second statement disables our debugging technique for the lexical analyzer: for debugging, we returned the terminal symbol names via

```
#define token(x) int "x"
```

as printable C strings. Now we return the *values* for those names, as defined in *y.tab.h*.

The very last pattern in the lexical analyzer:

```
{other}          return token(yytext [OD ;
```

takes care of returning the character values for all unrecognized single character terminal symbols. This pattern is placed last so that a single letter or digit is recognized by

[1] Terminal symbol names can also be introduced through a first appearance in %left, %right, o %nonassoc statements. We prefer to always define terminal symbols first with %token.

earlier patterns as an Identifier or a Constant.

We produce the recognizer with the following commands:

```
lex samplec.l
yacc -d samplec.y
cc lex.yy.c y.tab.c -ll
```

Actually — yet not quite. If we compile in this fashion, we pick the main() routine from the *lex* library and as we saw in chapter 2, this routine will only call `yylex()` once and the parser `yyparse` not at all! We need to supply a different main() routine such as

```
main()
{
        yyparse();
}
```

One more routine must also be provided by the user of yam When the parser executes an error operation in the transition matrix, there is a syntax error in the input file. At this point, an error message should be written, and yyparse () therefore issues the call

```
yyerror("syntax errors);
```

It is up to us to program a suitable `yyerror()` routine indicating the input position, etc. A trivial solution is the following:

```
#include <stdio.h>

yyerror(s)
        char * s;
{
        fputs(s, stderr), putc('\n', stderr);
}
```

We need not count the individual errors. This is handled automatically by yyparse () in the int variable `yynerrs`.

If those two routines are in a file *extra.c,* we can complete the compilation begun above with the command

```
cc extra.c lex.yy.o y.tab.o -ll
```

The resulting recognizer in file *a.out* can be executed as follows:

```
a.out
main () { }
^D
```

Nothing happens, and this is as it should be:

```
main 0 {
```

is a very small, legal *sampleC* program. Our recognizer will produce nothing at all if the input is in fact sentence. If it is not, we will be faced by a curt syntax error.

3.3 Auxiliary functions

The main() and yyerror() routines shown in the preceding section are of the software engineering quality of *lex*. User-friendly compilers should pinpoint input errors at least at the line number level, and the error messages should be more than just syntax error. In this section we discuss a general approach to this problem.

We begin with the main() routine. It turns out that it is quite convenient to always be able to invoke the C preprocessor to obtain standard file inclusion, text replacement, and conditional compilation facilities. Also, the C preprocessor will remove C-style comments, unless the —C option is specified. Our standard main() routine will invoke the C preprocessor if at least one of the option arguments known to the preprocessor is used; we can thus always provoke preprocessing through the —P or —E option:

```
/*
 *      main() -- possibly run C preprocessor before yyparse()
 */

#include <stdio.h>

static usage(name)
        register char * name;
{
        fputs("usage: ", stderr);
        fputs(name, stderr);
        fputs(" [C preprocessor options] [source]\n", stderr);
        exit(1);
}

main(argc, argv)
        int argc;
        char ** argv;
{       char ** argp;
        int cppflag = 0;

        for (argp = argv; ++argp && **argp == '-'; )
                switch ((*argp)[1]) {
                default:
                        usage(argv[0]);
                case 'C':
                case 'D':
                case 'E':
                case 'I':
                case 'P':
                case 'U':
                        cppflag = 1;
                }
        if (argp[0] && argp[1])
                usage(argv[0]);
        if (*argp && ! freopen(*argp, "r", stdin))
                perror(*argp), exit(1);
        if (cppflag && cpp(argc, argv))
                perror("C preprocessor"), exit(1);
        exit(yyparse());
}
```

The routine checks all arguments. Options always start with -. If there is an option known to the C preprocessor, the preprocessor will be invoked through our function `cpp()` described below, which is given access to all the arguments. Following the options, there may be one file name argument, which will be opened as the source file. After argument processing, main() calls yyparse() and propagates the return value as exit() code of the process. As a test bed, this routine is quite convenient. When a compiler nears completion, the routine is usually extended with more option processing and usage information output for illegal arguments.

The routine `cpp()`, which actually runs the C preprocessor, is somewhat involved. An elegant solution which avoids having temporary files hinges upon being able to connect a pipeline so that `yylex()` is implicitly forced to read from it. By definition, `yylex()` reads single characters by calling a routine input() defined as a macro by *lex*. It turns out that this routine in turn reads from the file pointer yyin which is externally accessible. This can be used in `cpp()`:

```
/*
 *      cpp() -- preprocess lex input() through C preprocessor
 */

#include <stdio.h>

#ifndef CPP                     /* filename of C preprocessor */
#       define  CPP     "/lib/cpp"
#endif

int cpp(argc, argv)
        int argc;
        char ** argv;
{       char ** argp, * cmd;
        extern FILE * yyin;     /* for lex input() */
        extern FILE * popen();
        int i;

        for (i = 0, argp = argv; **++argp; )
                if (**argp == '-' &&
                    index("CDEIUP", (*argp)[1]))
                        i += strlen(*argp) + 1;

        if (! (cmd = (char *) calloc(i + sizeof CPP, sizeof(char))))
                return -1;      /* no room */

        strcpy(cmd, CPP);
        for (argp = argv; **++argp; )
                if (**argp == '-' &&
                    index("CDEIPU", (*argp)[1]))
                        strcat(cmd, " "), strcat(cmd, *argp);

        if (yyin = popen(cmd, "r"))
                i = 0;          /* all's well */
        else
                i = -1;         /* no preprocessor */
        cfree(cmd);
        return i;
}
```

cpp() first measures the string length of all options known to the C preprocessor. It then acquires memory for a composite string and builds a command to call the C preprocessor with those options. Using `popen()` from the standard library, the C preprocessor is connected as a filter to `yyin`, the command string is freed, and `cpp()` returns zero it all this has worked.

Turning now to error reporting, we first present a standard header for messages giving a sensible amount of positioning information. We would like to include the name of the source file, especially when files are preprocessed by the C preprocessor. The current line number `yylineno` is maintained correctly by *lex* as long as no preprocessing takes place. The current or next token can be found in yytext [] but this token may be a line feed; at the end of the source file it might even be empty. In both cases, the actual line number of the error is less than `yylineno`. A lot of information can be provided in an automated fashion, but it needs to be carefully formatted:

```
/*
 *      yywhere() -- input position for yyparse()
 *      yymark() -- get information from '# line file'
 */

#include <stdio.h>

extern FILE * yyerfp;           /* error stream */

extern char yytext[];           /* current token */
extern int yyleng;              /* and its length */
extern int yylineno;            /* current input line number */

static char * source;           /* current input file name */

yywhere()                       /* position stamp */
{       char colon = 0;         /* a flag */

        if (source && *source && strcmp(source, "\"\""))
        {       char * cp = source;
                int len = strlen(source);

                if (*cp == '"')
                        ++cp, len -= 2;
                if (strncmp(cp, "./", 2) == 0)
                        cp += 2, len -= 2;
                fprintf(yyerfp, "file %.*s", len, cp);
                colon = 1;
        }
        if (yylineno > 0)
        {       if (colon)
                        fputs(", ", yyerfp);
                fprintf(yyerfp, "line %d",
                        yylineno -
                        (*yytext == '\n' || ! *yytext));
                colon = 1;
        }
        if (*yytext)
        {       register int i;
```

```
                for (i = 0; i < 20; ++ i)
                        if (!yytext[i] || yytext[i] == '\n')
                                break;
                if (i)
                {       if (colon)
                                putc(' ', yyerfp);
                        fprintf(yyerfp, "near \"%.*s\"", i, yytext);
                        colon = 1;
                }
        }
        if (colon)
                fputs(": ", yyerfp);
}

yymark()                        /* retrieve from '# digits text' */
{
        if (source)
                cfree(source);
        source = (char *) calloc(yyleng, sizeof(char));
        if (source)
                sscanf(yytext, "# %d %s", &yylineno, source);
}
```

We define yyerfp as a separate file pointer, which is used for all messages. The compiler designer can thus still choose to emit the messages as standard output (the default), or to write them to a separate file simply by assigning a different file pointer to yyerfp.

yymark() is a routine which extracts source file and line information from the position stamps emitted by the C preprocessor; see section 2.7.

If all possible parts are present, the message header produced by yywhere() would appear as follows:

```
source.c, line 10 near "badsymbol":
```

where **badsymbol** usually is the symbol following the error.

We are now ready to write a standard yyerror() routine which better pinpoints the location of an error

```
#include <stdio.h>

FILE * yyerfp = stdout;         /* error stream */

yyerror(s)
        register char * s;
{       extern int yynerrs;     /* total number of errors */

        fprintf(yyerfp, "[error %d] ", yynerrs+1);
        yywhere();
        fputs(s, yyerfp);
        putc('\n', yyerfp);
}
```

yynerrs is a counter which is maintained by *yacc*; this counter is incremented once after each call to yyerror() is issued. Especially in a long compilation protocol,

it is quite helpful if the error messages are sequentially numbered.

3.4 Debugging the parser

The parser is ready for testing. Unfortunately, nothing at all will happen if we present a correct input file to our parser for processing. If the input file is incorrect, we will receive one more or less useful error message. If we believe that the input file is correct, but the error message appears anyhow, things can get messy: we need to discover why `yyparse()` decided to issue a call to the error message routine.

The technique outlined in section 2.6 enables us to verify that the lexical analyzer is not at fault, i.e., that `yyparse()` actually received the symbols which we assume to be in the input file. Once this has been verified, we can use a debugging facility provided by *yacc*: if we compile the parser *y.tab.c* with the symbol `YYDEBUG` defined

```
cc -DYYDEBUG y.tab.c lex.yy.c main.c yyerror.c -ll
```

the resulting yyparse () function contains a tracing option which can be enabled by setting the `int` variable yydebug to a nonzero value. yydebug is a global variable and can for example be set with *adb:*

```
adb -w a.out
yydebug?w 1
$q
```

We illustrate the results of tracing with slightly modified versions of the simple expression grammar introduced in section **1.4.** Here is a rudimentary lexical analyzer:

```
#include "y.tab.h"

[a-z]+          return IDENTIFIER;
[ \t\n]+        ;
.                return yytext[0];
```

The first version of the grammar defines - to be left-associative as a disambiguating rule:

```
%token  IDENTIFIER
%left   '-'
%%

expression
        : expression '-' expression
        | IDENTIFIER

%%

main()
{
#ifdef  YYDEBUG
        extern int yydebug;

        yydebug = 1;
```

```
#endif
        printf("yyparse() == %d\n", yyparse());
}
```

Just like a *lex* specification, the input file for *yacc* may also contain a third part preceded by %%. This part is copied into *y.tab.c*; here it contains a main() routine which will set the debug flag to call yydebug if it exists. For illustration purposes, the return value of yyparse is shown.

yacc will produce the following *y.output* file for this parser:

```
state 0
        $accept : _expression $end

        IDENTIFIER  shift 2
        .  error

        expression  goto 1

state 1
        $accept :  expression_$end
        expression :  expression_- expression

        $end  accept
        -  shift 3
        .  error


state 2
        expression :  IDENTIFIER_    (2)

        .  reduce 2


state 3
        expression :  expression -_expression

        IDENTIFIER  shift 2
        .  error

        expression  goto 4

state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        .  reduce 1
```

The statistical information has been omitted.

Assume that the inputs to *lex* and *yacc* are in the files *exp.l* and *exp.y*. We construct the parser and execute it with a correct input:

```
yacc -dv exp.y
lex exp.l
cc -DYYDEBUG y.tab.c lex.yy.c -ll -o exp
exp
a - b - c
^D
```

```
state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        -  shift 3
        .  reduce 1
```

```
[yydebug] recovery pops 3, uncovers 1
[yydebug] recovery pops 1, uncovers 0
[yydebug] recovery pops 0, stack is empty
yyparse() == 1
```

```
%token  IDENTIFIER
%nonassoc '-'
%%

expression
        : expression '-' expression
        | IDENTIFIER
```

```
state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        -  error
        .  reduce 1
```

Tracing the input

```
a - b - c
```

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reading '-'
[error 1] line 1 near '-': syntax error
[yydebug] recovery pops 4, uncovers 3
[yydebug] recovery pops 3, uncovers 1
[yydebug] recovery pops 1, uncovers 0
[yydebug] recovery pops 0, stack is empty
yyparse() == 1
```

```
/*
 *      desk calculator
 */

%token  Constant
%left   '+' '-'
%left   '*' '/'
%%

line
        : /* empty */
        | line expression '\n'
                { printf("%d\n", $2); }

expression
        : expression '+' expression
                { $$ = $1 + $3; }
        | expression '-' expression
                { $$ = $1 - $3; }
        | expression '*' expression
                { $$ = $1 * $3; }
        | expression '/' expression
                { $$ = $1 / $3; }
        | '(' expression ')'
                { $$ = $2; }
        | Constant
                /* $$ = $1; */
```

```
expression : expression '+' expression
{   $$     =     $1      +      $3;  }
```

```
expression : '(' expression ')'
{   $$     =          $2;      }
```

```
expression : Constant
/*  $$     =   $1; */
```

```
line : line expression '\n'
{ printf("%d\n", $2     );}
```

```
%{
/*
 *      lexical analyzer for desk calculator
 */

#include "y.tab.h"
extern int yylval;
%}
%%

[0-9]+          { yylval = atoi(yytext); return Constant; }
[ \t]+          ;
\n              |
                 return yytext[0];
```

The library function `atoi()` computes the integer value of a string of digits. This value is recorded in `yylval` as the actual value of the Constant.

The value to be pushed during a goto operation, i.e., during acceptance of a non-terminal symbol produced by a reduce operation, is taken from the global variable `yyval` defined by *yacc*; it can be set from within the action executed during the reduce operation.

As the example shows, the action usually needs to access the values placed on the value stack during acceptance of the symbols for the formulation which is about to be reduced. The notation $1 within an action represents the value for the ith symbol in the formulation presently on the value stack; the notation $$ represents yyval, i.e., it represents the value which will be pushed onto the value stack during acceptance of the non-terminal symbol by the goto operation following the reduction.

The action

```
{ $$ = $1; }
```

is supplied by default. It states that the value stack entry of the first symbol in the formulation will become the value stack entry of the non-terminal symbol to which the formulation is reduced.

Our desk calculator works as advertised. For each Constant, the lexical analyzer provides the actual value in `yylval` which yyparse O places on the value stack. Once a formulation such as

```
expression : expression '-' expression
```

is reduced, the associated action

```
$$ = $1 - $3; }
```

computes the appropriate difference, which is pushed onto the value stack following the reduction. The printf 0 function call at the top shows the value of each expression line presented to the desk calculator.

3.6 Typing the value stack

`yylval`, `yyval`, and the value stack can be used to hold a large variety of information. By default the value stack consists of `int` elements. For our desk calculator double elements may actually be more interesting. In a compiler, `yylval` will most likely hold a pointer to a symbol table entry for each "large" terminal symbol. Unfortunately, double values — and on certain machines even pointers cannot be stored and retrieved from `int` variables.

The value stack maintained by the parser can be typed from within a *yacc* specification. In this section we will describe a way of typing the value stack which is entirely transparent to *yacc* itself; a more elaborate typing facility which prompts *yacc* to perform rather extensive semantic checks will be discussed in section 5.4.

`yylval`, `yyval`, and the value stack are defined in the parser to be of the type `YYSTYPE` itself is defined *with the C preprocesor as* int, unless an explicit definition is supplied ptphlcdrst yacc specification. Transparent to *yacc*, this definition can be supplie follows:

```
%{
#define YYSTYPE double
%}

%%

/* grammar, etc. */
```

```
typedef char *  CHAR_PTR;
#define YYSTYPE CHAR_PTR
```

```
main(a, b)
        int     a, b;
{
        a = b;
        {
                a;
                b;
        }
        if (a == b)
        {
                a;
                b;
        }
        if (a == b + 1)
```

```
                a;
        else
                b;
        while (a == b)
        {
                a;
                break;
                continue;
        }
        return;
        ;
}

int     f()
{       int     x;
        int     y;

        return nothing;
}
```

```
/*
 *      formatting call mnemonic parameters
 */

#define IN      1               /* margin inward */
#define EX      (-1)            /* margin outward */
#define AT      0               /* margin as is */

/*
 *      yacc value stack type (pass texts!)
 */

typedef char * CHAR_PTR;
#define YYSTYPE CHAR_PTR
```

```
/*
 *      sample c -- utilities for formatter
 */
```

```
#include "fmt.h"

/*
 *      rudimentary symbol table routine:
 *              save text of every symbol.
 */

s_lookup(yylex)
        int yylex;              /* Constant or Identifier */
{       extern YYSTYPE yylval;  /* semantic value for parser */
        extern char yytext[];   /* text value of symbol */
        extern char * strsave();

        yylval = strsave(yytext);
}

/*
 *      formatter calls:
 *
 *      at(AT)          no-op
 *      at(IN)          set margin inward
 *      at(EX)          set margin outward
 *      nl(delta)       emit newline, at(delta)
 *      cond(IN)        nl(IN)
 *      cond(EX)        if directly preceded by cond(IN): at(EX)
 *      uncond(AT)      nl(AT)
 *      uncond(EX)      unless just after uncond(AT): at(EX)
 *      out(s)          emit string s
 *
 *      Margin settings take effect just prior to first out(s) on
 *      the new line.  'cond' calls fiddle with braces and else.
 */

static int lmargin = 0;  /* left margin, in tabs */
static int atmargin = 1; /* set: we are at left margin */
static int condflag = 0; /* managed by cond(), = 0 by out() */
static int uncdflag = 0; /* managed by uncond(),  = 0 by out() */

at(delta)
        int delta;
{
        lmargin += delta;
}

cond(delta)
        int delta;
{
        switch (delta) {
        case IN:
                ++condflag;
                nl(IN);
                break;
        case EX:
                if (condflag)
                {       at(EX);
                        condflag = 0;
```

```
                }
        }
}

uncond(delta)
        int delta;
{
        switch (delta) {
        case AT:
                ++uncdflag;
                nl(AT);
                break;
        case EX:
                if (! uncdflag)
                        at(EX);
        }
}

nl(delta)
        int delta;
{
        at(delta);
        putchar('\n');
        atmargin = 1;
}

out(s)
        char *s;
{
        if (atmargin)
        {       register int i;

                for (i = 0; i < lmargin; i++)
                        putchar('\t');
                atmargin = 0;
        }
        fputs(s, stdout);
        condflag = uncdflag = 0;
}
```

tended:

```
/*
 *      sample c
 *      syntax analysis
 *      formatting actions
 *      (s/r conflict: one on ELSE)
 */
```

```
%{
#include "fmt.h"        /* formatter action mnemonics */
%}

/*
 *      terminal symbols
 */

/*
 *      precedence table
 */

%%

program

definitions

definition
        : function_definition
        | int function_definition
        | declaration

function_definition
        : identifier lp optional_parameter_list rp
                { nl(IN); }
          parameter_declarations
                { at(EX); }
          compound_statement
                { nl(AT); }

optional_parameter_list

parameter_list
        : identifier
        | parameter_list co identifier

parameter_declarations

parameter_declaration
        : int parameter_declarator_list sc

parameter_declarator_list
        : identifier
        | parameter_declarator_list co identifier

compound_statement
        : lr declarations
                { nl(AT); }
          statements rr

declarations

declaration
        : int declarator_list sc
```

```
declarator_list
        : identifier
        | declarator_list co identifier

statements

statement
        : expression sc
        | sc
        | break sc
        | continue sc
        | return sc
        | return
                { out(" "); }
          expression sc
        | compound_statement
        | if_prefix statement
                { uncond(EX); }
        | if_prefix statement else statement
                { uncond(EX); }
        | loop_prefix statement
                { uncond(EX); }

if_prefix
        : if lp expression rp
                { cond(IN); }

loop_prefix
        : while lp expression rp
                { cond(IN); }

expression
        : binary
        | expression co binary

binary
        : identifier
        | constant
        | lp expression rp
        | identifier lp optional_argument_list rp
        | pp identifier                 %prec PP
        | mm identifier                 %prec PP
        | binary pl binary              %prec '+'
        | binary mi binary              %prec '+'
        | binary mu binary              %prec '*'
        | binary di binary              %prec '*'
        | binary rm binary              %prec '*'
        | binary gt binary              %prec '>'
        | binary lt binary              %prec '>'
        | binary ge binary              %prec '>'
        | binary le binary              %prec '>'
        | binary eq binary              %prec EQ
        | binary ne binary              %prec EQ
        | binary an binary              %prec '&'
        | binary xo binary              %prec '^'
        | binary or binary              %prec '|'
```

```
        | identifier as binary          %prec '='
        | identifier pe binary          %prec '='
        | identifier me binary          %prec '='
        | identifier te binary          %prec '='
        | identifier de binary          %prec '='
        | identifier re binary          %prec '='

optional_argument_list

argument_list
        : binary
        | argument_list co binary

/*
 *      printing the terminal symbols
 */

int     : INT           { out("int\t"); }
identifier : Identifier { out($1); }
lp      : '('           { out("("); }
rp      : ')'           { out(")"); }
co      : ','           { out(", "); }
sc      : ';'           { out(";"); nl(AT); }
break   : BREAK         { out("break"); }
continue: CONTINUE      { out("continue"); }
return  : RETURN        { out("return"); }
lr      : '{'           { cond(EX); out("{\t"); at(IN); }
rr      : '}'           { at(EX); out("}"); uncond(AT); }
if      : IF            { out("if "); }
else    : ELSE          { at(EX); out("else"); cond(IN); }
while   : WHILE         { out("while "); }
constant: Constant      { out($1); }
pp      : PP            { out(" ++ "); }
mm      : MM            { out(" -- "); }
pl      : '+'           { out(" + "); }
mi      : '-'           { out(" - "); }
mu      : '*'           { out(" * "); }
di      : '/'           { out(" / "); }
rm      : '%'           { out(" % "); }
gt      : '>'           { out(" > "); }
lt      : '<'           { out(" < "); }
ge      : GE            { out(" >= "); }
le      : LE            { out(" <= "); }
eq      : EQ            { out(" == "); }
ne      : NE            { out(" != "); }
an      : '&'           { out(" & "); }
xo      : '^'           { out(" ^ "); }
or      : '|'           { out(" | "); }
as      : '='           { out(" = "); }
pe      : PE            { out(" += "); }
me      : ME            { out(" -= "); }
te      : TE            { out(" *= "); }
de      : DE            { out(" /= "); }
re      : RE            { out(" %= "); }
```

```
if (1)
        1;
```

and

```
if (1)
{
        1;
}
```

In both cases, `cond(IN);` is called during the reduction of `if_prefix` and it sets the left margin inward. In the first case, `statement` turns out to be an expression and the indentation remains. In the second case, statement will be a `compound_statement` and therefore `lr` will be reduced for the left brace before any other reduction takes place. The action during reduction of `lr` calls `cond(EX);` which notes from `condflag` that a call `cond` (IN) ; has just taken place. The left brace can therefore be exdented and placed underneath `if`. As promised, static variables are used to pass context information between neighboring reductions. The technique has little to recommend it other than that it avoids a lot of devious rewriting of the grammar itself.

The formulation of `compound_statement` shows a second method of specifying actions:

```
compound_statement
        : lr declarations
                { nl(AT); }
          statements rr
```

Our formatting style requires that an empty line follow declarations. The call `nl(AT);` will issue the blank line. An action can be placed anywhere in a formulation; if it does not follow the entire formulation, yacc will generate an anonymous non-terminal symbol in place of the action and define the non-terminal symbol with a empty formulation followed by the action. The formulation shown above is really expanded as follows:

```
compound_statement
          lr declarations $$123 statements rr

$$123
          /* empty */
                { nl (AT) ; }
```

Actions cannot be placed entirely at will in this fashion; the anonymous non-terminal symbols can introduce conflicts.

One problem has not been discussed: this formatter only deals with a source file that has been preprocessed, i.e., which contains no comments! In a realistic implementation, the lexical analyzer would have to collect the comments, and pass them to the formatting routine, probably attached to terminal symbols. This involves a significant amount of bookkeeping, and the problem of formatting comments is nontrivial.

3.8 Problems

1. Extend the desk calculator example so that it uses variables. A very simple extension is to predefine twenty-six variables, a through z, and the storage in which to save their values. A more interesting problem is to allow arbitrary variable names; in this case, storage both for the strings that name the variables and for their values should be acquired dynamically.

2. The formatting style used in section 3.7 may not be your favorite format. Modify the formatting program given in that section so that it conforms to your preferred standards.

3. Write a formatting program for a subset of Pascal, perhaps the one used for problem 1 of section 1.8.

4. The formatter given in section 3.7 produces a blank line following each function. This means that if the last definition in a file is a function_ definition, there will be a useless blank line as the last line of the output. Change the program to suppress this extra blank line. Hint: it may be easier to control the blank line if you emit it before each function, rather than after.

5. Write a formatting program for EBNF. It should display an EBNF grammar in a standard format, such as the one suggested in section 1.7. It is convenient to start with the solution to problem 3 of section 1.8.

6. Write a program which will convert a grammar written in EBNF to BNF. Hint: for reasons of efficiency internal to *yacc*, use left-recursion for iterations. See the discussion of the treatment of its stack by *yacc* in the left- and right-recursive definitions of

```
expression : expression '-' expression
```

in section 3.4.

7. Modify the standard */usr/lib/yaccpar* to produce the trace format shown in this chapter. For some important hints, see section A.8 in the appendix.

8. In the last paragraph of the previous section, it was mentioned that inclusion of comments in the formatted output is somewhat difficult. An approach to the solution of this problem was also suggested. Modify the example from that section, or the program from question 2, 3, or 4 above, to display comments. Suggestion: if a comment is the first "thing" on a line, display it at the beginning of the line (perhaps at the current indenting level). If the comment follows something else on the same line, display it at a user-settable or predetermined "tab" position.

9. Using techniques similar to those used in solving the previous problem, write a program which will display preprocessor lines (lines with # in column 1) as part of a program.

Real compilers deal mostly with incorrec
we can make our parser robust against input
happens inside the parser when an input err
far the parser would "fall off the stack" du
provides a special error terminal symbol to in
4.2 we demonstrate what happens if a pars
error.

The problem in general is to build robu
mulations added to some rules. Fortunately v
a number of constructs frequently found in pr
they cope with any error. Our technique is pr
we show how a robust recognizes for *sample(*
strate its behavior in case of erroneous inputs.

4.1 The problem

Thus far, our parser can recognize and
correct input file. One of the examples in s
present the incorrect input

```
a - -
```

to a parser based on the rule

```
expression
        : expression '-' expressior
        | IDENTIFIER
```

with — defined to be left-associative. yyparse

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading '-'
[error 1] line 1 near '-': expecti
[yydebug] recovery pops 3, uncover
[yydebug] recovery pops 1, uncover
[yydebug] recovery pops 0, stack i
yyparse() == 1
```

The second — in state 3 leads to an error ope

```
state 3
        expression : expression -_

        IDENTIFIER shift 2
```

Chapter 4
Error Recovery

ct input files. This chapter discusses bow
errors. We first use tracing to show what
ror is encountered. It turns out that thus
ring an error operation. However, *yacc*
nfluence the parsing algorithm. In section
ser is properly prepared to cope with an

st grammars using error symbols *in* for-
we found a straightforward way to extend
rogramming languages in such a way that
resented in section 4.3, and in section 4.5
is defined, and how we can fully demon-

l perhaps manipulate a sentence, i.e., *a*
section 3.4 showed what happens if we

n

falls off the stack:

```
ng: IDENTIFIER
s 1
s 0
s empty
```

ration in the transition matrix:

```
expression
```

```
error

expression goto 4
```

Once the error message has been issued, yyparse() seems to remove all states from the stack — obviously looking for something. Since the stack is cleared in the process, `yyparse()` returns with a function value of one, and the recognition procedure is aborted on encountering the first error in the **input!**

4.2 Basic techniques

Our lexical analyzers usually have the following entry at the end of the pattern table:

```
return yytext[0];
```

The pattern is intended to pick up all single character operators. However, this entry will return the integer value of any single character as function value of `yylex()`, i.e., as a terminal symbol, as long as the character has not been recognized by an earlier pattern.

Unexpected input characters are thus passed from the lexical analyzer to the parser as if they were legitimate terminal symbols, represented by single characters. This results in a uniform treatment of **all** input errors. An alternative approach at this level would be to have the lexical analyzer report its own problems, and then to ignore illegal characters; however, in this case it is hard to avoid cascades of messages.

At the symbol level, we can add formulations to the grammar that are probable although illegal. This technique makes our recognizer more tolerant **than** the language designer intended it to be. While we can forgive the most frequent user errors in this fashion, the technique does not have a high probability of complete success — it is nearly impossible to exactly predict an incorrect input sequence.

A better approach is to treat an input error as a special case of a terminal symbol: error is a predefined terminal symbol for *yacc*. error can be used in formulations just like **a** terminal symbol; however, error is (normally) not produced by a call to the lexical analyzer. Instead, the parser believes error to be the next terminal symbol if the actual next terminal symbol leads to an error operation in the transition matrix for the current state. Once the error symbol has been internally generated in this fashion, and the obligatory error message issued, `yyparse()` will set out to accept error almost like any other symbol.

Consider the following modification to the rule above:

```
expression
        : expression '-' expression
        | IDENTIFIER
        | error
```

A **parser based on** this grammar will silently accept erroneous input. To understand why this is the case, we need to once again follow the traces of a few examples. `yacc` will produce the following *y.output* file:

```
state 0
        $accept : _expression $end
```

```
        error shift 3
        IDENTIFIER shift 2
        . error

        expression goto 1

state 1
        $accept : expression Send
        expression :  expression_- expression

        Send accept
        - shift 4
        . error

state 2
        expression :  IDENTIFIER_    (2)

        . reduce 2

state 3
        expression :  error_    (3)

        . reduce 3

state 4
        expression :  expression -_expression

        error  shift 3
        IDENTIFIER  shift 2
        . error

        expression  goto 5

state 5
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        . reduce 1
```

Note that error appears as an operation in a number of states; in state 0 and in state 4 it also appears as a terminal symbol, for which the operation shift 3 is present in the transition matrix.

Let us now see how this parser reacts to the erroneous input

```
a - - b
```

The trace shows that up to reading the second – the parser acts just like before:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
```

```
[yydebug] reading '-'
[yydebug] push state 4
[yydebug] reading '-'
[error 1] line 1 near "-": expecting: IDENTIFIER
```

```
[yydebug] accepting $error
[yydebug] push state 3
```

```
[yydebug] reduce by (3), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] push state 4
```

```
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading [end of file]
yyparse() = 0
```

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '+'
[error 1] line 1 near "+": expecting: '-'
```

```
[yydebug] recovery pops 1, uncovers 0
[yydebug] accepting $error
[yydebug] push state 3
[yydebug] reduce by (3), uncover 0
[yydebug] push state 1
```

```
[yydebug] recovery discards '+'
[yydebug] reading '-'
[yydebug] push state 4
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading [end of file]
yyparse() = 0
```

```
a - - b + - c
```

produces only one error message:

```
[error 1] line 1 near '-': expecting: IDENTIFIER
yyparse() = 0
```

The trace reveals that both errors are, in fact, discovered:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
...
[yydebug] reading '-'
[error 1] line 1 near '-': expecting: IDENTIFIER
[yydebug] accepting $error
...
[yydebug] reading '+'
[yydebug] recovery pops 1, uncovers 0
```

```
[yydebug] accepting $error
...
[yydebug] recovery discards '+'
...
yyparse 0 = 0
```

In order to avoid a cascade of error messages, the parser must shift three terminal symbols beyond the point of error, before another error results in an error message. This way a cluster of errors may result in only a single error message. In this example, this explains the absence of the second error message.

With the `yyerrok;` action, the parser can be persuaded to feel that it has accepted enough terminal symbols, and thus to report errors in close proximity to one another.

There is a drawback, though: if `yyerrok;` is attached as an action to a formulation consisting only of error, `yyparse()` immediately believes that enough terminal symbols have been shifted, and thus can never discard an erroneous input symbol!

A more sensible example for the `yyerrok;` action is the following extension to our grammar:

```
expression
        : expression '-' expression
          IDENTIFIER
                  yyerrok; }
        | error
```

Once we have seen an IDENTIFIER following an error, it is reasonable to assume we are back on the right track, and thus to request to be informed of subsequent errors. This extension will produce two error messages for our example:

```
[error 1] line 1 near '-': expecting: IDENTIFIER
[error 2] line 1 near      expecting: '-'
yyparse == 0
```

The error symbol and the `yyerrok;` action are the *yacc* features to use in making a parser robust. The tricky problem is to employ these basic tools judiciously.

4.3 Adding the "error" symbols

The placement of error symbols is guided by the following, conflicting goals:

- as close as possible to the start symbol of the grammar.

  This way there is always a point to recover from, since there should always be a state very low on the stack in which error can be accepted. The parser then is never able to clear its stack early, i.e., to not complete by recognizing the end of file from the lexical analyzer.

- as close as possible to each terminal symbol.

  This way only a small amount of input would be skipped on each error. This can be improved using `yyerrok;` actions.

- without introducing conflicts.

  This may be quite difficult. In fact, accepting shift/reduce conflicts is reasonable as long as they serve to lengthen strings. E.g., one can continue parsing an expression past an error, rather then accepting the same error at the statement

level, thus trashing the rest of the expression.

Following these goals, we recommend the following typical positions for error symbols:

- into each recursive construct, i.e., into each repetition.
- preferably not at the end of a formulation.

This should result in a robust recovery, i.e., in a recovery from which the continuation is meaningful. Adding a trailing error and `yyerrok;` action may lead to cascading error messages, or even to loops if the parser cannot discard input.

- non-empty lists require two error variants, one for a problem at the beginning of the list, and one for a problem at the current end of the list.
- possibly empty lists require an `error` symbol in the empty branch.

If this proves impossible, add the symbol to the places where the possibly empty list is being used.

The following table is our recommendation for the placement of error symbols in the most frequent repetitive constructs[1]:

construct	EBNF	*yacc* input

optional sequence	`x: y`	`x: /* null */`
		`  x  yyerrok; }`
		`I x error`

sequence	`x: y{ y }`	`x:`
		`I x y { yyerrok }`
		`I error`
		`x error`

list	`x: y { T y }`	`x:y`
		`I x T y { yyerrok; }`
		`I error`
		`I x error`
		`x error y { yyerrok; }`
		`x T error`

We will demonstrate the three cases in turn. In each case, we use the lexical analyzer constructed for the desk calculator in section 3.5. Error recovery for the optional sequence can be studied using the following input for *yacc*:

[1] This way of extending repetitive constructs has a drawback due to a bug in *yacc* (as distributed with Bell version 7, Berkeley 4.2bsd, and various derivatives): if in a state the default action is to reduce, and if the next terminal symbol cannot be shifted but error could be (e.g., on a trailing error in a rule), *yacc's* tables dictate that the reduction take place, even if the next terminal symbol cannot be shifted subsequently. In this case error recovery takes place "too late", and the parser can, in fact, go into a loop, mistakenly reduce rules several times, etc. The 4.1bsd distribution actually contains a correction for this bug, based on [Gra79]. Essentially, in these cases all possible inputs must be enumerated, so that the error can be detected; this results in slightly larger parser tables. The correction in 4.1bsd contains a typographical error, however. A definite correction is available from the authors (S. Johnson, personal communication, 1982).

```
%{
#include <stdio.h>
#define put(x)  printf("%d ", x)
#define err(x)  fputs("err x ", stdout)
%}
%token  Constant
%%

line
        : /* empty */
        | line optional_sequence '\n'
                { putchar('\n');
                  yyerrok;
                }

optional_sequence
        : /* empty */
        | optional_sequence Constant
                { put($2);
                  yyerrok;
                }
        | optional_sequence error
                { err(1); }

%%

main(argc)
{       extern FILE * yyerfp;

        yyerfp = stderr;        /* separate listings */
        printf("yyparse() = %d\n", yyparse());
}
```

We assign stderr to our file pointer for error messages (section 3.3), so that we obtain the error messages separately from the action output. The actions are designed to exhibit the reduction behavior of this parser. The input

```
10 20

10 +
10 + 20
```

produces the output

```
10 20

10 err 1
10 err 1 20
yyparse() = 0
```

and the error messages

```
[error 1] line 3 near "+": expecting: '\n' Constant
[error 2] line 4 near "+": expecting: '\n' Constant
```

All terminal symbols are properly reduced, in spite of the input errors.

If the sequence must contain at least one element, we need to change the parser slightly:

```
sequence
        : Constant
                { put($1); }
        | sequence Constant
                { put($2);
                  yyerrok;
                }
        | error
                { err(1); }
        | sequence error
                { err(2); }
```

The same input produces

```
10 20
err 1
10 err 2
10 err 2 20
yyparse() = 0
```

and one additional error message for the empty line:

```
[error 1] line 2: expecting: Constant
[error 2] line 3 near "+": expecting: '\n' Constant
[error 3] line 4 near "+": expecting: '\n' Constant
```

Again, all terminal symbols have been properly reduced.

The list, a sequence with at least one element, and delimiters between any two elements, has more error possibilities:

```
list
        : Constant
                { put($1); }
        | list ',' Constant
                { put($3);
                  yyerrok;
                }
        | error
                { err(1); }
        | list error
                { err(2); }
        | list error Constant
                { err(3);
                  put($3);
                  yyerrok;
                }
        | list ',' error
                { err(4); }
```

The test data reflect the additional structure:

```
10 . 20

10 +
10 20
10 .
10 + 20
```

The output

```
10 20
err 1
10 err 2
10 err 3 20
10 err 4
10 err 2
yyparse() = 0
```

and the error messages

```
[error 1] line 2: expecting: Constant
[error 2] line 3 near "+": expecting: '\n' ','
[error 3] line 4 near "20": expecting: '\n' ','
[error 4] line 5: expecting: Constant
[error 5] line 6 near "+": expecting: '\n' ','
```

demonstrate that we are able to recover in all cases. Unfortunately, the case

```
10 + 20
```

is recovered through the rule

```
list : list error
```

and the second element of the list is discarded! If we eliminate this formulation, however, recognition does not terminate properly in the case of a trailing error.

Our recommendations for the placement of error symbols do not guarantee that a useful input symbol is not ignored in some error situations. Actual use, however, has convinced us that these recommendations lead to very robust parsers for common language constructs in a systematic fashion.

4.4 Adding the "yyerrok" actions

`yyerrok;` should be placed following terminal symbols at all points at which a formulation can end in error and is then followed by a reasonably significant terminal symbol. The repetitive constructs described above have already included the relevant actions.

This way, once the terminal symbol is reduced, any subsequent error would again be reported — the three-symbol-rule notwithstanding.

In effect, some symbols become rather important, in *sampleC* for example

```
sc      ;
rp
rr      }
```

```
40: shift/reduce conflict (shift 46, red'n 39) on error
state 40
        compound_statement :  { declarations_statements rr
        declarations :  declarations_declaration
        declarations :  declarations_error
        statements : _     (39)

        error  shift 46
        Identifier  reduce 39
                ...
        .  error

        declaration  goto 45
        statements  goto 44

53: reduce/reduce conflict (red'ns 41 and 56 ) on error
53: reduce/reduce conflict (red'ns 41 and 56 ) on ;
state 53
        statements :  statements error_     (41)
        expression :  error_     (56)

        ,  reduce 56
        .  reduce 41

83: shift/reduce conflict (shift 115, red'n 49) on ELSE
state 83
        statement :  if_prefix statement_     (49)
        statement :  if_prefix statement_ELSE statement

        ELSE  shift 115
        .  reduce 49
```

```
expression
        : error
```

by

```
expression
        : error ',' binary
        | error binary
```

the reduce/reduce conflicts disappear as planned, but we have five new shift/reduce conflicts. *y.output* suggests why:

```
53: shift/reduce conflict (shift 66, red'n 41) on Identifier
53: shift/reduce conflict (shift 67, red'n 41) on Constant
53: shift/reduce conflict (shift 68, red'n 41) on (
53: shift/reduce conflict (shift 69, red'n 41) on PP
53: shift/reduce conflict (shift 70, red'n 41) on MM
state 53
        statements :  statements error_    (41)
        expression :  error_, binary
        expression :  error_binary

        Identifier  shift 66
        Constant  shift 67
        (  shift 68
        PP  shift 69
        MM  shift 70
        ,  shift 75
        .  reduce 41

        binary  goto 76
```

certain spots:

```
if_prefix
        : IF error

loop_prefix
        : WHILE error

binary
        : '(' error rp
```

```
/*
 *      sample c
 *      syntax analysis with error recovery
 *      (s/r conflicts: one on ELSE, one on error)
 */

%{
#define ERROR(x) yywhere(), puts(x)
%}

/*
 *      terminal symbols
 */

/*
 *      precedence table
 */

%%

program

definitions
        : definition
        | definitions definition
                { yyerrok; }
        | error
                { ERROR("definitions: error"); }
        | definitions error
                { ERROR("definitions: definitions error"); }

definition

function_definition
        : Identifier '(' optional_parameter_list rp
          parameter_declarations compound_statement

optional_parameter_list

parameter_list
        : Identifier
        | parameter_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("parm_list: error"); }
        | parameter_list error
                { ERROR("parm_list: parm_list error"); }
        | parameter_list error Identifier
                { ERROR("parm_list: parm_list error Id");
                  yyerrok;
                }
        | parameter_list ',' error
                { ERROR("parm_list: parm_list ',' error"); }

parameter_declarations
        : /* null */
```

```
        | parameter_declarations parameter_declaration
                { yyerrok; }
        | parameter_declarations error
                { ERROR("parm_decls: parm_decls error"); }

parameter_declaration
        : INT parameter_declarator_list sc

parameter_declarator_list
        : Identifier
        | parameter_declarator_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("parm_decl_1: error"); }
        | parameter_declarator_list error
                { ERROR("parm_decl_1: parm_decl_1 error"); }
        | parameter_declarator_list error Identifier
                { ERROR("parm_decl_1: parm_decl_1 error Id");
                  yyerrok;
                }
        | parameter_declarator_list ',' error
                { ERROR("parm_decl_1: parm_decl_1 ',' error"); }

compound_statement
        : '{' declarations statements rr

declarations
        : /* null */
        | declarations declaration
                { yyerrok; }
        | declarations error
                { ERROR("declarations: declarations error"); }

declaration
        : INT declarator_list sc

declarator_list
        : Identifier
        | declarator_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("decl_list: error"); }
        | declarator_list error
                { ERROR("decl_list: decl_list error"); }
        | declarator_list error Identifier
                { ERROR("decl_list: decl_list error Id");
                  yyerrok;
                }
        | declarator_list ',' error
                { ERROR("decl_list: decl_list ',' error"); }

statements
        : /* null */
        | statements statement
                { yyerrok; }
        | statements error
```

```
                { ERROR("statements: statements error"); }

statement
        : expression sc
        | sc
        | BREAK sc
        | CONTINUE sc
        | RETURN sc
        | RETURN expression sc
        | compound_statement
        | if_prefix statement
        | if_prefix statement ELSE statement
        | loop_prefix statement

if_prefix
        : IF '(' expression rp
        | IF error
                { ERROR("if_prefix: IF error"); }

loop_prefix
        : WHILE '(' expression rp
        | WHILE error
                { ERROR("loop_prefix: WHILE error"); }

expression
        : binary
        | expression ',' binary
                { yyerrok; }
        | error ',' binary
                { ERROR("expression: error ',' binary");
                  yyerrok;
                }
        | expression error
                { ERROR("expression: expression error"); }
        | expression ',' error
                { ERROR("expression: expression ',' error"); }

binary
        : Identifier
        | Constant
        | '(' expression rp
        | '(' error rp
                { ERROR("binary: '(' error ')'"); }
        | Identifier '(' optional_argument_list rp
        | PP Identifier
        | MM Identifier
        | binary '+' binary
        | binary '-' binary
        | binary '*' binary
        | binary '/' binary
        | binary '%' binary
        | binary '>' binary
        | binary '<' binary
        | binary GE binary
        | binary LE binary
        | binary EQ binary
```

```
        | binary NE binary
        | binary '&' binary
        | binary '^' binary
        | binary '|' binary
        | Identifier '=' binary
        | Identifier PE binary
        | Identifier ME binary
        | Identifier TE binary
        | Identifier DE binary
        | Identifier RE binary

optional_argument_list

argument_list
        : binary
        | argument_list ',' binary
                { yyerrok; }
        | error
                { ERROR("arg_list: error"); }
        | argument_list error
                { ERROR("arg_list: arg_list error"); }
        | argument_list ',' error
                { ERROR("arg_list: arg_list ',' error"); }


/*
 *      make certain terminal symbols very important
 */

rp      : ')'   { yyerrok; }
sc      : ';'   { yyerrok; }
rr      : '}'   { yyerrok; }
```

The following input file exercises the error recovery mechanisms introduced in the grammar:

```
f1() { }
char x;         /* 2: bad definition */
char y;         /* this one is swallowed -- no yyerrok */

f2() { }        /* this is parsed again */

f3(a,
        int,    /* 7: bad parameter */
        c) { }

f4(int          /* 10: bad parameter */
        ) { }

f5(a, b)
        int a;
        while ; /* 15: bad declaration */
        int b;
        { }

int a,
        while,  /* 20: bad declarator */
```

```
        b;

f6() {
        break   /* 24: bad statement */
        break;
        return;
        }

f7() {
        a,
        int,    /* 31: bad expression */
        b;
        }

f8() {
        f7(a,
        int,    /* 37: bad argument */
        b);
        }
```

We obtain the following result:

```
[error 1] line 2 near "x": expecting: '('
line 2 near "x": definitions: definitions error
line 2 near ";": definitions: definitions error
line 3 near "y": definitions: definitions error
line 3 near ";": definitions: definitions error
[error 2] line 8 near "int": expecting: Identifier
line 8 near "int": parm_list: parm_list ',' error
[error 3] line 11 near "int": expecting: Identifier
line 11 near "int": parm_list: error
[error 4] line 16 near "while": expecting: '{' INT
line 16 near "while": parm_decls: parm_decls error
[error 5] line 21 near "while": expecting: Identifier
line 21 near "while": decl_list: decl_list ',' error
[error 6] line 26 near "break": expecting: ';'
line 26 near "break": statements: statements error
[error 7] line 32 near "int": expecting: '(' Ident. Const. PP MM
line 32 near "int": expression: expression ',' error
[error 8] line 38 near "int": expecting: '(' Ident. Const. PP MM
line 38 near "int": arg_list: arg_list ',' error
```

The output shows that some clustered errors are not reported individually.

4.6 Problems

1. Generate an input file that can be used with the version of *sampleC* in section 4.5 to exercise every formulation involving error.

2. In the example, several errors in definitions are reported as one. How can this be improved? Try your solution with an input file containing some suitable errors.

3. Add error recovery to the desk calculator example in section 3.5, or to the one you developed for problem 1 in section 3.8. Hint: at the `line` level, the basic technique has been demonstrated in section 4.3. Adding error recovery to the `expression` level is a more challenging problem.

4. Integrate the techniques used for expression in problem 3 into the error recovery for ***sample C.***

5. Change the desk calculator of problem 3 so that after any error, it will prompt the user for a corrected input line. Hint: special care must be taken in the placement of `yyerrok;` statements, since it would clearly be unacceptable for the parser to discard tokens from the re-entered line as part of its response to errors in the original line.

8. Add error recovery features to the grammar produced for a Pascal subset in problem **1** of section 1.8. Test your grammar with an input file containing a suitable selection of erroneous Pascal code.

Chapter 5
Semantic Restrictions

We now turn from the general problem of robust language recognition to the more specific problem of analyzing a program text in order to produce a translated, executable version of it. The word program will therefore be used in place of ***sentence,*** i.e., it is defined as a sequence of terminal symbols, for which a unique parse tree with respect to a grammar can be built. This chapter discusses how we impose additional restrictions on a program, thus completing the ***analysis*** part of a compiler; the following chapters describe the synthesis of an executable version of the algorithm described by a program.

5.1 The problem

A program can be syntactically correct and still contain semantic errors. Some typical examples are the following:

In Pascal, labels are digit strings which must be declared in a `label` declaration before they can be used. While the lexical analyzer might return such a digit string as IntegerConstant, if it is used in a goto statement, a compiler has to verify that it is a declared label.

In Pascal, labels must be declared before they are used. In C a label is an Identifier; **if it is** newly introduced following `goto`, it is implicitly declared to be a label. In almost all languages, labels can be ***defined after*** they have been used. For all labels, the compiler must verify that they have, in fact, been defined.

Labels are just one — sticky — example of ***scope*** problems: a user-defined object is only known within a particular area of the program text, known as the ***scope*** of a name. In Basic, the scope of a variable name is the entire `program` text (with the exception of user-defined function parameter names). In Fortran, variable names are known only within a program unit, i.e., a function or subroutine; program unit names and common area names are known throughout all modules which are bound into an ***image,*** a file which may be executed. In Pascal and other Algol-like languages, user-defined names are known within a ***block,*** Le., a syntactically delimited area of the program text which contains the definition for the name; blocks can be nested, and the definition of a name in an outer block can be hidden for the extent of an interior block by a new definition for the name in the interior block. C combines Algol block structure and the module concept of Fortran: compound statements are blocks which can be nested, can contain declarations, and limit their scope; function names are known globally and need not necessarily be declared before use. It is the compiler's responsibility to monitor the correct use of user-defined names within their respective scopes, as well as to generate code providing appropriate access to the various objects.

Names cannot in general be declared twice in the same context, e.g., two parameters may not have the same name, two local variables in the same block must use different names, two components of the same struct, union, or record construct must differ. While C **permits** `struct` and other names to be identical, some versions of C require component names to be distinct even for different structures.

Declarations in a program convey the intended use of a name to the compiler. Once the use has been agreed upon, abuse must be prevented. Consider:

In most languages — with the notable exception of PL/I, or deliberately lenient tools like *awk* — strings and numerical values cannot be combined, e.g., for addition. In Pascal, ***mixed mode*** expressions, i.e., combinations of real and integer values, are permitted for most operators, but certain restrictions apply: `div` expresses integer division only, / delivers a real result even for two integer operands, := permits assignment from integer to real but not conversely, etc.

Operators change their precise meaning based on the types of their operands. In some dialects of Basic, + denotes addition for numerical values and concatenation for strings. In Pascal, + denotes addition for numerical values and union for `set` values — at least in common representations of the language. In C, + can describe involved address manipulations if it combines a pointer and an integer value. Numerically, and as a machine instruction, + is quite a different operation between integer or between floating point values: the result of the integer operation is independent of the order of its operands and of implicitly placed parentheses, whereas the floating point result can critically depend on it.

Component selection in struct, union, or record constructs requires in general that the selector name belong to the structure of the variable from which the selection is to be made, i.e., operators like . , and -> have rather strict requirements for the types of their operands. C is — intentionally — rather permissive in this respect.

Enumerating a fixed, maximum number of identical phrases is a cumbersome technique in BNF. It also cannot handle some features usually found in programming languages. Consider:

Basic arrays normally may have one or two dimensions. Some versions of Fortran limit arrays to seven dimensions. A compiler must limit the number of indexing expressions in general, and it must verify for each specific array reference that the correct, individual number of indices is used. In Pascal or C, arbitrarily many dimensions can be defined; however, the number of indices used determines the data type of the reference.

A similar problem arises with function parameters. Number and types of the arguments are predefined for built-in functions, and follow from the definition for user-defined functions. A Pascal compiler must at least verify that argument values and parameters fit together; C is quite permissive in this respect. A PL/I compiler is even responsible for argument conversion.

Parameter passing poses another problem: if, as in Pascal, a subprogram may indicate a desire to modify some of its arguments, care must be taken to insure that only suitable arguments are handed down. In Fortran, all parameters can be modified, but only certain arguments (*l-values* in the sense of C) will be changed as a consequence — this is a code generation problem.

Most semantic restrictions deal with user-defined objects, i.e., constants, types, variables, subprograms and labels. We will need a symbol table, into which all information from declarations and definitions is entered, and which is consulted whenever a user-defined name is referenced.

Some semantic restrictions, however, deal with problems which defy a simple syntactic resolution. Consider:

In Pascal or C, all constants in the context of `case` must be distinct. In C, `case` and `default` labels must be positioned within a statement dependent on a switch clause; interestingly enough, this dependent statement need not even be a compound statement!

Similarly, the `break` and continue statements of C must be placed in a context from which the desired escape makes sense.

Restrictions like these require a certain amount of local testing associated with particular constructs. Our implementation of *sampleC* will demonstrate how one can check `break` and continue by means of a separate stack; in general a certain amount of ingenuity is required, since these problems do not fit a uniform framework.

5.2 Symbol table principles

A symbol table is the central place in which the compiler keeps all information associated with user-defined names and constants. While the design of a symbol table entry depends on the information required by the compiler and obtainable from the declarations in a program, the organization of the entries for searching reflects the scope rules of the language:

Basic, for example, can be handled with a table to which each new name is simply added. All names are globally known; thus the table never needs to be pruned unless the entire information about a program is erased. Parameters for user-defined functions are only known within the function; they can be entered into a second table, which is erased as soon as work on the function has been completed.

Fortran essentially requires two tables: one table contains information about all identifiers introduced within a subprogram, while a second table might be used to store subprogram names. The first table would be erased after compilation of each subprogram unit. The second table is not really required if subprograms are combined with a linker; in this case the subprogram names would be reported to the linker.

Pascal has a strict ***declare before use*** rule and nested scopes. The nested scopes are reflected by using a stack as a symbol table: new names are pushed on top of the stack, and the stack is appropriately popped once the end of a scope, i.e., the end of a function or procedure definition, is reached. Whenever a name is referenced, we can search the stack top-down and thus locate the innermost definition for the name.

Other members of the Algol family do not necessarily require that names be declared before use, as long as a declaration is present ***within*** the scope of the name. This situation is somewhat involved: we need two passes over a program, the first one to collect all declarations and to essentially propagate them to the beginning of their scope, and the second one to then deal with references based on the information collected in the first pass.

C permits nested scopes for variables, but functions cannot be nested. There is a ***declare before use*** rule, except that functions with `int` result need not be

declared. In general, a stack of names will do, as long as we keep functions in a global table, even if references to them are discovered locally.

What information is stored in a symbol table entry? For searching purposes, the entry must have access to the user-defined name; for usage verification we must represent the type of the object; and during code generation we will need to store information about the representation of the object — location on a stack, offset or absolute address, length, etc. If the symbol table is organized as a stack, the entries will be linked; if the stack is popped, we need to remember at which scope nesting level the entry was defined.

Representing the type of an object might be difficult. In Fortran there is only a small number of types, and additionally the object can be dimensioned as an array; this can be represented with a few integer values in the symbol table. In languages such as Pascal or C, with a rich set of data type constructors, a recursive description will have to be built, which will usually involve pointers to further symbol table entries.

Name searching is another area where a number of different techniques are available. In general, when the lexical analysis function has assembled a user-defined name or literal constant, it will immediately locate it in the symbol table, or enter it there if it is as yet unknown. From then on, a pointer to the symbol table entry is passed along providing access to information about the symbol, and eliminating the need to *search* the symbol table more then once. The initial search is thus performed by a routine which is called only from the lexical analyzer; in order to speed up this search, data structures such as hash tables might be used in addition to the symbol table itself. Since a lexical analyzer spends a significant amount of processing time on the name search, a lot of literature is available on the subject of table searching; for starters consult, e.g., chapter 3.D. by W. McKeeman in [Bau76].

5.3 Example

For *sampleC* we stick with a symbol table stack, represented as a linear list of entries. In order to keep things as simple as possible, we will not use additional data structures to speed up searching — this is left as an exercise. We manage the entries dynamically using the `calloc()` and `cfree()` library routines.

We will search the symbol table stack backwards from newest to oldest symbol. In this fashion, the innermost declaration of a name will be found first, provided that we pop local entries off the stack once we leave a compound statement.

We therefore need to know where the local declarations of each open block begin. This could be done by a stack of open block descriptors, from which a linked list connects the relevant symbol table entries. To simplify, at the expense of some processing time during block closure, we maintain a global counter of nesting depth of compound statements, and copy the current value of this counter into each symbol table entry when a declaration for the symbol is performed. Local entries on the symbol table stack then are precisely those which were marked with the current nesting depth. Defective programs may result in some symbols never being declared; they are marked with an initialization value for the depth field and are also removed at block closure.

One complication arises from the fact that functions need not be defined before they are used. However, functions may not be nested, so this problem can be solved by moving function descriptors in the symbol table to the outermost block. In our dynamically linked scheme, this is quite easy to accomplish: we simply relink a function descriptor at the bottom of the symbol table stack. We do need to save function descriptors, since we are building a one pass, load-and-go compiler, which does not employ a linker for image assembly, and which therefore must itself check that all referenced functions have actually been defined.

Another, smaller complication is the fact that in C parameters need not be explicitly declared: once their names have been mentioned in a parameter list, they become int variables by default. This can be handled by chaining the parameters in the symbol table, when they are initially found in the parameter list. Once the `parameter_declarations` have been reduced, we can follow the chain and default all remaining, undeclared parameters.

While it is not required, we will check consistent use of functions, i.e., we will at least count that they are always called with the same number of arguments.

There is one massive simplification in *sampleC*: since the language only supports an int data type, we need not worry about type incompatibilities. In general, semantic restrictions need to be enforced in this context; this is best done by computing result types as the various operations are recognized by the parser, and by passing the result types along on the *yacc* stack. The resulting analysis is bulky enough so that we decided to omit it here by not supporting additional data types.

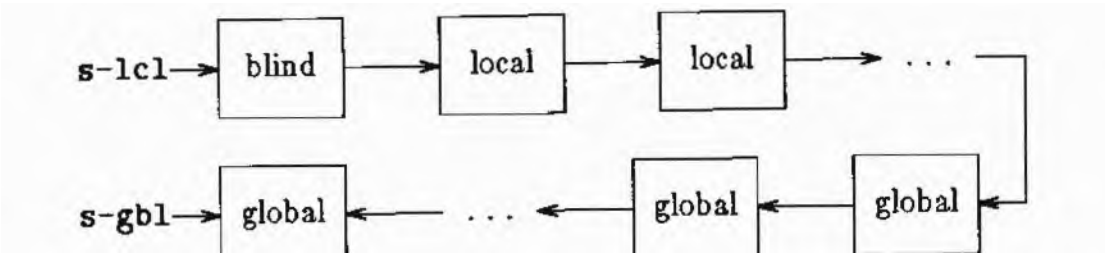
First we design a symbol table entry and define possible values for certain fields. This information is placed into *symtab.h*:

```
/*
 *      sample c -- header file for symbol table
 */

struct symtab {
        char *  s_name;         /* name pointer */
        int     s_type;         /* symbol type */
        int     s_blknum;       /* static block depth */
        union {                 /* multi-purpose */
                int s__num;
                struct symtab * s__link;
                } s__;
        int     s_offset;       /* symbol definition */
        struct symtab * s_next; /* next entry */
        };

#define s_pnum  s__.s__num      /* count of parameters */
#define NOT_SET (-1)            /* no count yet set */
#define s_plist s__.s__link     /* chain of parameters */

/*
 *      s_type values
 */

#define UDEC    0       /* not declared */
#define FUNC    1       /* function */
```

```
#define UFUNC   2       /* undefined function */
#define VAR     3       /* declared variable */
#define PARM    4       /* undeclared parameter */

/*
 *      s_type values for S_TRACE
 */

#define SYMmap  "undeclared", "function", "undefined function", \
                "variable", "parameter"

/*
 *      typed functions, symbol table module
 */

struct symtab * link_parm();    /* chain parameters */
struct symtab * s_find();       /* locate symbol by name */
struct symtab * make_parm();    /* declare parameter */
struct symtab * make_var();     /* define variable */
struct symtab * make_func();    /* define function */

/*
 *      typed library functions
 */

char * strsave();               /* dynamically save a string */
char * calloc();                /* dynamically obtain memory */
```

currently,

```
/*
 *      sample c -- symbol table definition and manipulation
 */

#include "symtab.h"
#include "y.tab.h"

/*
 *      symbol table
 */

static  struct  symtab
        symtab,                 /* blind element */
        * s_gbl;                /* global end of chain */
#define s_lcl   (& symtab)      /* local end of chain */

/*
 *      block table
 */

static int blknum = 0;          /* current static block depth */
```




```
static struct symtab * s_create(name)
        register char * name;
{       register struct symtab * new_entry = (struct symtab *)
                        calloc(1, sizeof(struct symtab));

        if (new_entry)
        {       new_entry->s_next = s_lcl->s_next;
                s_lcl->s_next = new_entry;
                new_entry->s_name = strsave(name);
                new_entry->s_type = UDEC;
                new_entry->s_blknum = 0;
                new_entry->s_pnum = NOT_SET;
                return new_entry;
        }
        fatal("No more room for symbols.");
        /*NOTREACHED*/
}
```

```
static s_move(symbol)
        register struct symtab * symbol;
{       register struct symtab * ptr;

        /* find desired entry in symtab chain (bug if missing) */
        for (ptr = s_lcl; ptr->s_next != symbol; ptr = ptr->s_next)
                if (! ptr->s_next)
                        bug("s_move");

        /* unlink it from its present position */
        ptr->s_next = symbol->s_next;

        /* relink at global end of symtab */
        s_gbl->s_next = symbol;
        s_gbl = symbol;
```

```
        s_gbl >s_next = (struct symtab *) 0;
}
```

Note that the entry is only linked into a different position on the symbol table stack; the entry itself is not moved in memory, so the pointer value referencing the element and passed as an argument does not change.

We placed a blind element on top of the symbol table stack, so that we do not need to check if we are moving the current top of the stack, i.e., so that we do not need to adjust `s_lcl` as a special case.

Initially, however, `s_gbl` may not point at the blind element — if it did, we would add the first (global) definition *following* `s_gbl`, local definitions might get positioned *between* `s_gbl` and this global definition, eventually one of them might be moved to become global.. and general mayhem would result! `s_gbl` must be initialized to point to a global entry — a blind element there would split the symbol table into two halves and would thus create another special case. Fortunately, there is a useful global entry: every *sampleC* program must contain a main() function initially, we therefore open the outermost block, and initialize `s_gbl` to point to an entry for main as an undefined function:

```
init()
{
        blk_push();
        s_gbl = s_create("main");
        s_gbl >s_type    UFUNC;
}
```

`init()` must be called before the symbol table can be accessed. A call to `init()` therefore can be placed into the main() function of our compiler, prior to the call to `yyparse()`. Another, more visible solution is to call `init` 0 very early from the parser itself. This is the first action added to the parser:

```
program
        .       { init(); }
          definitions
                { blk_pop(); }
```

In this fashion, `init()` will be called before any calls to the lexical analyzer.

`init` 0 pushes the block stack:

```
blk_push()

        ++ blknum;
}
```

Every call to `blk_push()` must be balanced by a call to `blk_pop()` to pop henceforth inaccessible symbols from the symbol table stack, to discover undefined functions, etc. We will defer a discussion of `blk_pop()` until we have seen how symbols are actually entered into the symbol table.

Every user-defined name is first seen by the lexical analyzer. `yylex()` must enter every symbol into the symbol table, as long as it is not already there. We have already placed calls to a function s lookup() into the lexical analyzer for this purpose; `s_lookup()` is called with the terminal symbol representation in yytext E and with

```
s_lookup(yylex)                         /* Constant or Identifier */
        int yylex;
{       extern char yytext[];           /* text of symbol */

        switch (yylex) {
        case Constant:
                yylval.y_str = strsave(yytext);
                break;
        case Identifier:
                if (yylval.y_sym = s_find(yytext))
                        break;
                yylval.y_sym = s_create(yytext);
                break;
        default:
                bug("s_lookup");
        }
}
```

```
struct symtab * s_find(name)
        char * name;
{       register struct symtab * ptr;

        /* search symtab until match or end of symtab chain */
        for (ptr = s_lcl->s_next; ptr; ptr = ptr->s_next)
                if (! ptr->s_name)
                        bug("s_find");
                else
                        /* return ptr if names match */
                        if (strcmp(ptr->s_name, name) == 0)
                                return ptr;
        /* search fails, return NULL */
        return (struct symtab *) 0;
}
```

```
parameter_list
        : Identifier
                { $$ = link_parm($1, 0); }
        | parameter_list ',' Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | parameter_list error
        | parameter_list error Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | parameter_list ',' error
```

```
struct symtab * link_parm(symbol, next)
        register struct symtab * symbol, * next;
{
        switch (symbol->s_type) {
        case PARM:
                error("duplicate parameter %s", symbol->s_name);
                return next;
        case FUNC:
        case UFUNC:
        case VAR:
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        default:
                bug("link_parm");
        }
        symbol->s_type = PARM;
        symbol->s_blknum = blknum;
        symbol->s_plist = next;
        return symbol;
}
```

```
parameter_declarator_list
        : Identifier
                { make_parm($1); }
        | parameter_declarator_list ',' Identifier
                { make_parm($3);
                  yyerrok;
                }
        | error
        | parameter_declarator_list error
        | parameter_declarator_list error Identifier
                { make_parm($3);
                  yyerrok;
                }
        | parameter_declarator_list ',' error
```

```
struct symtab * make_parm(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
                if (symbol->s_blknum == 2)
                {       error("parameter %s declared twice",
                                symbol->s_name);
                        return symbol;
                }
        case UDEC:
        case FUNC:
        case UFUNC:
                error("%s is not a parameter", symbol->s_name);
```

```
                symbol = s_create(symbol->s_name);
        case PARM:
                break;
        default:
                bug("make_parm");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}
```

```
declarator_list
        : Identifier
                { make_var($1); }
        | declarator_list ',' Identifier
                { make_var($3);
                  yyerrok;
                }
        | error
        | declarator_list error
        | declarator_list error Identifier
                { make_var($3);
                  yyerrok;
                }
        | declarator_list ',' error
```

```
struct symtab * make_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
        case FUNC:
        case UFUNC:
                if (symbol->s_blknum == blknum
                    || symbol->s_blknum == 2 && blknum == 3)
                        error("duplicate name %s", symbol->s_name);
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                break;
        default:
                bug("make_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}
```

```
function_definition
        : Identifier '('
                { make_func($1); blk_push(); }
          optional_parameter_list rp
          declarations
                { chk_parm($1, parm_default($4)); }
          compound_statement
                { blk_pop(); }

optional_parameter_list
        : /* null */
                { $$ = 0;  /* no formal parameters */ }
        | parameter_list
                /* $$ = $1 = chain of formal parameters */
```

```
struct symtab * make_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UFUNC:
        case UDEC:
                break;
        case VAR:
                error("function name %s same as global variable",
                        symbol->s_name);
                return symbol;
        case FUNC:
                error("duplicate function definition %s",
                        symbol->s_name);
                return symbol;
        default:
                bug("make_func");
        }
        symbol->s_type = FUNC;
        symbol->s_blknum = 1;
        return symbol;
}
```

Checking or setting the parameter count is very simple:

```
chk_parm(symbol, count)
        register struct symtab * symbol;
        register int count;
{
        if (symbol->s_pnum == NOT_SET)
                symbol->s_pnum = count;
        else if ((int) symbol->s_pnum != count)
                warning("function %s should have %d argument(s)",
                        symbol->s_name, symbol->s_pnum);
}
```

```
int parm_default(symbol)
        register struct symtab * symbol;
{       register int count = 0;

        while (symbol)
        {       ++ count;
                if (symbol->s_type == PARM)
                        symbol->s_type = VAR;
                symbol = symbol->s_plist;
        }
        return count;
}
```

```
compound_statement
        : '{'
                { blk_push(); }
          declarations statements rr
                { blk_pop(); }
```

```
blk_pop()
{       register struct symtab * ptr;

        for (ptr = s_lcl->s_next;
             ptr &&
             (ptr->s_blknum >= blknum || ptr->s_blknum == 0);
             ptr = s_lcl->s_next)
        {
                if (! ptr->s_name)
                        bug("blk_pop null name");

#ifdef TRACE
                {       static char * type[] = { SYMmap };

                        message("Popping %s: %s, depth %d, offset %d",
                                ptr->s_name, type[ptr->s_type],
                                ptr->s_blknum, ptr->s_offset);

                }
#endif TRACE
                if (ptr->s_type == UFUNC)
                        error("undefined function %s",
                                ptr->s_name);
                cfree(ptr->s_name);
                s_lcl->s_next = ptr->s_next;
```

```
                cfree(ptr);
        }
        -- blknum;
}
```

```
binary
        : Identifier
                { chk_var($1); }
        | PP Identifier
                { chk_var($2); }
        | Identifier '=' binary
                { chk_var($1); }
```

```
chk_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                error("undeclared variable %s", symbol->s_name);
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                break;
        case FUNC:
        case UFUNC:
                error("function %s used as variable",
                        symbol->s_name);
        case VAR:
                return;
        default:
                bug("check_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
```

```
binary
        : Identifier '('
                { chk_func($1); }
          optional_argument_list rp
                { chk_parm($1,$4); }

optional_argument_list
        : /* null */
                { $$ = 0;  /* # of actual arguments */ }
        | argument_list
                /* $$ = $1 = # of actual arguments */

argument_list
        : binary
                { $$ = 1; }
        | argument_list ',' binary
                { ++ $$;
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | argument_list error
        | argument_list ',' error
```

Here is chk_func(), the last of the symbol table utilities:

```
chk_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                symbol->s_pnum = NOT_SET;
                return;
        case VAR:
                error("variable %s used as function",
                        symbol->s_name);
                symbol->s_pnum = NOT_SET;
        case UFUNC:
        case FUNC:
                return;
```

```
        default:
                bug("check_func");
        }
        s_move(symbol);
        symbol->s_type = UFUNC;
        symbol->s_blknum = 1;
}
```

```
main(a,b) int a,b;
{       a=b;
        {a;b;}
        if (a==b) {a;b;}
        if (a==b+1) a; else b;
        while (a==b) {a; break;}
        return;
}
int f() { int x; int y; return x+y; }
f(a,b)                  /* 14: duplicate function definition */
        int a;          /* undeclared parm b (not an error) */
{       int b,c;        /* 16: symbol b duplicates parm b */
        int c,d;        /* 17: duplicate symbol c */
        e=b;            /* 18: undeclared e */
        f=c+d;          /* 19: use func for var */
        a(b);           /* 20: use var for func */
        g();            /* 21: undefined function g */
}
h(a,a)                  /* 23: duplicate parameter */
        int f;          /* 24: not a parameter */
        int a;
        int a;          /* 26: duplicate parameter */
{ }
```

Syntax and semantic analysis of this program result in the following messages (assuming TRACE is defined):

```
line 12 near "}": Popping b: variable, depth 2, offset 0
line 12 near "}": Popping a: variable, depth 2, offset 0
line 13 near "}": Popping y: variable, depth 3, offset 0
line 13 near "}": Popping x: variable, depth 3, offset 0
[error 1] line 14 near "(": duplicate function definition f
[warning] line 16 near "{": function f should have 0 argument(s)
[error 2] line 16 near "b": duplicate name b
[error 3] line 17 near "c": duplicate name c
[error 4] line 18 near ";": undeclared variable e
[error 5] line 19 near ";": function f used as variable
[error 6] line 20 near "(": variable a used as function
line 22 near "}": Popping e: variable, depth 3, offset 0
line 22 near "}": Popping d: variable, depth 3, offset 0
line 22 near "}": Popping c: variable, depth 3, offset 0
line 22 near "}": Popping c: variable, depth 3, offset 0
line 22 near "}": Popping b: variable, depth 3, offset 0
line 22 near "}": Popping b: variable, depth 2, offset 0
line 22 near "}": Popping a: variable, depth 2, offset 0
[error 7] line 23 near "a": duplicate parameter a
[error 8] line 24 near "f": f is not a parameter
[error 9] line 26 near "a": parameter a declared twice
line 27 near "}": Popping f: variable, depth 2, offset 0
line 27 near "}": Popping a: variable, depth 2, offset 0
line 27: Popping h: function, depth 1, offset 0
line 27: Popping f: function, depth 1, offset 0
line 27: Popping main: function, depth 1, offset 0
line 27: Popping g: undefined function, depth 1, offset 0
[error 10] line 27: undefined function g
```

5.4 Typing the value stack

While implementing the symbol table facilities and semantic checks, we have made heavy use of the value stack: for Identifier terminal symbols, we passed symbol table pointers from the lexical analyzer to the parser; for constant terminal symbols, we passed pointers to dynamically acquired string storage; we chained the parameter list using symbol table pointers; and we counted up the number of expressions in the argument_list on the value stack as well. This last use of stack elements unfortunately necessitates a union of types for the value stack. We count using an int variable, but we point to the symbol table using a pointer data type. While pointers in C can be cast as pointers to any data type, it is still a good idea to employ different data type specifications when pointing to strings and to symbol table elements.

Even as a union, the value stack could still be typed as described in section 3.6. However, once we refer to stack elements using the $i syntax within actions, we need to inform *yacc* just what component of the union should be referenced in each case. To put it differently, we must associate a data type syntactically represented as a union component with all those symbols presented to *yacc* which we reference through Si or $$. This of course requires certain extensions of the *yacc* specification syntax described up to now.

We prefer to "type" our grammar in a separate editing pass following construction of all the actions. During this pass we need to note all terminal and non-terminal symbols which are referenced on the value stack and decide on a data type for the corresponding value stack element. **If** we rely on the default action

```
$$ = $1;
```

to actually pass a value, we need to consider the associated symbols even if the default action is not explicitly specified. (This is one reason why we usually comment those points in a *yacc* specification in which we rely on the default action.)

Once all the necessary data types are known together with those symbols which need to be typed, we can proceed to modify the *yacc* specification. We will describe the modification using the *sampleC* specification as a concrete example.

First we must define the data type of value stack elements. This is done in the first part of the *yacc* specification using a union declaration in the style of C, prefixed by a % character[1]. In our case, value stack elements can be pointers to the symbol table, pointers to character strings, and integer values for counting. We define:

```
%union {
        struct symtab * y_sym;   /* Identifier */
        char * y_str;            /* Constant */
        int y_num;               /* count */
```

[1] There are other methods to define the data type, but we believe this technique to be both visible in the yacc specification, and convenient, since the resulting union declaration as well as an extern declaration for yylval are automatically placed into the file *y.tab.h*.

Next we type those terminal symbols for which during lexical analysis a value is assigned to yylval. Syntactically, this is achieved by placing the name of a union component, enclosed in angle brackets, between %token and the list of terminal names to be so typed. In our case, Identifier and Constant have corresponding values on the stack:

```
%token <y_sym> Identifier
%token <y_str> Constant
```

The values are assigned to yylval by the routine s_lookup(). These assignments must, of course, also use union components — this has already been tacitly done correctly in section 5.3.

Finally we must type all those non-terminal symbols for which $1 or $$ are referenced. This is accomplished by making a %type definition in a manner very similar to a %token definition for terminal symbols. We must type argument_list and optional_argument_list for counting purposes, and parameter_list and optional_parameter_list to pass the parameter chain header:

```
%type  <y_sym> optional_parameter_list, parameter list
%type  <y_num> optional_argument_list, argument_list
```

It should be noted that as soon as %union, %type, or the < > syntax is used, *yacc* very strictly checks that *all* references to the value stack are typed appropriately. Any omission immediately causes *yacc* to terminate with a fatal error indicating the offending line in the specification.

One subtle typing facility, required especially for anonymous non-terminal symbols (see section 3.7), has not yet been discussed. It will be shown when it is required in section 6.2.

5.5 Problems

1. Write a *sampleC* test program containing deliberate errors to provoke all semantic error messages in the compiler.

2. Change the method of storing symbols **for** *sampleC* to use a hash scheme, to speed up the search for names. Demonstrate, e.g., using the C profiling option, that there is a gain in efficiency.

3. Add to *sampleC* a block descriptor stack, which can essentially be maintained as part of the *yacc* value stack. What changes will this require in typing of the value stack? Again, try to measure the gain in efficiency.

4. Remove those parts of symbol table management which will be unnecessary if a linking loader is used. Decide how (and when) to pass information to the linker.

5. Extend the desk calculator with simple string operations. Use the operator + to denote both addition and string concatenation. Should the resulting problem be handled in the grammar, or by separate semantic routines? I.e., should the grammar know Constant, or rather Number and String?

Chapter
Memory Allocation

We are now ready to define an *implementation* of the *sampleC* language for a particular machine. We must develop policies for *memory allocation* and *code generation.*

It is interesting to note that the entire problem of language recognition could be solved without any knowledge of the target machine. This serves to emphasize that a significant amount of the code of a compiler can be completely target- and host-machine independent.

In this and the following chapters, we will emphasize the principles rather than attempt to cope with the peculiarities of a particular machine. We will therefore describe the implementation for a fictitious machine which is adapted to the requirements of the *sampleC* source language.

6.1 Principles

The memory allocation policy defines the representation of variables, i.e., the implementation of declarations. We must decide how much memory to allocate to an object of each data type, and how to address the object, i.e., where to place it during program execution, to support particular life expectancies.

Run time memory assignment tends to mirror symbol table organization to some extent. Consider:

In Basic, there are only global variables, kept in a global symbol table, which for an interpreter might as well also hold the values of the variables during execution. The parameters of user-defined functions can be handled with the same stack which is normally used **for** expression evaluation.

In Fortran, subprograms cannot be called recursively. Since the values of local variables must be preserved between successive calls to the same function, we must assign a unique memory cell for each variable in each subprogram.

In C, Pascal, and other Algol-like languages, subprograms can be called recursively; hence, we must dynamically allocate space for the local variables of a subprogram as it is called. Since local variables normally cease to exist once a subprogram terminates, we can free and reuse their space. Function invocation is a stack discipline, and this discipline must be employed for managing the local variables as well.

The nesting of scopes in Algol-like languages poses an additional problem. If functions may be nested during definition, they have access by name only to **a** selective subset of the local variables of the currently activated functions.

The nesting of compound statements, with the associated rules for the life expectancy of variables declared within **a** compound statement, leads to a reuse of memory which *can* be managed at compile time. Compound statements can be viewed as anonymous subprograms which can only he entered sequentially, i.e., in the same order as the compiler sees them. Since compound statements are anonymous, they can themselves not be called recursively; thus, the compiler has full information about the behavior of the life expectancy of local variables defined in a compound statement.

```
        symbol = make_var(symbol);

        /* if not in parameter region, assign suitable offset */
        switch (symbol->s_blknum) {
        default:                        /* local region */
                symbol->s_offset = l_offset++;
        case 2:                         /* parameter region */
                break;
        case 1:                         /* global region */
                symbol->s_offset = g_offset++;
                break;
        case 0:
                bug("all_var");
        }
```

```
        { if (l_offset > l_max)
              l_max = l_offset;
          l_offset = $<y_lab>2;
          blk_pop();
        }
```

```
%union  {
        struct symtab * y_sym;  /* Identifier */
        char * y_str;           /* Constant */
        int y_num;              /* count */
        int y_lab;              /* label */
        }
```

```
function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                }
          compound_statement
                { all_func($1); }
```

Once processing of a function is complete, `l_max` contains the required value:

```
all_func(symbol)
        struct symtab * symbol;
{
        blk_pop();

#ifdef  TRACE
        message("local region has %d word(s)", l_max);
#endif
}
```

```
| parameter_list error Identifier
        { $$ = link_parm($3, $1);
          yyerrok;
        }
| parameter_list ',' error
```

```
        f=c+d;          /* 19: use func for var */
        a(b);           /* 20: use var for func */
        g();            /* 21: undefined function g */
}
h(a,a)                  /* 23: duplicate parameter */
        int f;          /* 24: not a parameter */
        int a;
        int a;          /* 26: duplicate parameter */
{ }
```

Output from the memory allocator:

```
line 6 near "{": parameter region has 2 word(s)
line 12 near "}": Popping b: variable, depth 2, offset 1
line 12 near "}": Popping a: variable, depth 2, offset 0
line 12 near "}": local region has 0 word(s)
line 13 near "{": parameter region has 0 word(s)
line 13 near "}": Popping y: variable, depth 3, offset 1
line 13 near "}": Popping x: variable, depth 3, offset 0
line 13 near "}": local region has 2 word(s)
[error 1] line 14 near "(": duplicate function definition f
[warning] line 16 near "{": function f should have 0 argument(s)
line 16 near "{": parameter region has 2 word(s)
[error 2] line 16 near "b": duplicate name b
[error 3] line 17 near "c": duplicate name c
[error 4] line 18 near ";": undeclared variable e
[error 5] line 19 near ";": function f used as variable
[error 6] line 20 near "(": variable a used as function
line 22 near "}": Popping e: variable, depth 3, offset 0
line 22 near "}": Popping d: variable, depth 3, offset 3
line 22 near "}": Popping c: variable, depth 3, offset 2
line 22 near "}": Popping c: variable, depth 3, offset 1
line 22 near "}": Popping b: variable, depth 3, offset 0
line 22 near "}": Popping b: variable, depth 2, offset 1
line 22 near "}": Popping a: variable, depth 2, offset 0
line 22 near "}": local region has 4 word(s)
[error 7] line 23 near ")": duplicate parameter a
[error 8] line 24 near "f": f is not a parameter
[error 9] line 26 near "a": parameter a declared twice
line 27 near "{": parameter region has 1 word(s)
line 27 near "}": Popping f: variable, depth 2, offset 0
line 27 near "}": Popping a: variable, depth 2, offset 0
line 27 near "}": local region has 0 word(s)
line 27: Popping h: function, depth 1, offset 0
line 27: Popping f: function, depth 1, offset 0
line 27: Popping main: function, depth 1, offset 0
line 27: Popping g: undefined function, depth 1, offset 0
[error 10] line 27: undefined function g
line 27: global region has 1 word(s)
```

6.3 Problems

1. Suppose that the output from our compiler is to be processed by an assembler, i.e., the compiler will emit assembler code rather than machine code. What would be needed to allocate global variables by name? How would this simplify the compilation?

2. Suppose that *sampleC* is to be implemented on a machine in which one word occupies more than one addressable memory location (e.g., one word is made up of two separately-addressable bytes). What changes must be made to the memory allocation routines?

3. Many languages allow nesting of function definitions. What part of the allocation algorithm for *sampleC* would have to be changed so that function definitions could be nested? Keep in mind that a variable in an outer nesting block may be accessed as a global variable by an internally nested function (as long as there is no conflicting definition of its name in the internal function). How can the internal function reference such variables?

4. Using right recursion, it was simple to reverse the order of acceptance of the names in a parameter_ list. What happens if we write an `argument_list` in a right recursive fashion? Since right recursion is not acceptable in this case, what other technique could be used to arrange for the arguments to a function call to be pushed onto the stack in reverse order?

Now that we are able to allocate memory
to produce a compiler that will generate code
showing how to generate code for a stack ma
operations available on our machine, proceeds
tion of expression values and for assignment
lems of generating code to implement **if** a
attempt is made to optimize the generated c
the chapter suggest how some additional effici

7.1 Principles

The code generation policy determines
each action that can be expressed in the pr
approach, given a stack machine, is to arran
postfix notation. Making local and global vari
depending on the addressing structure of the t

The only difficult aspect, which for an act
the design of the calling sequence for functio
"suitable" instructions; managing the activa
require that appropriate subprograms be desi
gram. Such subprograms are a natural place
of the compiled code.

The implementation of iteration and d
number of forward branches, e.g., around
tunately, these forward branch instructions
with an unknown target address, so that sp
later with the corrected address. If we actua
need to fix the relevant memory locations.

If we decide to emit assembler text, we c
direct the assembler later to the proper prog
jump instruction once we know its actual tar
this case, however, is to simply generate a sy
with the forward branch.

The break and continue statements of
relevant label information is pushed when a w
a switch statement is included in the languag
information for continue can significantly dif
stacks aLso serve to determine the legality of
test which we had deliberately postponed in ch

Chapter 7
Code Generation

y in our fictitious machine, we are ready
e for it. Most of this chapter is devoted to
chine. Section 7.2 begins by defining the
s to the generation of code for the calcula-
statements, and then considers the prob-
nd while control structures. Very little
ode; however, the problems at the end of
ency could be achieved.

what instructions need to be issued for
ogramming language. A straightforward
ge for the expressions to be converted to
ables addressable might pose a problem,
arget machine.

tual implementation is quite important, is
n calls. For our example we will assume
tion record stack on a real machine may
gned and linked with each compiled pro-
to insert traces for debugging or profiling

lecision statements usually requires a
the else part of an **if** statement. Unfor-
s usually have to be emitted twice: once
pace for the instruction is reserved, and
ally construct a program in memory, we

could use an origin pseudo-instruction to
gram address so that we may reissue the
rget address. A much better technique in
mbolic label, and let the assembler deal

f *sampleC* require stacks onto which the
while or similar statement is processed. If
e, two stacks are required, since then the
ffer from the information for **break**. The
f the use of these statements, a semantic
hapter 5.

7.2 Example

We will demonstrate how to generate assembler code for our fictitious machine. The assembler source format is similar to that of many existing assemblers: one instruction per line; an optional label field starts in column one; the second, mandatory field contains a mnemonic operation code; a third field might contain an operation modifier or other information; fields are separated by white space; and a comment may follow at the end of the line. The details of the format are not really important at this point, since they are easily changed in the code generation routines.

As a quick overview of the available instructions, consider their definition in the following header file *gen.h*:

```
/*
 *      operation codes for pseudo machine
 */

#define OP_ALU    "alu"         /* arithmetic-logic-op  */
#define OP_DEC    "dec"         /* region,offset        */
#define OP_INC    "inc"         /* region,offset        */
#define OP_LOAD   "load"        /* region,offset        */
#define OP_STORE  "store"       /* region,offset        */
#define OP_POP    "pop"         /*                      */
#define OP_JUMPZ  "jumpz"       /* label                */
#define OP_JUMP   "jump"        /* label                */
#define OP_CALL   "call"        /* parm-count,address   */
#define OP_ENTRY  "entry"       /* local-frame-size     */
#define OP_RETURN "return"      /*                      */

/*
 *      region modifiers
 */

#define MOD_GLOBAL "gbl"        /* global region        */
#define MOD_PARAM  "par"        /* parameter region     */
#define MOD_LOCAL  "lcl"        /* local region         */
#define MOD_IMMED  "con"        /* load only: Constant  */

/*
 *      OP_ALU modifiers
 */

#define ALU_ADD "+"             /* addition             */
#define ALU_SUB "-"             /* subtraction          */
#define ALU_MUL "*"             /* multiplication       */
#define ALU_DIV "/"             /* division             */
#define ALU_MOD "%"             /* remainder            */
#define ALU_LT  "<"             /* compares as: <       */
#define ALU_GT  ">"             /*              >       */
#define ALU_LE  "<="            /*              <=      */
#define ALU_GE  ">="            /*              >=      */
#define ALU_EQ  "=="            /*              ==      */
#define ALU_NE  "!="            /*              !=      */
#define ALU_AND "&"             /* bit-wise and         */
#define ALU_OR  "|"             /* bit-wise or          */
#define ALU_XOR "^"             /* bit-wise excl. or    */
```

```
/*
        typed functions, code generator
 */

char * gen_mod(); /* region modifier */
```

The precise definition of each instruction is implied by the simulator presented in chapter 8. For the purposes of code generation, the following remarks should suffice to characterize the effect of each instruction:

Our fictitious machine is a stack machine. Code generation for expressions is therefore quite simple: the values of variables and constants must be pushed onto the stack, using load instructions provided for this purpose, and for operators an alu instruction must be issued_ The alu instruction has a modifier which indicates what arithmetic or logic operation is to be performed on the two elements on top of the stack. The result of the operation then replaces the two elements on the stack. Modifiers happen to exist corresponding to each operator in *sampleC*.

Assignments, of course, correspond to store instructions. In *sarnpleC,* however, assignment can be an embedded operation. The store instruction therefore will *not* remove a value from the stack. This is instead accomplished by an explicit pop instruction, which must be coded whenever an expression value is to be discarded.

Code generation for if and while statements involves the construction of appropriate branching instructions. There are some forward references, e.g., to the else part of an if statement; these are handled by generating unique labels based on a counter, passing them on the semantic stack in *yacc* actions, and letting the assembler resolve the definitions.

break and continue pose a more subtle problem: in *sampleC* they are allowed only inside a while loop. This is best monitored by separate stacks, on which each while statement is expected to deposit appropriate labels, which are removed at the end of the dependent statement.

Function calls are handled by a call instruction, which expects the function arguments to be on the stack. This instruction contains the number of actual arguments, so that the parameter segment can be set up properly.

At the beginning of a function we need to code an entry instruction specifying the amount of space to reserve on the stack for local variables. The combination of call and entry instructions is assumed to handle all problems associated with parameter passing and dynamic allocation of local variables.

The return instruction will remove the local activation record from the stack, and restore all relevant hardware registers. If the return instruction is preceded by the evaluation of an expression, the value must be saved prior to return, since code following call must be generated to remove all arguments from the stack. This code is also expected to push any result value of a function back onto the stack.

We will again show additions to the grammar next to the new code generation functions called by these actions. A complete listing is in section 6 in the appendix. Let us start by considering code generation for arithmetic expressions:

```
binary
        : binary '+' binary
                { gen_alu(ALU_ADD, "+"); }
```

```
gen_alu(mod, comment)
        char * mod;             /* mnemonic modifier */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t%s\t\t; %s\n", OP_ALU, mod, comment);
}
```

```
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}

gen(op, mod, val, comment)
        char * op;              /* mnemonic operation code */
        char * mod;             /* mnemonic modifier */
        int val;                /* offset field */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t%s,%d\t\t; %s\n", op, mod, val, comment);
}
```

```
%type   <y_lab> loop_prefix
%%

loop_prefix
        : WHILE '('
                { $<y_lab>$ = gen_label(new_label());
                  push_continue($<y_lab>$);
                }
          expression rp
                { $$ = $<y_lab>3; }
        | WHILE error
                { $$ = gen_label(new_label());
                  push_continue($$);
                }
```

```
statement
        : loop_prefix
                { $<y_lab>$ = gen_jump(OP_JUMPZ, new_label(),
                        "WHILE");
                  push_break($<y_lab>$);
                }
          statement
                { gen_jump(OP_JUMP, $1, "repeat WHILE");
                  gen_label($<y_lab>2);
                  pop_break();
                  pop_continue();
                }
```

Code for the dependent statement can then be generated. An unconditional branch to the continuation point follows, and then the label must be defined to which we branch when we want to leave the while construct.

The resulting code is not optimally efficient: if we moved the code for the condition to *follow* the dependent statement, we could use the conditional branch to iterate the loop and thus save one branch per iteration. This, however, would require that we save the code for an expression somewhere.

sampleC has break and continue statements, which within a while construct transfer control to the termination or iteration points. The statements must be implemented as unconditional jumps, and we must supply appropriate labels. Since while statements can be nested, the labels must be stacked; since in C a switch establishes a new nesting level for break but not for continue, we chose to implement two stacks, although *sampleC* could be implemented with one.

We have already pushed the stacks in the `loop_prefix` and in the while statement expansion; following the while construct, we have popped the labels. Assuming the existence of push() and pop() stack management functions, the routines can be

```
push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

pop_break()
{
        b_top = pop(b_top);
}

pop_continue()
{
        c_top = pop(c_top);
}
```

```
statement
        : BREAK sc
                { gen_break(); }
        | CONTINUE sc
                { gen_continue(); }

gen_break()
{
        gen_jump(OP_JUMP, top(b_top), "BREAK");
}

gen_continue()
{
        gen_jump(OP_JUMP, top(c_top), "CONTINUE");
}
```

```
static struct bc_stack {
        int bc_label;           /* label from new_label */
        struct bc_stack * bc_next;
        } * b_top,              /* head of break stack */
          * c_top;              /* head of continue stack */
```

```
static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));

        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("No more room to compile loops.");
        /*NOTREACHED*/
}
```

```
static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue stack underflow");
        /*NOTREACHED*/
}
```

```
static int top(stack)
        struct bc_stack * stack;
{
        if (! stack)
        {       error("no loop open");
                return 0;
        }
        else
                return stack->bc_label;
}
```

```
binary
        : Identifier '('
                { chk_func($1); }
```

```
        optional_argument_list rp
                { gen_call($1,$4); }
```

```
gen_call(symbol, count)
        struct symtab * symbol; /* function */
        int count;              /* # of arguments */
{
        chk_parm(symbol, count);
        printf("\t%s\t%d,%s\n", OP_CALL, count, symbol->s_name);
        while (count-- > 0)
                gen_pr(OP_POP, "pop argument");
        gen(OP_LOAD, MOD_GLOBAL, 0, "push result");
}
```

```
statement
        : RETURN sc
                { gen_pr(OP_RETURN, "RETURN"); }
        | RETURN expression sc
                { gen(OP_STORE, MOD_GLOBAL, 0, "save result");
                  gen_pr(OP_RETURN, "RETURN");
                }
```

```
function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                  $<y_lab>$ = gen_entry($1);
                }
          compound_statement
```

```
        { all_func($1);
          gen_pr(OP_RETURN, "end of function");
          fix_entry($1, $<y_lab>7);
        }
```

```
int gen_entry(symbol)
        struct symtab * symbol; /* function */
{       int label = new_label();

        printf("%s\t", symbol->s_name);
        printf("%s\t%s\n", OP_ENTRY, format_label(label));
        return label;
}
```

```
fix_entry(symbol, label)
        struct symtab * symbol; /* function */
        int label;
{       extern int l_max;       /* size of local region */

        printf("%s\tequ\t%d\t\t; %s\n", format_label(label),
                l_max, symbol->s_name);
}
```

```
program
        :       { init(); }
          definitions
                { end_program(); }

end_program()
{       extern int g_offset;    /* size of global region */

        all_program();          /* allocate global variables */
        printf("\tend\t%d,main\n", g_offset);
}
```

As an example of code generation, consider the following implementation of Euclid's algorithm to compute the greatest common divisor of two positive integers:

```
/*
 *      Euclid's algorithm
 */

main()
{       int a,b;

        a = 36;
        b = 54;

        while (a != b)
                if (a > b)
                        a -= b;
                else
                        b -= a;
}
```

The resulting assembler code is as follows:

```
main    entry   $$1
        load    con,36
        store   lcl,0           ; a
        pop                     ; clear stack
        load    con,54
        store   lcl,1           ; b
        pop                     ; clear stack
$$2     equ     *
        load    lcl,0           ; a
        load    lcl,1           ; b
        alu     !=              ; !=
        jumpz   $$3             ; WHILE
        load    lcl,0           ; a
        load    lcl,1           ; b
        alu     >               ; >
        jumpz   $$4             ; IF
        load    lcl,0           ; a
        load    lcl,1           ; b
        alu     -               ; -
        store   lcl,0           ; a
        pop                     ; clear stack
        jump    $$5             ; past ELSE
$$4     equ     *
        load    lcl,1           ; b
        load    lcl,0           ; a
        alu     -               ; -
        store   lcl,1           ; b
        pop                     ; clear stack
$$5     equ     *
        jump    $$2             ; repeat WHILE
$$3     equ     *
        return                  ; end of function
$$1     equ     2               ; main
        end     1,main
```

```
        jump    $$5             ; past ELSE
        ...
$$5     equ     *
        jump    $$2             ; repeat WHILE
```

Chapter 8
A Load-and-Go System

The preceding chapter demonstrated that it is relatively simple to construct a code generator if the target machine architecture is close enough to the source language. Once the *implementation,* Le., a memory allocation policy and code sequences for the various actions in the source language, has been defined, we can turn to simulation of a suitable machine architecture as one means of completing the programming system- This chapter will show the construction of a simulator for the machine assumed in the previous two chapters, and it will show how a load-and-go system is developed using such a simulator. The discussion is necessarily quite specific to *sample C*. However, several languages have been implemented in a similar fashion, e.g., Pascal and Mod u la-2.

8.1 A machine simulator

A simulator for our fictitious machine is actually quite simple to construct. First we need to define numerical codes for the individual instructions and instruction modifiers. The resulting header file *sim.h* provides a different representation for all names previously defined in *gen.h*:

```
/*
 *      operation codes for pseudo machine
 */

#define OP_ALU    1     /* alu   arithmetic-logic-op    */
#define OP_DEC    2     /* dec   region,offset          */
#define OP_INC    3     /* inc   region,offset          */
#define OP_LOAD   4     /* load  region,offset          */
#define OP_STORE  5     /* store region,offset          */
#define OP_POP    6     /* pop                          */
#define OP_JUMPZ  7     /* jumpz label                  */
#define OP_JUMP   8     /* jump  label                  */
#define OP_CALL   9     /* call  routine-address        */
#define OP_ENTRY  10    /* entry local-frame-size       */
#define OP_RETURN 11    /* return                       */

/*
 *      region modifiers
 */

#define MOD_GLOBAL 1    /* global region        */
#define MOD_PARAM  2    /* parameter region     */
#define MOD_LOCAL  3    /* local region         */
#define MOD_IMMED  4    /* load only: Constant  */

/*
 *      OP_ALU modifiers
 */

#define ALU_ADD 1       /* addition             */
#define ALU_SUB 2       /* subtraction          */
```

```
#define ALU_MUL 3       /* multiplication          */
#define ALU_DIV 4       /* division                */
#define ALU_MOD 5       /* remainder               */
#define ALU_LT  6       /* compares as: <          */
#define ALU_GT  7       /*              >          */
#define ALU_LE  8       /*              <=         */
#define ALU_GE  9       /*              >=         */
#define ALU_EQ  10      /*              ==         */
#define ALU_NE  11      /*              !=         */
#define ALU_AND 12      /* bit-wise and            */
#define ALU_OR  13      /* bit-wise or             */
#define ALU_XOR 14      /* bit-wise excl. or       */
```

```
/*
 *      registers
 */

static struct prog * inst;      /* -> current instruction */
#define G       0               /* global segment */
static int P;                   /* current parameter segment */
static int L;                   /* current local segment */
static int T;                   /* top of stack */
```

```
simulate(pc_limit, global, pc)
        int pc_limit, global, pc;
{
        /* initialize */

        printf("\nExecution begins...\n\n");

        for (;;)
        {       /* fetch */
                if (pc < 0 || pc >= pc_limit)
                        bug("pc not in program area");
                inst = &prog[pc++];

                /* decode operation and dispatch */
                switch (inst->p_op) {
                default:
                        printf("%d:\thalt\n", inst-prog);
                        return;

                /* other instructions */
                }
        }
}
```

```
case OP_ALU:
        if (T <= L+1)
                bug("simulator stack underflow");
        switch (inst->p_mod) {
        default:
                bug("illegal ALU instruction");
        case ALU_ADD:
                NEXT += TOP;
                break;
        case ALU_LT:
                NEXT = NEXT < TOP;
                break;
        }
        POP;
        break;
```

```
case OP_LOAD:
        if (T >= DIM(data))
                fatal("Too much data.");
        if (inst->p_mod == MOD_IMMED)
                PUSH = inst->p_val;
        else
                PUSH = MEMORY;
        break;
```

```
case OP_STORE:
        if (T <= L)
                bug("simulator stack underflow");
        printf("%d:\tstore\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, MEMORY = TOP);
        break;
```

```
case OP_INC:
        if (T >= DIM(data))
                fatal("Too much data.");
        printf("%d:\tinc\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
```

```
                inst->p_val, PUSH = ++ MEMORY);
        break;

case OP_DEC:
        if (T >= DIM(data))
                fatal("Too much data.");
        printf("%d:\tdec\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, PUSH = -- MEMORY);
        break;
```

```
case OP_CALL:
        printf("%d:\tcall\t%d\n", inst-prog,
                inst->p_val);
        PUSH = pc;
        pc = inst->p_val;
        PUSH = P;
        P = T - 2 - inst->p_mod;
        break;
```

```
        if (global >= DIM(data))
                fatal("Not enough room for global data.");
        T = global + 2;
```

```
#include "sim.h"
#include "symtab.h"

/*
 *      program memory
 */
struct prog prog[L_PROG];
static int pc = 1;              /* current program counter */
                                /* HALT (0) is at address 0 */

/*
 *      generate a single instruction
 */
```

```
int gen(op, mod, val, comment)
        int op;                 /* operation code */
        int mod;                /* modifier */
        int val;                /* offset field */
        char * comment;         /* instruction comment */
{
        if (pc >= DIM(prog))
                fatal("Not enough program memory.");
        prog[pc].p_op = op;
        prog[pc].p_mod = mod;
        prog[pc].p_val = val;
        printf("%d:\t%d\t%d,%d\t; %s\n",
                pc, op, mod, val, comment);
        return pc ++;
}
```

```
int gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}
```

```
gen_alu(mod, comment)
        int mod;                /* modifier */
        char * comment;         /* instruction comment */
{
        gen(OP_ALU, mod, 0, comment);
}

gen_li(const)
        char * const;           /* Constant value */
{
        gen(OP_LOAD, MOD_IMMED, atoi(const), const);
```

```
}

gen_pr(op, comment)
        int op;                 /* operation code */
        char * comment;         /* instruction comment */
{
        gen(op, 0, 0, comment);
}
```

```
if_prefix
        : IF '(' expression rp
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }

statement
        : if_prefix statement
                { gen_label($1); }
```

```
int gen_jump(op, label, comment)
        int op;                 /* operation code */
        int label;              /* target of jump */
        char * comment;         /* instruction comment */
{       int pc = gen(op, 0, label, comment);

        if (label <= 0)
                return -pc;     /* new head of chain */
        else
                return label;   /* already defined */
}
```

```
int new_label()
{
        return 0;               /* end of chain */
}
```

```
int gen_label(chain)
        int chain;
{       int next;

        while (chain < 0)
        {       chain = - chain;
                next = prog[chain].p_val;
                if (next > 0)
                        break;  /* already ok */
                prog[chain].p_val = pc;
                printf("%d:\t(fixup)\t%d\n", chain, pc);
                chain = next;
        }
        return pc;
}
```

```
loop_prefix
        : WHILE '('
                { $$ = gen_label(new_label());
                  push_continue($$);
                }
          expression rp
                { $$ = $3; }

statement
        : loop_prefix
                { $$ = gen_jump(OP_JUMPZ, new_label(), "WHILE");
                  push_break($$);
```

```
        }
    statement
        { gen_jump(OP_JUMP, $1, "repeat WHILE");
          gen_label($2);
          pop_break();
          pop_continue();
        }
```

```
gen_continue()
{
        *top(&c_top) = gen_jump(OP_JUMP, *top(&c_top), "CONTINUE");
}
```

```
if (next > 0)
```

```
gen_call(symbol, count)
        struct symtab * symbol; /* function */
        int     count;          /* # of arguments */
{       int pc;

        chk_parm(symbol, count);
        pc = gen(OP_CALL, count, symbol->s_offset, symbol->s_name);
        if (symbol->s_offset <= 0)
                symbol->s_offset = -pc; /* head of chain */
        while (count-- > 0)
                gen_pr(OP_POP, "pop argument");
        gen(OP_LOAD, MOD_GLOBAL, 0, "push result");
}
```

```
int gen_entry(symbol)
        struct symtab * symbol; /* function */
{
        symbol->s_offset = gen_label(symbol->s_offset);
        gen(OP_ENTRY, 0, 0, symbol->s_name);
        return symbol->s_offset;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* function */
        int label;
{       extern int l_max;       /* size of local region */

        prog[label].p_val = l_max;
        printf("%d:\tentry\t%d\t; %s\n",
                label, l_max, symbol->s_name);
}
```

```
end_program()
{       extern int g_offset;    /* size of global region */
        extern struct symtab * s_find();
        int main = s_find("main") -> s_offset;

        all_program();          /* allocate global variables */
        printf("%d:\tend\t%d,%d\n", pc, g_offset, main);
        simulate(pc, g_offset, main);
}
```

8.3 Example

Consider this recursive specification of Euclid's algorithm:

```
/*
 *      Euclid's algorithm (recursively)
 */

main()
{
        gcd(36,54);
}

gcd(a,b)
{
        if (a == b)
                return a;
        else if (a > b)
                return gcd(a-b, b);
        else
                return gcd(a, b-a);
}
```

Execution of the program yields the following result:

```
1:      10      0,0     ; main
2:      4       4,36    ; 36
3:      4       4,54    ; 54
4:      9       2,0     ; gcd
5:      6       0,0     ; pop argument
6:      6       0,0     ; pop argument
7:      4       1,0     ; push result
8:      6       0,0     ; clear stack
9:      11      0,0     ; end of function
1:      entry   0       ; main
4:      (fixup) 10
```

```
10:     10      0,0     ; gcd
11:     4       2,0     ; a
12:     4       2,1     ; b
13:     1       10,0    ; ==
14:     7       0,0     ; IF
15:     4       2,0     ; a
16:     5       1,0     ; save result
17:     11      0,0     ; RETURN
18:     8       0,0     ; past ELSE
14:     (fixup) 19
19:     4       2,0     ; a
20:     4       2,1     ; b
21:     1       7,0     ; >
22:     7       0,0     ; IF
23:     4       2,0     ; a
24:     4       2,1     ; b
25:     1       2,0     ; -
26:     4       2,1     ; b
27:     9       2,10    ; gcd
28:     6       0,0     ; pop argument
29:     6       0,0     ; pop argument
30:     4       1,0     ; push result
31:     5       1,0     ; save result
32:     11      0,0     ; RETURN
33:     8       0,0     ; past ELSE
22:     (fixup) 34
34:     4       2,0     ; a
35:     4       2,1     ; b
36:     4       2,0     ; a
37:     1       2,0     ; -
38:     9       2,10    ; gcd
39:     6       0,0     ; pop argument
40:     6       0,0     ; pop argument
41:     4       1,0     ; push result
42:     5       1,0     ; save result
43:     11      0,0     ; RETURN
33:     (fixup) 44
18:     (fixup) 44
44:     11      0,0     ; end of function
10:     entry   0       ; gcd
45:     end     1,1

Execution begins...

4:      call    10
14:     jumpz   19
22:     jumpz   34
38:     call    10
14:     jumpz   19
27:     call    10
16:     store   1,0     to 18
17:     return  18 to 28
31:     store   1,0     to 18
32:     return  18 to 39
42:     store   1,0     to 18
43:     return  18 to 5
```

```
9:      return  18 to 0
0:      halt
```

The example shows nicely that the code could be shortened by postprocessing, since, e.g., of a series of return instructions only the first can actually be reached.

8.4 Problems

1. Improve the simulator by having return pass result values on the stack, and by eliminating from `gen_call` the pop and load instructions following call. (See section 8.1.)

2. As was mentioned, the call, return, and entry instructions for our fictitious machine are not very realistic. On real machines, these instructions tend to be much simpler. Design more "realistic" instructions, and change the code generator and the simulator to use them. Hint: you will probably need some register manipulation instructions.

3. Add run-time options to the simulator, either through compiler options or by special comments which generate appropriate pseudo-instructions. Possible options include: printing or suppressing a listing of the compiled code before execution begins; an option of whether to execute the program, based on the presence or absence of compilation errors, or on the number or type of errors (ordinary errors, or warnings); a limit on the number of instructions to be simulated; a more extensive trace option, perhaps with a limit to the number or type of instructions to be traced.

4. Modify the simulator to handle arithmetic exceptions, such as division by zero.

5. Modify the simulator to use dynamically managed memory, rather than the fixed-size arrays of "tunable size" defined near the beginning of this chapter.

8. The program segment structure, defined near the beginning of this chapter to simplify decoding, is neither realistic of ordinary machine architecture, nor particularly compact. Modify this feature of the simulator to make **it** more efficient and more realistic.

8.5 Projects

1. Reorganize the compiler so that it consists of several separate programs: **(1)** the compiler of chapter 7, which generates assembler code; (2) a simple assembler (the assembler can, of course, be written using yacc and *lex!*); and (3) the simulator.

2. Using the three programs of the previous project, rearrange the compiler so that functions can be compiled separately, and the results combined either by the assembler or by a separate linking loader. Note that global variables might be included in any function compilation, and all must appear in the final executed program.

3. A large number of language extensions is conceivable: `char` or other integer-type variables, floating point variables, vectors with or without pointers, structures; other control features, such as `switch`, `for`, **`do while`.**

4. More interesting language extensions result from introducing parallelism, e.g., by introducing standard procedures (i.e., simulator instructions) for coroutine jumps in the style of Modula-2, or by adding a parallel control structure. Vectors should be added also to make this project more realistic.

```
#include "gen.h"

#include "sim.h"
```

samplec.y

```
/*
 *      sample c
 *      syntax analysis with error r
 *      symbol table
 *      memory allocation
 *      code generation
 *      (s/r conflicts: one on ELSE,
 */

%{
#include "symtab.h"      /* symbol ta
#include "gen.h"         /* code gene

#define OFFSET(x)        ( ((struct s
#define NAME(x)          ( ((struct s

extern int l_offset, l_max;
%}

%union  {
        struct symtab * y_sym;  /* I
        char * y_str;           /* C
        int y_num;              /* c
        int y_lab;              /* l
        }

/*
 *      terminal symbols
 */

%token  <y_sym> Identifier
%token  <y_str> Constant
```

```
ecovery


 one on error)


ble mnemonics */
ration mnemonics */

ymtab *) x) -> s_offset )
ymtab *) x) -> s_name )




dentifier */
onstant */
ount */
abel */
```

```
%token  INT
%token  IF
%token  ELSE
%token  WHILE
%token  BREAK
%token  CONTINUE
%token  RETURN
%token  ';'
%token  '('
%token  ')'
%token  '{'
%token  '}'
%token  '+'
%token  '-'
%token  '*'
%token  '/'
%token  '%'
%token  '>'
%token  '<'
%token  GE      /* >= */
%token  LE      /* <= */
%token  EQ      /* == */
%token  NE      /* != */
%token  '&'
%token  '^'
%token  '|'
%token  '='
%token  PE      /* += */
%token  ME      /* -= */
%token  TE      /* *= */
%token  DE      /* /= */
%token  RE      /* %= */
%token  PP      /* ++ */
%token  MM      /* -- */
%token  ','

/*
 *      typed non-terminal symbols
 */

%type   <y_sym> optional_parameter_list, parameter_list
%type   <y_num> optional_argument_list, argument_list
%type   <y_lab> if_prefix, loop_prefix

/*
 *      precedence table
 */

%right  '=' PE ME TE DE RE
%left   '|'
%left   '^'
%left   '&'
%left   EQ NE
%left   '<' '>' GE LE
%left   '+' '-'
%left   '*' '/' '%'
```

```
%right  PP MM

%%

program
        :       { init(); }
          definitions
                { end_program(); }

definitions
        : definition
        | definitions definition
                { yyerrok; }
        | error
        | definitions error

definition
        : function_definition
        | INT function_definition
        | declaration

function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                  $<y_lab>$ = gen_entry($1);
                }
          compound_statement
                { all_func($1);
                  gen_pr(OP_RETURN, "end of function");
                  fix_entry($1, $<y_lab>7);
                }

optional_parameter_list
        : /* no formal parameters */
                { $$ = (struct symtab *) 0; }
        | parameter_list
                /* $$ = $1 = chain of formal parameters */

parameter_list
        : Identifier
                { $$ = link_parm($1, (struct symtab *) 0); }
        | Identifier ',' parameter_list
                { $$ = link_parm($1, $3);
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | error parameter_list
                { $$ = $2; }
```

```
        | Identifier error parameter_list
                { $$ = link_parm($1, $3); }
        | error ',' parameter_list
                { $$ = $3;
                  yyerrok;
                }

parameter_declarations
        : /* null */
        | parameter_declarations parameter_declaration
                { yyerrok; }
        | parameter_declarations error

parameter_declaration
        : INT parameter_declarator_list sc

parameter_declarator_list
        : Identifier
                { make_parm($1); }
        | parameter_declarator_list ',' Identifier
                { make_parm($3);
                  yyerrok;
                }
        | error
        | parameter_declarator_list error
        | parameter_declarator_list error Identifier
                { make_parm($3);
                  yyerrok;
                }
        | parameter_declarator_list ',' error

compound_statement
        : '{'
                { $<y_lab>$ = l_offset;
                  blk_push();
                }
          declarations statements rr
                { if (l_offset > l_max)
                        l_max = l_offset;
                  l_offset = $<y_lab>2;
                  blk_pop();
                }

declarations
        : /* null */
        | declarations declaration
                { yyerrok; }
        | declarations error

declaration
        : INT declarator_list sc

declarator_list
        : Identifier
                { all_var($1); }
        | declarator_list ',' Identifier
```

```
                { all_var($3);
                  yyerrok;
                }
        | error
        | declarator_list error
        | declarator_list error Identifier
                { all_var($3);
                  yyerrok;
                }
        | declarator_list ',' error

statements
        : /* null */
        | statements statement
                { yyerrok; }
        | statements error

statement
        : expression sc
                { gen_pr(OP_POP, "clear stack"); }
        | sc
        | BREAK sc
                { gen_break(); }
        | CONTINUE sc
                { gen_continue(); }
        | RETURN sc
                { gen_pr(OP_RETURN, "RETURN"); }
        | RETURN expression sc
                { gen(OP_STORE, MOD_GLOBAL, 0, "save result");
                  gen_pr(OP_RETURN, "RETURN");
                }
        | compound_statement
        | if_prefix statement
                { gen_label($1); }
        | if_prefix statement ELSE
                { $<y_lab>$ = gen_jump(OP_JUMP, new_label(),
                        "past ELSE");
                  gen_label($1);
                }
          statement
                { gen_label($<y_lab>4); }
        | loop_prefix
                { $<y_lab>$ = gen_jump(OP_JUMPZ, new_label(),
                        "WHILE");
                  push_break($<y_lab>$);
                }
          statement
                { gen_jump(OP_JUMP, $1, "repeat WHILE");
                  gen_label($<y_lab>2);
                  pop_break();
                  pop_continue();
                }

if_prefix
        : IF '(' expression rp
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }
```

```
        | IF error
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }

loop_prefix
        : WHILE '('
                { $<y_lab>$ = gen_label(new_label());
                  push_continue($<y_lab>$);
                }
          expression rp
                { $$ = $<y_lab>3; }
        | WHILE error
                { $$ = gen_label(new_label());
                  push_continue($$);
                }

expression
        : binary
        | expression ','
                { gen_pr(OP_POP, "discard"); }
          binary
                { yyerrok; }
        | error ',' binary
                { yyerrok; }
        | expression error
        | expression ',' error

binary
        : Identifier
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
                }
        | Constant
                { gen_li($1); }
        | '(' expression rp
        | '(' error rp
        | Identifier '('
                { chk_func($1); }
          optional_argument_list rp
                { gen_call($1,$4); }
        | PP Identifier
                { chk_var($2);
                  gen(OP_INC, gen_mod($2), OFFSET($2), NAME($2));
                }
        | MM Identifier
                { chk_var($2);
                  gen(OP_DEC, gen_mod($2), OFFSET($2), NAME($2));
                }
        | binary '+' binary
                { gen_alu(ALU_ADD, "+"); }
        | binary '-' binary
                { gen_alu(ALU_SUB, "-"); }
        | binary '*' binary
                { gen_alu(ALU_MUL, "*"); }
        | binary '/' binary
                { gen_alu(ALU_DIV, "/"); }
        | binary '%' binary
```

```
        { gen_alu(ALU_MOD, "%"); }
| binary '>' binary
        { gen_alu(ALU_GT, ">"); }
| binary '<' binary
        { gen_alu(ALU_LT, "<"); }
| binary GE binary
        { gen_alu(ALU_GE, ">="); }
| binary LE binary
        { gen_alu(ALU_LE, "<="); }
| binary EQ binary
        { gen_alu(ALU_EQ, "=="); }
| binary NE binary
        { gen_alu(ALU_NE, "!="); }
| binary '&' binary
        { gen_alu(ALU_AND, "&"); }
| binary '^' binary
        { gen_alu(ALU_XOR, "^"); }
| binary '|' binary
        { gen_alu(ALU_OR, "|"); }
| Identifier '=' binary
        { chk_var($1);
          gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
        }
| Identifier PE
        { chk_var($1);
          gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
        }
  binary
        { gen_alu(ALU_ADD, "+");
          gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
        }
| Identifier ME
        { chk_var($1);
          gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
        }
  binary
        { gen_alu(ALU_SUB, "-");
          gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
        }
| Identifier TE
        { chk_var($1);
          gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
        }
  binary
        { gen_alu(ALU_MUL, "*");
          gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
        }
| Identifier DE
        { chk_var($1);
          gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
        }
  binary
        { gen_alu(ALU_DIV, "/");
          gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
        }
| Identifier RE
```

```
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1), NAME($1));
                }
          binary
                { gen_alu(ALU_MOD, '%');
                  gen(OP_STORE, gen_mod($1), OFFSET($1), NAME($1));
                }

optional_argument_list
        : /* no actual arguments */
                { $$ = 0; }
        | argument_list
                /* $$ = $1 = # of actual arguments */

argument_list
        : binary
                { $$ = 1; }
        | argument_list ',' binary
                { ++ $$;
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | argument_list error
        | argument_list ',' error

/*
 *      make certain terminal symbols very important
 */

rp      : ')'   { yyerrok; }
sc      : ';'   { yyerrok; }
rr      : '}'   { yyerrok; }
```

A parser is then prepared with the following command:

```
pace -d samplec.y
```

The —d option instructs *yacc* to produce both the parser *y.tab.c* and the token definition file *y.tab.h.*

A.2 Lexical analysis

The lexical analyzer function in file *samplec.l* was shown in section 2.7. From this file, the lexical analyzer is prepared with the following command:

```
lex samplec.l
```

A.3 Messages

File *message.c* contains the C functions introduced in chapter 5 to issue error messages. (The comments /*VARARGS1*/ prevent lint from complaining about the varying number of arguments with which these functions are called.)

message.c

```
/*
 *      message routines
 */

#include <stdio.h>

#define VARARG  fmt, v1, v2, v3, v4, v5
#define VARPARM (VARARG) char * fmt;

extern FILE * yyerfp;

                                        /*VARARGS1*/
message VARPARM
{
        yywhere();
        fprintf(yyerfp, VARARG);
        putc('\n', yyerfp);
}

                                        /*VARARGS1*/
error VARPARM
{       extern int yynerrs;

        fprintf(yyerfp, "[error %d] ", ++ yynerrs);
        message(VARARG);
}

                                        /*VARARGS1*/
warning VARPARM
{
        fputs("[warning] ", yyerfp);
        message(VARARG);
}

                                        /*VARARGS1*/
fatal VARPARM
{
        fputs("[fatal error] ", yyerfp);
        message(VARARG);
        exit(1);
}

                                        /*VARARGS1*/
bug VARPARM
{
        fputs("BUG: ", yyerfp);
        message(VARARG);
        exit(1);
}
```

```
char * strsave(s)
        register char * s;
{       register char * cp = calloc(strlen(s)+1, 1);

        if (cp)
        {       strcpy(cp, s);
                return cp;
        }
        fatal("No more room to save strings.");
}
```

symtab.h

```
/*
 *      sample c -- header file for symbol table
 */

struct symtab {
        char *  s_name;         /* name pointer */
        int     s_type;         /* symbol type */
        int     s_blknum;       /* static block depth */
        union {                 /* multi-purpose */
                int s__num;
                struct symtab * s__link;
                } s__;
        int     s_offset;       /* symbol definition */
        struct symtab * s_next; /* next entry */
        };

#define s_pnum  s__.s__num      /* count of parameters */
#define NOT_SET (-1)            /* no count yet set */
#define s_plist s__.s__link     /* chain of parameters */

/*
 *      s_type values
 */

#define UDEC    0       /* not declared */
#define FUNC    1       /* function */
#define UFUNC   2       /* undefined function */
#define VAR     3       /* declared variable */
#define PARM    4       /* undeclared parameter */

/*
 *      s_type values for S_TRACE
 */

#define SYMmap  "undeclared", "function", "undefined function", \
                "variable", "parameter"

/*
 *      typed functions, symbol table module
```

```
 */

struct symtab * link_parm();     /* chain parameters */
struct symtab * s_find();        /* locate symbol by name */
struct symtab * make_parm();     /* declare parameter */
struct symtab * make_var();      /* define variable */
struct symtab * make_func();     /* define function */

/*
 *      typed library functions
 */

char * strsave();                /* dynamically save a string */
char * calloc();                 /* dynamically obtain memory */
```

symtab.c

```
/*
 *      sample c -- symbol table definition and manipulation
 */

#include "symtab.h"
#include "y.tab.h"

/*
 *      symbol table
 */

static  struct  symtab
        symtab,                  /* blind element */
        * s_gbl;                 /* global end of chain */
#define s_lcl   (& symtab)       /* local end of chain */

/*
 *      block table
 */

static int blknum = 0;           /* current static block depth */

/*
 *      add a new name to local region
 */

static struct symtab * s_create(name)
        register char * name;
{       register struct symtab * new_entry = (struct symtab *)
                        calloc(1, sizeof(struct symtab));

        if (new_entry)
        {       new_entry->s_next = s_lcl->s_next;
                s_lcl->s_next = new_entry;
                new_entry->s_name = strsave(name);
                new_entry->s_type = UDEC;
                new_entry->s_blknum = 0;
```

```
                new_entry->s_pnum = NOT_SET;
                return new_entry;
        }
        fatal("No more room for symbols.");
        /*NOTREACHED*/
}

/*
 *      move an entry from local to global region
 */

static s_move(symbol)
        register struct symtab * symbol;
{       register struct symtab * ptr;

        /* find desired entry in symtab chain (bug if missing) */
        for (ptr = s_lcl; ptr->s_next != symbol; ptr = ptr->s_next)
                if (! ptr->s_next)
                        bug("s_move");

        /* unlink it from its present position */
        ptr->s_next = symbol->s_next;

        /* relink at global end of symtab */
        s_gbl->s_next = symbol;
        s_gbl = symbol;
        s_gbl->s_next = (struct symtab *) 0;
}

/*
 *      initialize symbol and block table
 */

init()
{
        blk_push();
        s_gbl = s_create("main");
        s_gbl->s_type = UFUNC;
}

/*
 *      push block table
 */

blk_push()
{
        ++ blknum;
}

/*
 *      locate entry by name
 */

struct symtab * s_find(name)
        char * name;
{       register struct symtab * ptr;
```

```
        /* search symtab until match or end of symtab chain */
        for (ptr = s_lcl->s_next; ptr; ptr = ptr->s_next)
                if (! ptr->s_name)
                        bug("s_find");
                else
                        /* return ptr if names match */
                        if (strcmp(ptr->s_name, name) == 0)
                                return ptr;
        /* search fails, return NULL */
        return (struct symtab *) 0;
}

/*
 *      interface for lexical analyzer:
 *      locate or enter Identifier, save text of Constant
 */

s_lookup(yylex)
        int yylex;              /* Constant or Identifier */
{       extern char yytext[];   /* text of symbol */

        switch (yylex) {
        case Constant:
                yylval.y_str = strsave(yytext);
                break;
        case Identifier:
                if (yylval.y_sym = s_find(yytext))
                        break;
                yylval.y_sym = s_create(yytext);
                break;
        default:
                bug("s_lookup");
        }
}

/*
 *      mark entry as part of parameter_list
 */

struct symtab * link_parm(symbol, next)
        register struct symtab * symbol, * next;
{
        switch (symbol->s_type) {
        case PARM:
                error("duplicate parameter %s", symbol->s_name);
                return next;
        case FUNC:
        case UFUNC:
        case VAR:
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        default:
                bug("link_parm");
        }
        symbol->s_type = PARM;
```

```
        symbol->s_blknum = blknum;
        symbol->s_plist = next;
        return symbol;
}

/*
 *      declare a parameter
 */

struct symtab * make_parm(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
                if (symbol->s_blknum == 2)
                {       error("parameter %s declared twice",
                                symbol->s_name);
                        return symbol;
                }
        case UDEC:
        case FUNC:
        case UFUNC:
                error("%s is not a parameter", symbol->s_name);
                symbol = s_create(symbol->s_name);
        case PARM:
                break;
        default:
                bug("make_parm");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}

/*
 *      define a variable
 */

struct symtab * make_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
        case FUNC:
        case UFUNC:
                if (symbol->s_blknum == blknum
                    || symbol->s_blknum == 2 && blknum == 3)
                        error("duplicate name %s", symbol->s_name);
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                break;
        default:
                bug("make_var");
```

```
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}

/*
 *      define a function
 */

struct symtab * make_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UFUNC:
        case UDEC:
                break;
        case VAR:
                error("function name %s same as global variable",
                        symbol->s_name);
                return symbol;
        case FUNC:
                error("duplicate function definition %s",
                        symbol->s_name);
                return symbol;
        default:
                bug("make_func");
        }
        symbol->s_type = FUNC;
        symbol->s_blknum = 1;
        return symbol;
}

/*
 *      set or verify number of parameters
 */

chk_parm(symbol, count)
        register struct symtab * symbol;
        register int count;
{
        if (symbol->s_pnum == NOT_SET)
                symbol->s_pnum = count;
        else if ((int) symbol->s_pnum != count)
                warning("function %s should have %d argument(s)",
                        symbol->s_name, symbol->s_pnum);
}

/*
 *      default undeclared parameters, count
 */

int parm_default(symbol)
        register struct symtab * symbol;
{       register int count = 0;
```

```
        while (symbol)
        {       ++ count;
                if (symbol->s_type == PARM)
                        symbol->s_type = VAR;
                symbol = symbol->s_plist;
        }
        return count;
}

/*
 *      pop block table
 */

blk_pop()
{       register struct symtab * ptr;

        for (ptr = s_lcl->s_next;
             ptr &&
             (ptr->s_blknum >= blknum || ptr->s_blknum == 0);
             ptr = s_lcl->s_next)
        {
                if (! ptr->s_name)
                        bug("blk_pop null name");
#ifdef TRACE
        {       static char * type[] = { SYMmap };

                message("Popping %s: %s, depth %d, offset %d",
                        ptr->s_name, type[ptr->s_type],
                        ptr->s_blknum, ptr->s_offset);
        }
#endif TRACE
                if (ptr->s_type == UFUNC)
                        error("undefined function %s",
                                ptr->s_name);
                cfree(ptr->s_name);
                s_lcl->s_next = ptr->s_next;
                cfree(ptr);
        }
        -- blknum;
}

/*
 *      check reference or assignment to variable
 */

chk_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                error("undeclared variable %s", symbol->s_name);
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                break;
        case FUNC:
```

```
        case UFUNC:
                error("function %s used as variable",
                        symbol->s_name);
        case VAR:
                return;
        default:
                bug("check_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
}

/*
 *      check reference to function, implicitly declare it
 */

chk_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                break;
        case PARM:
                error("unexpected parameter %s", symbol->s_name);
                symbol->s_pnum = NOT_SET;
                return;
        case VAR:
                error("variable %s used as function",
                        symbol->s_name);
                symbol->s_pnum = NOT_SET;
        case UFUNC:
        case FUNC:
                return;
        default:
                bug("check_func");
        }
        s_move(symbol);
        symbol->s_type = UFUNC;
        symbol->s_blknum = 1;
}
```

A.5 Memory allocation

The C functions for memory allocation, introduced in chapter 6, are contained in file *mem.c*.

mem.c

```
/*
 *      sample c -- memory allocation
 */

#include "symtab.h"

/*
 *      global counters
```

```
 */

int     g_offset = 1,           /* offset in global region */
        l_offset = 0,           /* offset in local region */
        l_max;                  /* size of local region */

/*
 *      allocate a (global or local) variable
 */

all_var(symbol)
        register struct symtab * symbol;
{       extern struct symtab * make_var();

        symbol = make_var(symbol);

        /* if not in parameter region, assign suitable offset */
        switch (symbol->s_blknum) {
        default:                        /* local region */
                symbol->s_offset = l_offset++;
        case 2:                         /* parameter region */
                break;
        case 1:                         /* global region */
                symbol->s_offset = g_offset++;
                break;
        case 0:
                bug("all_var");
        }
}

/*
 *      complete allocation
 */

all_program()
{
        blk_pop();

#ifdef  TRACE
        message("global region has %d word(s)", g_offset);
#endif
}

/*
 *      allocate all parameters
 */

all_parm(symbol)
        register struct symtab * symbol;
{       register int p_offset = 0;

        while (symbol)
        {       symbol->s_offset = p_offset ++;
                symbol = symbol->s_plist;
        }
```

```
#ifdef  TRACE
        message("parameter region has %d word(s)", p_offset);
#endif
}

/*
 *      complete allocation of a function
 */

all_func(symbol)
        struct symtab * symbol;
{
        blk_pop();

#ifdef  TRACE
        message("local region has %d word(s)", l_max);
#endif
}
```

```
/*
 *      sample c -- header file for code generation
 */

/*
 *      operation codes for pseudo machine
 */

#define OP_ALU    "alu"         /* arithmetic-logic-op  */
#define OP_DEC    "dec"         /* region,offset        */
#define OP_INC    "inc"         /* region,offset        */
#define OP_LOAD   "load"        /* region,offset        */
#define OP_STORE  "store"       /* region,offset        */
#define OP_POP    "pop"         /*                      */
#define OP_JUMPZ  "jumpz"       /* label                */
#define OP_JUMP   "jump"        /* label                */
#define OP_CALL   "call"        /* parm-count,address   */
#define OP_ENTRY  "entry"       /* local-frame-size     */
#define OP_RETURN "return"      /*                      */

/*
 *      region modifiers
 */

#define MOD_GLOBAL "gbl"        /* global region        */
#define MOD_PARAM  "par"        /* parameter region     */
#define MOD_LOCAL  "lcl"        /* local region         */
#define MOD_IMMED  "con"        /* load only: Constant  */

/*
```

```
 *      OP_ALU modifiers
 */

#define ALU_ADD "+"             /* addition             */
#define ALU_SUB "-"             /* subtraction          */
#define ALU_MUL "*"             /* multiplication       */
#define ALU_DIV "/"             /* division             */
#define ALU_MOD "%"             /* remainder            */
#define ALU_LT  "<"             /* compares as: <       */
#define ALU_GT  ">"             /*              >       */
#define ALU_LE  "<="            /*              <=      */
#define ALU_GE  ">="            /*              >=      */
#define ALU_EQ  "=="            /*              ==      */
#define ALU_NE  "!="            /*              !=      */
#define ALU_AND "&"             /* bit-wise and         */
#define ALU_OR  "|"             /* bit-wise or          */
#define ALU_XOR "^"             /* bit-wise excl. or    */

/*
 *      typed functions, code generator
 */

char * gen_mod();               /* region modifier */
```

File *gen.c* includes the C functions which generate assembler code for our fictitious *sampleC* machine. These routines were introduced in chapter 7.

gen.c

```
/*
 *      sample c -- code generation
 */

#include "symtab.h"
#include "gen.h"

/*
 *      generate various instruction formats
 */

gen_alu(mod, comment)
        char * mod;             /* mnemonic modifier */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t%s\t\t; %s\n", OP_ALU, mod, comment);
}

gen_li(const)
        char * const;           /* Constant value */
{
        printf("\t%s\t%s,%s\n", OP_LOAD, MOD_IMMED, const);
}

char * gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
```

```
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}

gen(op, mod, val, comment)
        char * op;              /* mnemonic operation code */
        char * mod;             /* mnemonic modifier */
        int val;                /* offset field */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t%s,%d\t\t; %s\n", op, mod, val, comment);
}

gen_pr(op, comment)
        char * op;              /* mnemonic operation code */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t\t\t; %s\n", op, comment);
}

/*
 *      generate printable internal label
 */

#define LABEL   "$$%d"

static char * format_label(label)
        int label;
{       static char buffer[sizeof LABEL + 2];

        sprintf(buffer, LABEL, label);
        return buffer;
}

/*
 *      generate jumps, return target
 */

int gen_jump(op, label, comment)
        char * op;              /* mnemonic operation code */
        int label;              /* target of jump */
        char * comment;         /* instruction comment */
{
        printf("\t%s\t%s\t\t; %s\n", op, format_label(label),
                comment);
        return label;
}

/*
 *      generate unique internal label
 */
```

```
int new_label()
{       static int next_label = 0;

        return ++next_label;
}

/*
 *      define internal label
 */

int gen_label(label)
        int label;
{
        printf("%s\tequ\t*\n", format_label(label));
        return label;
}

/*
 *      label stack manager
 */

static struct bc_stack {
        int bc_label;           /* label from new_label */
        struct bc_stack * bc_next;
        } * b_top,              /* head of break stack */
          * c_top;              /* head of continue stack */

static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));

        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("No more room to compile loops.");
        /*NOTREACHED*/
}

static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue stack underflow");
        /*NOTREACHED*/
}
```

```
static int top(stack)
        struct bc_stack * stack;
{
        if (! stack)
        {       error("no loop open");
                return 0;
        }
        else
                return stack->bc_label;
}

/*
 *      BREAK and CONTINUE
 */

push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

pop_break()
{
        b_top = pop(b_top);
}

pop_continue()
{
        c_top = pop(c_top);
}

gen_break()
{
        gen_jump(OP_JUMP, top(b_top), "BREAK");
}

gen_continue()
{
        gen_jump(OP_JUMP, top(c_top), "CONTINUE");
}

/*
 *      function call
 */

gen_call(symbol, count)
        struct symtab * symbol; /* function */
        int count;              /* # of arguments */
{
        chk_parm(symbol, count);
```

```
        printf("\t%s\t%d,%s\n", OP_CALL, count, symbol->s_name);
        while (count-- > 0)
                gen_pr(OP_POP, "pop argument");
        gen(OP_LOAD, MOD_GLOBAL, 0, "push result");
}

/*
 *      function prologue
 */

int gen_entry(symbol)
        struct symtab * symbol; /* function */
{       int label = new_label();

        printf("%s\t", symbol->s_name);
        printf("%s\t%s\n", OP_ENTRY, format_label(label));
        return label;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* function */
        int label;
{       extern int l_max;       /* size of local region */

        printf("%s\tequ\t%d\t\t; %s\n", format_label(label),
                l_max, symbol->s_name);
}

/*
 *      wrap-up
 */

end_program()
{       extern int g_offset;    /* size of global region */

        all_program();          /* allocate global variables */
        printf("\tend\t%d,main\n", g_offset);
}
```

```
/*
 *      sample c -- header file for simulation
 */

/*
 *      operation codes for pseudo machine
 */

#define OP_ALU    1     /* alu   arithmetic-logic-op    */
#define OP_DEC    2     /* dec   region,offset          */
#define OP_INC    3     /* inc   region,offset          */
#define OP_LOAD   4     /* load  region,offset          */
#define OP_STORE  5     /* store region,offset          */
#define OP_POP    6     /* pop                          */
#define OP_JUMPZ  7     /* jumpz label                  */
#define OP_JUMP   8     /* jump  label                  */
#define OP_CALL   9     /* call  routine-address        */
#define OP_ENTRY  10    /* entry local-frame-size       */
#define OP_RETURN 11    /* return                       */

/*
 *      region modifiers
 */

#define MOD_GLOBAL 1    /* global region        */
#define MOD_PARAM  2    /* parameter region     */
#define MOD_LOCAL  3    /* local region         */
#define MOD_IMMED  4    /* load only: Constant  */

/*
 *      OP_ALU modifiers
 */

#define ALU_ADD 1       /* addition             */
#define ALU_SUB 2       /* subtraction          */
#define ALU_MUL 3       /* multiplication       */
#define ALU_DIV 4       /* division             */
#define ALU_MOD 5       /* remainder            */
#define ALU_LT  6       /* compares as: <       */
#define ALU_GT  7       /*              >       */
#define ALU_LE  8       /*              <=      */
#define ALU_GE  9       /*              >=      */
#define ALU_EQ  10      /*              ==      */
#define ALU_NE  11      /*              !=      */
#define ALU_AND 12      /* bit-wise and         */
#define ALU_OR  13      /* bit-wise or          */
#define ALU_XOR 14      /* bit-wise excl. or    */

/*
 *      program memory structure
 */

struct prog {
        short p_op;     /* operation code */
```

```
        short p_mod;    /* modifier */
        int p_val;      /* offset or other value */
        };

/*
 *      tunable limits
 */

#define L_PROG  200     /* max. program size */
#define L_DATA  100     /* max. area for stack, etc. */
#define DIM(x)  (sizeof x / sizeof x[0]) /* extent */
```

```
/*
 *      sample c -- machine simulator
 */

#include "sim.h"

/*
 *      data and program memory
 */

static int data[L_DATA];
extern struct prog prog[];

/*
 *      registers
 */

static struct prog * inst;      /* -> current instruction */
#define G       0               /* global segment */
static int P;                   /* current parameter segment */
static int L;                   /* current local segment */
static int T;                   /* top of stack */

/*
 *      shorthand notations
 */

#define TOP     data[T-1]       /* right operand: top of stack */
#define NEXT    data[T-2]       /* left operand: below TOP */
#define PUSH    data[T++]       /* new cell to come onto stack */
#define POP     -- T            /* -> discarded cell from stack */
#define MEMORY  data[address()] /* effectively addressed cell */
#define RESULT  data[G]         /* result value of function */

/*
 *      address decoder
 */

static int address()    /* effective current data address */
{       register int ad;
```

```
        switch (inst->p_mod) {
        case MOD_GLOBAL:
                ad = G;
                break;
        case MOD_PARAM:
                ad = P;
                break;
        case MOD_LOCAL:
                ad = L;
                break;
        default:
                bug("invalid p_mod");
        }
        ad += inst->p_val;
        if (ad < 0 || ad >= T)
                bug("invalid effective address");
        return ad;
}

/*
 *      simulator
 */

simulate(pc_limit, global, pc)
        int pc_limit, global, pc;
{
        /* initialize */
        if (global >= DIM(data))
                fatal("Not enough room for global data.");
        T = global + 2;

        printf("\nExecution begins...\n\n");

        for (;;)
        {       /* fetch */
                if (pc < 0 || pc >= pc_limit)
                        bug("pc not in program area");
                inst = &prog[pc++];

                /* decode operation and dispatch */
                switch (inst->p_op) {
                default:
                        printf("%d:\thalt\n", inst-prog);
                        return;
                case OP_ALU:
                        if (T <= L+1)
                                bug("simulator stack underflow");
                        switch (inst->p_mod) {
                        default:
                                bug("illegal ALU instruction");
                        case ALU_ADD:   NEXT += TOP; break;
                        case ALU_SUB:   NEXT -= TOP; break;
                        case ALU_MUL:   NEXT *= TOP; break;
                        case ALU_DIV:   NEXT /= TOP; break;
                        case ALU_MOD:   NEXT %= TOP; break;
                        case ALU_LT:    NEXT = NEXT < TOP; break;
```

```
            case ALU_GT:    NEXT = NEXT > TOP; break;
            case ALU_LE:    NEXT = NEXT <= TOP; break;
            case ALU_GE:    NEXT = NEXT >= TOP; break;
            case ALU_EQ:    NEXT = NEXT == TOP; break;
            case ALU_NE:    NEXT = NEXT != TOP; break;
            case ALU_AND:   NEXT &= TOP; break;
            case ALU_OR:    NEXT |= TOP; break;
            case ALU_XOR:   NEXT ^= TOP; break;
            }
            POP;
            break;
    case OP_LOAD:
            if (T >= DIM(data))
                    fatal("Too much data.");
            if (inst->p_mod == MOD_IMMED)
                    PUSH = inst->p_val;
            else
                    PUSH = MEMORY;
            break;
    case OP_STORE:
            if (T <= L)
                    bug("simulator stack underflow");
            printf("%d:\tstore\t%d,%d\tto %d\n",
                    inst-prog, inst->p_mod,
                    inst->p_val, MEMORY = TOP);
            break;
    case OP_INC:
            if (T >= DIM(data))
                    fatal("Too much data.");
            printf("%d:\tinc\t%d,%d\tto %d\n",
                    inst-prog, inst->p_mod,
                    inst->p_val, PUSH = ++ MEMORY);
            break;

    case OP_DEC:
            if (T >= DIM(data))
                    fatal("Too much data.");
            printf("%d:\tdec\t%d,%d\tto %d\n",
                    inst-prog, inst->p_mod,
                    inst->p_val, PUSH = -- MEMORY);
            break;
    case OP_POP:
            if (T <= L)
                    bug("simulator stack underflow");
            POP;
            break;
    case OP_JUMP:
            printf("%d:\tjump\t%d\n", inst-prog,
                    inst->p_val);
            pc = inst->p_val;
            break;
    case OP_JUMPZ:
            if (T <= L)
                    bug("simulator stack underflow");
            if (data[POP] == 0)
            {       printf("%d:\tjumpz\t%d\n",
```

```
                                        inst->prog, inst->p_val);
                                pc = inst->p_val;
                        }
                        break;
                case OP_CALL:
                        printf("%d:\tcall\t%d\n", inst->prog,
                                inst->p_val);
                        PUSH = pc;
                        pc = inst->p_val;
                        PUSH = P;
                        P = T - 2 - inst->p_mod;
                        break;
                case OP_ENTRY:
                        PUSH = L;
                        L = T;
                        T += inst->p_val;
                        if (T >= DIM(data))
                                fatal("Too much data.");
                        break;
                case OP_RETURN:
                        if (T < L)
                                bug("simulator stack underflow");
                        T = L;
                        L = data[POP];
                        P = data[POP];
                        pc = data[POP];
                        printf("%d:\treturn\t%d to %d\n",
                                inst->prog, RESULT, pc);
                        break;
                }
        }
}
```

simgen.c

```
/*
 *      sample c -- code generator for simulator
 */

#include "sim.h"
#include "symtab.h"

/*
 *      program memory
 */
struct prog prog[L_PROG];
static int pc = 1;                      /* current program counter */
                                        /* HALT (0) is at address 0 */

/*
 *      generate a single instruction
 */

int gen(op, mod, val, comment)
```

```
        int op;                 /* operation code */
        int mod;                /* modifier */
        int val;                /* offset field */
        char * comment;         /* instruction comment */
{
        if (pc >= DIM(prog))
                fatal("Not enough program memory.");
        prog[pc].p_op = op;
        prog[pc].p_mod = mod;
        prog[pc].p_val = val;
        printf("%d:\t%d\t%d,%d\t; %s\n",
                pc, op, mod, val, comment);
        return pc ++;
}

/*
 *      region modifier
 */

int gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}

/*
 *      general instructions
 */

gen_alu(mod, comment)
        int mod;                /* modifier */
        char * comment;         /* instruction comment */
{
        gen(OP_ALU, mod, 0, comment);
}

gen_li(const)
        char * const;           /* Constant value */
{
        gen(OP_LOAD, MOD_IMMED, atoi(const), const);
}

gen_pr(op, comment)
        int op;                 /* operation code */
        char * comment;         /* instruction comment */
{
        gen(op, 0, 0, comment);
}

/*
```

```
 *      generate jump, return target or chain
 */

int gen_jump(op, label, comment)
        int op;                 /* operation code */
        int label;              /* target of jump */
        char * comment;         /* instruction comment */
{       int pc = gen(op, 0, label, comment);

        if (label <= 0)
                return -pc;     /* new head of chain */
        else
                return label;   /* already defined */
}

/*
 *      generate tail of forward branch chain
 */

int new_label()
{
        return 0;               /* end of chain */
}

/*
 *      resolve forward branch chain
 */

int gen_label(chain)
        int chain;
{       int next;

        while (chain < 0)
        {       chain = - chain;
                next = prog[chain].p_val;
                if (next > 0)
                        break;  /* already ok */
                prog[chain].p_val = pc;
                printf("%d:\t(fixup)\t%d\n", chain, pc);
                chain = next;
        }
        return pc;
}

/*
 *      label stack manager
 */

static struct bc_stack {
        int bc_label;           /* label from new_label */
        struct bc_stack * bc_next;
        } * b_top,              /* head of break stack */
          * c_top;              /* head of continue stack */

static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
```

```
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));

        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("No more room to compile loops.");
        /*NOTREACHED*/
}

static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue stack underflow");
        /*NOTREACHED*/
}

static int * top(stack)
        struct bc_stack ** stack;
{
        if (! *stack)
        {       error("no loop open");
                *stack = push(*stack, 0);
        }
        return & (*stack) -> bc_label;
}

/*
 *      BREAK and CONTINUE
 */

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

gen_break()
{
        *top(&b_top) = gen_jump(OP_JUMP, *top(&b_top), "BREAK");
```

```
}

gen_continue()
{
        *top(&c_top) = gen_jump(OP_JUMP, *top(&c_top), "CONTINUE");
}

pop_break()
{
        gen_label(*top(&b_top));
        b_top = pop(b_top);
}

pop_continue()
{
        gen_label(*top(&c_top));
        c_top = pop(c_top);
}

/*
 *      function call
 */

gen_call(symbol, count)
        struct symtab * symbol; /* function */
        int     count;          /* # of arguments */
{       int pc;

        chk_parm(symbol, count);
        pc = gen(OP_CALL, count, symbol->s_offset, symbol->s_name);
        if (symbol->s_offset <= 0)
                symbol->s_offset = -pc; /* head of chain */
        while (count-- > 0)
                gen_pr(OP_POP, "pop argument");
        gen(OP_LOAD, MOD_GLOBAL, 0, "push result");
}

/*
 *      function prologue and definition
 */

int gen_entry(symbol)
        struct symtab * symbol; /* function */
{
        symbol->s_offset = gen_label(symbol->s_offset);
        gen(OP_ENTRY, 0, 0, symbol->s_name);
        return symbol->s_offset;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* function */
        int label;
{       extern int l_max;       /* size of local region */

        prog[label].p_val = l_max;
        printf("%d:\tentry\t%d\t; %s\n",
```

```
                label, l_max, symbol->s_name);
}

/*
 *      wrap-up
 */

end_program()
{       extern int g_offset;    /* size of global region */
        extern struct symtab * s_find();
        int main = s_find("main") -> s_offset;

        all_program();          /* allocate global variables */
        printf("%d:\tend\t%d,%d\n", pc, g_offset, main);
        simulate(pc, g_offset, main);
}
```

```
cc sim.c simgen.c mem.c symtab.c message.c y.tab.c lex.yy.c \
        colib -ll -o sim
```

```
switch (yyerrflag) {
case 0:                         /* brand new error */
        yyerror("syntax error");
yyerrlab:
        ++yynerrs;
case 1:                         /* incompletely recovered */
case 2:                         /* ... try again */
        yyerrflag = 3;
        ...
```

```
switch (yyerrflag) {
case 0:                                 /* brand new error */
        if ((yyn = yypact[yystate]) > YYFLAG && yyn < YYLAST)
        {       register int x;

                for (x = yyn>0? yyn: 0; x < YYLAST; ++x)
                        if (yychk[yyact[x]] == x - yyn
                            && x - yyn != YYERRCODE)
                                yyerror(0, yydisplay(x-yyn));
        }
        yyerror(0,0);
yyerrlab:
        ++yynerrs;
        ...
```

```
yyerror(0,t)
```

```
yyerror(0,0)
```

```
/*
 *      yyerror() -- [detailed] error message for yyparse()
 */

#include <stdio.h>

FILE * yyerfp = stdout;                 /* error stream */

                                        /*VARARGS1*/
yyerror(s,t)                            /* "message" or 0,"token" */
        register char * s, * t;
{       extern int yynerrs;             /* total number of errors */
        static int list = 0;            /* sequential calls */

        if (s || ! list)                /* header necessary?? */
        {       fprintf(yyerfp, "[error %d] ", yynerrs+1);
                yywhere();
                if (s)                  /* simple message?? */
                {       fputs(s, yyerfp);
```

```
                putc('\n', yyerfp);
                return;
        }
        if (t)          /* first token?? */
        {       fputs("expecting: ", yyerfp);
                fputs(t, yyerfp);
                list = 1;
                return;
        }
                        /* no tokens acceptable */
        fputs("syntax error\n", yyerfp);
        return;
    }
    if (t)              /* subsequent token?? */
    {       putc(' ', yyerfp);
            fputs(t, yyerfp);
            return;
    }
                        /* end of enhanced message */
    putc('\n', yyerfp);
    list = 0;
}
```

```
#include <ctype.h>
#define DIM(x)  (sizeof x / sizeof x[0])

static char * yydisplay(ch)
        register int ch;
{       static char buf[15];
        static char * token[] = {
#include "y.tok.h"              /* token names */
                0 };

        switch(ch) {
        case 0:
                return "[end of file]";
        case YYERRCODE:
                return "[error]";
        case '\b':
                return "'\\b'";
        case '\f':
                return "'\\f'";
        case '\n':
                return "'\\n'";
        case '\r':
                return "'\\r'";
        case '\t':
```

```
                return "'\\t'";
        }
        if (ch > 256 && ch < 256 + DIM(token))
                return token[ch - 257];
        if (isascii(ch) && isprint(ch))
                sprintf(buf, "'%c'", ch);
        else if (ch < 256)
                sprintf(buf, "char %04.3o", ch);
        else
                sprintf(buf, "token %d", ch);
        return buf;
}
```

```
grep '^#.*define' y.tab.h \
| sed 's/^# define \([^ ]*\) [^ ]*$/    "\1",/'  > y.tok.h
```

```
cc -c main.c cpp.c yywhere.c yyerror.c
ar r colib main.o cpp.o yywhere.o yyerror.o
ranlib colib
```

```
#ifdef  YYDEBUG
int yydebug = 0;
#endif
int yychar;     /* current input token number */

yyparse()
{       register short yystate;
        ...
        yychar = -1;
```

```
yystack:        /* put a state and value onto the stack */

#ifdef  YYDEBUG
        if (yydebug)
                printf("state %d, char 0%o\n", yystate, yychar);
#endif

        ...
        if (yychar < 0)         /* lookahead available? */
                if ((yychar = yylex()) < 0)
                        yychar = 0;     /* end of file */
        ...
        if ( /* valid shift */ )
        {       ...
                yychar = -1;
                goto yystack;
        }
        ...
```

```
static yyyylex()
{
        if (yychar < 0)
        {       if ((yychar = yylex ()) < 0)
                        yychar = 0;
#ifdef YYDEBUG
        if (yydebug)
                printf("[yydebug] reading %s\n",
                        yydisplay(yychar));
#endif
        }
}
```

```
if test ! -d $OD
then    mkdir $OD
fi

for i in $ID/*
do      file=`basename $i`
        "$@" $i >$OD/\&$file
        echo end of task code $? >>$OD/\&$file
        if test -r $OD/$file
        then    if cmp -s $OD/\&$file $OD/$file
                then    echo "$i $file unchanged"
                        rm $OD/\&$file
                else    echo "$i $file changed: <new >old"
                        diff $OD/\&$file $OD/$file
                        mv $OD/$file $OD/${file}_old
                        mv $OD/\&$file $OD/$file
                fi
        else    echo "$i $file created"
                mv $OD/\&$file $OD/$file
        fi
done
exit 0
```

```
"$@" <$i >$OD/\&$file
```

References

[Aho74] A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys,* vol. 6, no. 2, pp. 99-124, June 1974.

[Aho77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

[Bau76] F. L. Bauer and J. Eickel (ed.), *Compiler Construction: An Advanced Course,* Springer, Berlin, 1974, 1976.

[Gra79] S. L. Graham, C. B. Haley, and W. N. Joy, "Practical LR error recovery," *SIGPLAN Notices,* Aug 1979.

[Gri71] D. Cries, *Compiler Construction for Digital Computers,* Wiley, New York, 1971.

[Jen75] K. Jensen and N Wirth, *Pascal: User Manual* and *Report,* Springer, Berlin, 1975.

[Joh78] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," in [Ker78a].

[Ker78a] B. W. Kernighan and M. D. McIlroy, *UNIX Programmer's Manual,* Bell Laboratories, 1978. Seventh Edition.

[Ker78b] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Les78a] M. E. Lesk, "Typing Documents on the UNIX System: Using the -ms Macros with Trod and Nroff," in [Ker78a].

[Les78b] M. E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator," in [Ker78a].

[Nau63] P. Naur (ed.), "Revised Report on the Algorithmic Language Algol 60," *CACM,* pp. 1-17, Jan. 1963.

[Wic73] B. A. Wichman, "The Definition of Comments in Programming Languages," NPL Report NAC-34, National Physics Laboratory, Division of Numerical Analysis and Computing, Teddington, England, May 1973.

[Wij75] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised Report on the Algorithmic Language Algol 68," *Acta Informatica,* pp. 1-236, 1, 1975.

[Wir77] N. Wirth, *Compilerbau,* B. G. Teubner, Stuttgart, 1977.

[Wir82] N. Wirth, Programming in *Modula-2,* Springer, Berlin, 1982.

INTRODUCTION TO COMPILER CONSTRUCTION WITH UNIX

Axel T. Schreiner • H[illegible]"e Friedman

The authors designed this practical book as a case study of two powerful, yet easy-to-use tools in the UNIX system. They show you how to create a compiler using generators such as *yacc* (LALR(1) parser generator) and *lex* (regular expression-based lexical analyzer generator).

SampleC, a simple subset of *C*, is defined and used as an example of compiler development. The implementation is intended to produce less than optimal object code, and suggestions for improvements to the code and extensions to the language provide problems in several chapters.

This tutorial shows how to get a simple, easily modifiable implementation quickly and reliably. It also helps the reader to learn practical details of the components of a compiler and the interfaces between them. The book is a short exposition preceding detailed algorithm studies in compiler construction and offers a description of how to employ productively the generators described.

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

ISBN 0-13-474396-2