Compilers for Algorithmic Languages
Second Edition

University of Kentucky CS 441G
Fall, 2005

Craig C. Douglas

Craig.Douglas@yale.edu
514H RMB (Robotics)
+1-859-257-2438

Modules of a Compiler

1. Symbol table
2. Lexical analysis
3. Parsing
4. Optimize
    a. Local
    b. Global
5. Code generation
6. Assembly
7. Loader/linker
8. Errors

Symbol Table

This is a data structure that holds information about symbols discovered by the compiler, e.g.,

- Names
  - Value
  - Type
  - Size and shape
  - Initial value (if any)
  - Scope (local, global, mixed)
- Constants:
  - Numbers
  - Strings
- Functions
  - Value
  - Scope (local, global, mixed)
  - Hints as to what it is and form
- Externals
  - Name
  - Hints as to what it is and form
- …

The table is usually a complicated data structure that is organized so that both the parser and code generator can easily access the information. The scoping information is particularly important to code optimization.

The design of a symbol table is usually done with personal taste involved. The implementation method used is frequently tied to the expected complexity of programs that will use it. Both have pluses and minuses.

Hashing, trees, and linked lists are common. Combinations and spaghetti coding structures are really common in commercial compilers. Part has to do with turnover of employees.

You are free to be creative, but plan for flexibility and expandability as you write first a lexer, then a parser, then a code generator.

Lexical Analysis

Recognizes the "words" of a programming language. Typically,

1. *Names for variables*: usually a letter followed by letters, digits, or underscores.
2. *Operators*: +, -, if, //, :=, !, ;
3. *Numbers*:
   a. Integers: strings of digits
   b. Reals: Fraction + Exponent
   c. Complex: two reals
4. *Character vectors* or *strings*: surrounded by quotes
5. *Space*: usually used as a delimiter and thrown away.

We usually divide our alphabet into classes:

1. Letters	A-Za-z
2. Operators	+-*/?.#%&!: …
3. Digits	0-9
4. Quote marks	‘ “ `
5. Space	

Alphabets: ASCII, EBCDIC, APL, Unicode, …

If the character set is small enough (e.g., ASCII), we can determine the class of a character by indexing a table:

```
Char Class[128] =
      { 0, …, 1, …, 5, … };

Class[‘A’] = 1;
Class[‘0’] = 3;
Class[‘\t’] = 5;
```

Then decide what to do for the next character by its class. Languages with associative memory constructs (e.g., Perl, Snobol, or ICON) can do this using built in operators trivially.

Of course, use a symbolic value for 1, 2, …, 5.

Standard coding style:

```
c = getchar();
t = Class[c];
switch (t) {
    1. Letter: Starting a name; break
    2. Operator: This character is the operator;
       break
    3. Digit: Starting a number; break
    4. Quote: Starting a string; break
    5. Space: Throw it away; break
}
```

What really happens here?

Case 2 (A single character operator)

a. Produce a token value and "type operator" as output
b. Start over at beginning of program with the next character

Case 1 (Word or variable name)

We really want to keep reading characters of a name, form a complete name, and make certain that it is in a dictionary (i.e., symbol table). Now the character types take on a different significance.

```
L:  c = getchar();
    t = Class[c];
    switch (t) {
        cases 1 and 3: (Letter or Digit)
            name = name || c;
            goto L;
        default:
            Look up name in symbol table;
            Produce a new token;
            Put c back into input queue;
    }
```

Case 3 (Number, e.g., an integer)

Read successive digits and form number.
Enter number in a table of numbers (which may be the symbol table).

```
n = 0;
While ( t == 3 ) {
    n = n*10 + ( c – ‘0’ );
    c = getchar();
    t = Class[c];
}
```

Output token value and class of number;
Put last c back into input queue;

In many languages, constructing a number can be done using a built in function or library function.

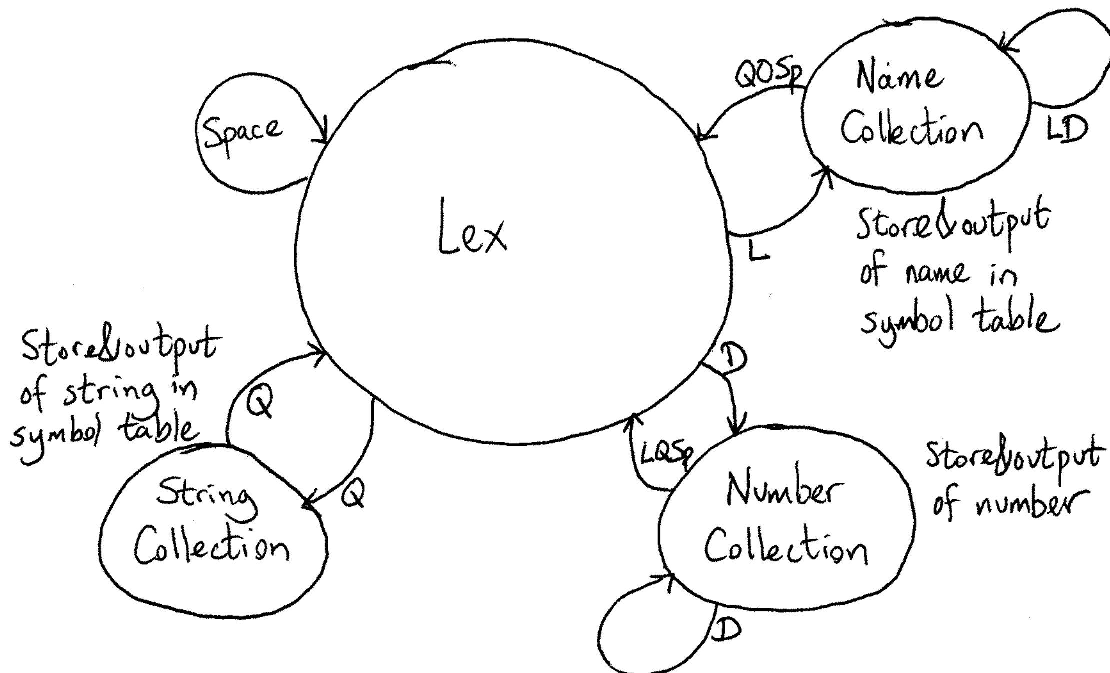
Throughout all of these programs we do the following:

1. Read a character and get its class
2. Branch to one of several small programs or code fragments
3. Loop back to one of several levels of read or branch calculation

We need to recognize that our process is in one of a number of states:

1. Initial: start of the program – ready for a new token
2. Identify an operator
3. Scanning a name: after first letter of name until its end
4. Scanning a number
5. Scanning a string

State transition diagram



This is easily stored and yields a simple program:

```
struct { int next, act, read; } tran[i][j];
     // Here i refers to the current state and j is
     // for the class of character
     // tran.next[i][j] = next state
     // tran.act[i][j] =  action to take
     // tran.read[i][j] = 1 if read new character

state = 1;
rd = 1;
while ( true ) {
     if ( rd ) { c = getchar(); t = Class[c]; }
```

```
        ac = tran.act[state][t];
        rd = tran.read[state][t];
        state = tran.next[state][t];
        switch ( ac ) {
              case 1:   name = c; break
              case 2:   name ||= c; break
              …
        }
}
```

Different Methods of Compiling

Multi-pass

Do lexical analysis: Text → *Lex* → Tokens
Do parsing: Tokens → *Parse* → Parse Tree
Do code generation: Parse Tree → *Gen* →
Assembly or object code

Each pass is done sequentially. Data is typically stored on disk.

Co-routines

Spawn 3 processes with pipes in between

Input file → *Lex* → *Parse* → *Gen* → Output file

A pipe is really just a queue.

As soon as process 1 (Lex) has written some tokens, process 2 (Parse) can wake up and read them, …

Conventional organization

Parse calls Lex as a subroutine to get another token.

```
int state, place;
switch (place) {
    case 1: initialize; break;
    case 2: … ; break;
    case 3: … ; break;
    …
}
```

Three pitfalls to watch out for:

1. All variables must be static, not on the stack.
2. Switch beginning may involve jumping into implied loops from previous call
3. Some compilers will optimize temporaries into registers or onto the stack

Format Systems for Describing Programming Languages

Context Free Grammars:

CFG Context Free Grammar

Letter → A | B | C | D (rules 1.1-1.4)
Name → Letter | Name Letter (rules 2.1-2.2)

BNF Backus-Naur Form
<Letter> ::= A | B | C | D
<Name> ::= <Letter> | <Name><Letter>

EBNF Extended BNF

[optional] and { repeat 0 or more times }

CFG starts at the top and tries to generate strings using the lower level definitions

BNF starts at the bottom of the rules and tries to make higher level symbols

EBNF can even describe itself, which BNF cannot do

grammar ::= rule { rule }

rule ::= NONTERMINAL '::=' [formulation]
{ '|' formulation }

formulation ::= symbol { symbol }

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

NONTERMINAL symbols are like variables in a programming language whereas TERMINAL symbols are like constants or key words. Space is usually ignored or used as a delimiter.

Ambiguities in Parsing ☹

Take BCD as input to the CFG example:

1. (init) Name →
2. (2.2) Name Letter →
3. (2.2) Name Letter Letter →
4. (1.3) Name C Letter →
5. (2.1) Letter C Letter →
6. (1.2) B C Letter →
7. (1.4) BCD

The other way around: Recognizer or parser approach

1. BCD (1.4) for D
2. BCD (1.2) for B
3. BCD (2.1) for B
4. BCD (1.3) for C
5. BCD (2.2) for BC
6. BCD (2.2) for BCD

Ambiguities should be avoided whenever possible ☺

Left and right recursion possible, too ☹^{2}

Letter → A | B| C | D
Name → Letter | Name Name

Pattern Matching

Unix users of ed, vi, vim, and similar editors might already know about one definition f regular expressions. See Chapter 6 of Lex and Yacc.

1. *Letters*, *digits*, and *some special characters* represent themselves
2. *Period* represents any character except a line feed
3. *Brackets* [] enclose a character class. Anything in the class matches unless [^] is used: then anything outside of the class matches, e.g., [^a]. Hyphens inside of the [] allow for ranges, e.g., [a-zA-Z].
4. * or ? ending a pattern means that a match can occur zero or as many times as possible; + ending a pattern means 1 or more matches
5. ^ before a pattern means match at the beginning of a line
6. $ ending a pattern means match at the end of a line
7. Escape mechanisms:
    a. \ before a character, e.g., \n for linefeed or \”
    b. “characters”
8. Multiple choice patterns: (pattern|pattern|…) matches one of the patterns.

Consider a C style comment:

```
/* single line comment */
/* … a multiline comment
            … */
/**/           (a null comment)
/*/ … */       (odd beginning of a comment)
```

A single line comment might be described by

“/*”.*“*/”

The other three cases fail with this regular expression. All four can be described by the opaque regular expression

“/*”“/”*([^*/]|[^*]“/”|“*”[^/])*“*”*“*/”

If you can explain this example in detail to someone else, you have mastered regular expressions far exceeding anything remotely reasonable.

How is an expression like this produced? Take one definition that is simple and make a regular expression. Then add a slightly harder case and modify the expression. Repeat until all of the cases are handled. Always do regression testing to make certain it is still correct for all cases, too.

Lex Programs

Input consists of up to three parts:

```
first part
%%
pattern         action
…
%%
third part
```

The first part is optional and contains lines controlling certain internal to lex table sizes, definitions for text replacement, and global C code within %{ and %} lines:

```
%{
C code
%}
```

The third part and its separator are optional, too. This is for C code that is taken as is.

The second part is quite line oriented. It starts at the first character and extends to the first non-escaped white space. Then an action appears after the white space. The longest expression that can be matched is used by lex. One line of C code can follow the

action, though multiple lines can be enclosed in brackets {}. There are no comments in Lex unless they are buried in the C code after the action.

Example: line numbering

```
%{
/* line numbering */
%}

%%

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

If this is stored in exuc.l, then it is compiled using

```
lex exuc.l
gcc lex.yy.c –ll –o exuc
```

The –ll is required and references the lex library. It has a default main program that just calls yylex().

Lex has some global variables that are useful:

yytext	character vector with the match
yyleng	integer length of yytext
yylineno	integer input line number
yylval	subvalue associated with yytext

Example: word count

```
%{
/* 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);
}
```

Grammar for Lexical Analysis

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

%%

"=="                                return token(EQ);
"<="                                return token(LE);

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

{digit}+                            return number();

{white_space}+

{other}                             return yytext[0];

%%
```

C functions for processing names and numbers.

This can be extended to cover a large subset of C and C++ fairly easily.

Screening for keywords

Normally, the number of keywords in a language is relatively small, but even a 15-20 keywords makes for a large transition table.

```
#include “tokens.h”

char *keywords[] = {
        “int”, “char”, “double”, …, “” };
int tokens[] = { INT, CHAR, DOUBLE, …, 0 };

int name( char *check ) {
        int i;
        for( i = 0; tokens[i]; i++ )
                if ( strcmp(check,keywords[i]) == 0 )
                        return tokens[i];
        return IDENTIFIER;
}
```

If there are many keywords, a faster search algorithm is justifiable. A binary search or hashing method can be substituted. The key is to not store the special words in the symbol table unless a fast search method is used.

This trick also works with operators.

When ambiguity is a ☺ in lex

Rules are usually highly ambiguous in lex, which resolves them using two rules:

1. lex always chooses the pattern representing the longest string match possible.
2. If multiple patterns represent the same string, the first one in the list is chosen.

Consider the two rules

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

Then

```
integer   matches the second rule
int       matches the first rule
```

Always put specific rules first followed by general rules. Having too many specific rules will produce huge transition tables.

Conflicts are frequently encountered in lex. The order of the rules can take bizarre forms that are completely illogical at first glance (or even the seventeenth glance).

Lexing complicted numbers

Integers are easy, but adding the full definition of a floating point number (either real or complex) is much harder. Using p. 23's definitions, we have

```
{digit}+
{digit}*"."{digit}+d("+" |"-"){digit}+
{digit}+"."{digit}*d("+" |"-"){digit}+
{digit}*"."{digit}+d{digit}+
{digit}+"."{digit}*d{digit}+
{digit}*"."{digit}+
{digit}+"."{digit}*
```

The actions are pretty simple: first translate the number into either an integer or a floating point number, e.g.,

```
atoi( yytext ) or atof( yytext );
```

Then enter the number in the symbol table.

Consider 1. 23d-4 or .000123. Note which rule applies to each.

Actually defining numbers using a small parser makes a lot of sense. This is commonly done using BNF.

BNF for numbers

Consider defining either integers, reals, or complex numbers. Only digits, +, -, d, (, and) are involved…

integer :	digit |
	integer digit
exponent :	d [+-] integer | d integer
mantissa :	integer "." integer |
	integer "." |
	"." integer
real :	mantissa exponent |
	mantissa
complex :	"(" real "," real ")"
number :	integer | real | complex

Add a collection of actions for the parser. Note that it is not until the number is constructed that the actual type is known. Hence, a constant can start as an integer and get promoted to either real or complex. We actually have to keep the integer parts as strings until we can determine that they are not to the right of a decimal point (consider .0032 versus .32).

Note that numbers are usually constructed in the lexer, not the parser. However, there is nothing to stop the lexer from using a parser to construct numbers.