Lexical Analysis - Part 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- What is lexical analysis? (covered in part 1)
- Why should LA be separated from syntax analysis? (covered in part 1)
- Tokens, patterns, and lexemes (covered in part 1)
- Difficulties in lexical analysis (covered in part 1)
- Recognition of tokens - finite automata and transition diagrams (covered in part 2)
- Specification of tokens - regular expressions and regular definitions (covered in part 2)
- LEX - A Lexical Analyzer Generator
Transition diagrams are generalized DFAs with the following differences:

- Edges may be labelled by a symbol, a set of symbols, or a regular definition.
- Some accepting states may be indicated as *retracting states*, indicating that the lexeme does not include the symbol that brought us to the accepting state.
- Each accepting state has an action attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value.

Transition diagrams are not meant for machine translation but only for manual translation.
TOKEN gettoken() {
    TOKEN mytoken; char c;
    while(1) {
        switch (state) {
        /* recognize reserved words and identifiers */
            case 0: c = nextchar(); if (letter(c))
                state = 1; else state = failure();
                break;
            case 1: c = nextchar();
                if (letter(c) || digit(c))
                    state = 1; else state = 2; break;
            case 2: retract(1);
                mytoken.token = search_token();
                if (mytoken.token == IDENTIFIER)
                    mytoken.value = get_id_string();
                return(mytoken);
        }
    }
}
Transition Diagram for Identifiers and Reserved Words

---

**Transition Diagram**

- **Start State (0)**: When a letter is encountered.
- **State 1**: Transition to state 1 on a letter, then to state 2 on other characters.
- **State 2**: Final state.

**Rules**:

- **letter** = [a-zA-Z_]
- **Identifier** = letter (letter | digit)*

**Return** (get_token_code(), name)

---

- ‘*’ indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)
/* recognize hexa and octal constants */
case 3: c = nextchar();
    if (c == '0') state = 4; break;
    else state = failure();
case 4: c = nextchar();
    if ((c == 'x') || (c == 'X'))
        state = 5; else if (digitoct(c))
        state = 9; else state = failure();
    break;

case 5: c = nextchar(); if (digithex(c))
    state = 6; else state = failure();
    break;
hex_const = 0 (x | X) dhex^+ (qualifier | ε)
oct_const = 0 doct^+ (qualifier | ε)
qualifier = u | U | I | L
dhex = [0-9A-F]
doct = [0-7]
case 6: c = nextchar(); if (digithex(c))
    state = 6; else if ((c == 'u') ||
    (c == 'U') || (c == 'l') ||
    (c == 'L')) state = 8;
else state = 7; break;

case 7: retract(1);
    /* fall through to case 8, to save coding */

case 8: mytoken.token = INT_CONST;
    mytoken.value = eval_hex_num();
    return(mytoken);

case 9: c = nextchar(); if (digitoct(c))
    state = 9; else if ((c == 'u') ||
    (c == 'U') || (c == 'l') || (c == 'L'))
    state = 11; else state = 10; break;
case 10: retract(1);
/* fall through to case 11, to save coding */
case 11: mytoken.token = INT_CONST;
        mytoken.value = eval_oct_num();
    return(mytoken);
Transition Diagrams for Integer Constants

\[ \text{int\_const} = \text{digit}^+ (\text{qualifier} \mid \varepsilon) \]
\[ \text{qualifier} = \text{u} \mid \text{U} \mid \text{l} \mid \text{L} \]
\[ \text{digit} = [0-9] \]
/* recognize integer constants */
case 12: c = nextchar(); if (digit(c))
    state = 13; else state = failure();
case 13: c = nextchar(); if (digit(c))
    state = 13; else if ((c == 'u') ||
    (c == 'U') || (c == 'l') || (c == 'L'))
    state = 15; else state = 14; break;
case 14: retract(1);
/* fall through to case 15, to save coding */
case 15: mytoken.token = INT_CONST;
    mytoken.value = eval_int_num();
    return(mytoken);
default: recover();
}
Different transition diagrams must be combined appropriately to yield an LA

- Combining TDs is not trivial
- It is possible to try different transition diagrams one after another
- For example, TDs for reserved words, constants, identifiers, and operators could be tried in that order
- However, this does not use the “longest match” characteristic (\textit{thenext} would be an identifier, and not \textit{reserved word} \textit{then} followed by identifier \textit{ext})
- To find the longest match, all TDs must be tried and the longest match must be used

Using LEX to generate a lexical analyzer makes it easy for the compiler writer
LEX has a language for describing regular expressions
It generates a pattern matcher for the regular expression specifications provided to it as input
General structure of a LEX program
{definitions} – Optional
%%
{rules} – Essential
%%
{user subroutines} – Essential
Commands to create an LA
- lex ex.l – creates a C-program lex.yy.c
- gcc -o ex.o lex.yy.c – produces ex.o
- ex.o is a *lexical analyzer*, that carves tokens from its input
/* LEX specification for the Example */

[A-Z]+ {ECHO; printf("\n");}
. | \n ;

yywrap() {}
main() {yylex();}

/* Input */
weevWEUFWIGhHkkH
sdcwehSDWEhTkFLksewT

/* Output */
WEUFWIG
H
SDWE
T
T
Definitions Section contains definitions and included code

- Definitions are like macros and have the following form:
  **name translation**

  ```
  digit [0-9]
  number {digit} {digit}*
  ```

- Included code is all code included between `%%{` and `%}`

  ```
  %{
     float number; int count=0;
  %}
  ```
Contains patterns and C-code
A line starting with white space or material enclosed in %{ and %} is C-code
A line starting with anything else is a pattern line
Pattern lines contain a pattern followed by some white space and C-code
\{pattern\} \{action (C-code)\}
C-code lines are copied verbatim to the the generated C-file
Patterns are translated into NFA which are then converted into DFA, optimized, and stored in the form of a table and a driver routine
The action associated with a pattern is executed when the DFA recognizes a string corresponding to that pattern and reaches a final state
Examples of strings: integer a57d hello

Operators:
" \ [ ] ^ - ? . * + | ( ) $ { } % <>
\ can be used as an escape character as in C

Character classes: enclosed in [ and ]
Only \, -, and ^ are special inside [ ]. All other operators are irrelevant inside [ ]
Examples:

[-+][0-9]+ ---> (-|+)(0|1|2|3|4|5|6|7|8|9)+
[a-d][0-4][A-C] ---> a|b|c|d|0|1|2|3|4|A|B|C
[^abc] ---> all char except a, b, or c,
    including special and control char
[+\-][0-5]+ ---> (+|-)(0|1|2|3|4|5)+
[^a-zA-Z] ---> all char which are not letters
Operators - Details

- **operator**: matches any character except newline
- **? operator**: used to implement $\epsilon$ option
  
  $ab?c$ stands for $a(b \mid \epsilon)c$

**Repetition, alternation, and grouping:**

$$(ab \mid cd^+)?(ef)^* \rightarrow (ab \mid c(d)^+ \mid \epsilon)(ef)^*$$

**Context sensitivity**: /, ^, $, are context-sensitive operators

- ^: If the first char of an expression is ^, then that expression is matched only at the beginning of a line. Holds only outside [ ] operator
- $: If the last char of an expression is $, then that expression is matched only at the end of a line
- \/: Look ahead operator, indicates trailing context

  ^ab ---> line beginning with ab  
  ab$  ---> line ending with ab (same as ab/\n)  
  DO/({letter}|{digit})* = ({letter}|{digit})*,
LEX Actions

- Default action is to copy input to output, those characters which are unmatched
- We need to provide patterns to catch characters
- `yytext`: contains the text matched against a pattern copying `yytext` can be done by the action `ECHO`
- `yyleng`: provides the number of characters matched
- LEX always tries the rules in the order written down and the longest match is preferred

```plaintext
integer    action1;
[a-z]+     action2;
```

The input `integers` will match the second pattern
```c
/* Input */
weevWEUFWIGHkkH
sdcewhSDFWEhTkFksewT

/* Output */
WEUFWIG
H
H
SDWE
T
FL
T
```
%%
^[ ]*\n
\n {ECHO; yylineno++;}
.* {printf("%d\t%s", yylineno, yytext);}
%

yywrap(){}
main(){ yylineno = 1; yylex(); }
/* Input and Output */

kurtrtotr
dvure

123456789
euhoyo854
shacg345845nkfg

1 kurtrtotr
2 dvure
3 123456789
4 euhoyo854
5 shacg345845nkfg
.lex

{%
FILE *declfile;
%
}

blanks \s*\t\s*\nletter [a-z]\n digit [0-9]\n id \s*({letter}|_)(\s*{letter}|{digit}|_)*\n number \s*{digit}+\n arraydeclpart \s*{id}"["\s*{number}"]\s*"\n declpart \s*({arraydeclpart}|{id})\n decllist \s*({declpart}\s*{blanks}"","\s*{blanks})\s*\n \s*{blanks}\s*{declpart}\s*{blanks}\n declaration \s*\(("int")\s*|\s*("float")\s*\)\s*{blanks}\n \s*{decllist}\s*{blanks};\n
LEX Example 3: EX-3.lex

Y.N. Srikant
Lexical Analysis - Part 3
%%
{declaration} fprintf(declfile,"%s\n",yytext);
%%

yywrap(){
fclose(declfile);
}
main(){
declfile = fopen("declfile","w");
yylex();
}
LEX Example 3: Input, Output, Rejection

wjwkfb1webg2; int ab, float cd, ef;
ewl2efo24hg2jhrto;ty;
int ght, asjhew[37], fuir, gj[45]; sdkvbwrkb;
float ire, dehj[80];
sdvjkjkw

=============================================
float cd, ef;
int ght, asjhew[37], fuir, gj[45];
float ire, dehj[80];
=============================================

wjwkfb1webg2; int ab,
ewl2efo24hg2jhrto;ty;
sdkvbwrkb;
sdvjkjkw

Y.N. Srikant  Lexical Analysis - Part 3
LEX Example 4: Identifiers, Reserved Words, and Constants (id-hex-oct-int-1.lex)

{%
int hex = 0; int oct = 0; int regular =0;
%
}
letter        [a-zA-Z_] 
digit         [0-9] 
digits        {digit}+ 
digit_oct     [0-7] 
digit_hex     [0-9A-F] 
int_qualifier [uU1lL] 
blanks        [ \t]+ 
identifier    {letter}({letter}|{digit})* 
integer       {digits}{int_qualifier}? 
hex_const     0[xX]{digit_hex}+{int_qualifier}? 
oct_const     0{digit_oct}+{int_qualifier}?
LEX Example 4: (contd.)

```c

if {printf("reserved word:%s\n",yytext);}
else {printf("reserved word:%s\n",yytext);}
while {printf("reserved word:%s\n",yytext);}
switch {printf("reserved word:%s\n",yytext);}
{identifier} {printf("identifier :%s\n",yytext);}
{hex_const} {sscanf(yytext,"%i",&hex);
    printf("hex constant: %s = %i\n",yytext,hex);}
{oct_const} {sscanf(yytext,"%i",&oct);
    printf("oct constant: %s = %i\n",yytext,oct);}
{integer} {sscanf(yytext,"%i",&regular);
    printf("integer : %s = %i\n",yytext, regular);}
.

yywrap(){}

int main(){yylex();}
```

Y.N. Srikant  
Lexical Analysis - Part 3
uorme while
0345LA 456UB 0x7861HABC
b0x34

identifier : uorme
reserved word : while
oct constant : 0345L = 229
identifier : A
integer : 456U = 456
identifier : B
hex constant : 0x7861 = 1926
identifier : HABC
identifier : b0x34
digits \[1-9]+  
exp ([Ee](\+|\-){digits})  
blanks [ \t\n]+  
float_qual [fF1L]  

{digits}{exp}{float_qual}<?,{blanks}  
{printf("float no fraction:%s\n",yytext);}  
[0-9]*\.{digits}{exp}?{float_qual}?<?,{blanks}  
{printf("float with optional  
integer part :%s\n",yytext);}  
{digits}\.[0-9]*{exp}?{float_qual}?<?,{blanks}  
{printf("float with  
optional fraction:%s\n",yytext);}  
.\n ;  

yywrap(){} int main(){yylex();}
LEX Example 5: Input and Output

123 345. 4565.3 675e-5 523.4e+2 98.1e5 234.3.4 345. .234E+09L 987E-6F 5432.E71

float with optional integer part : 4565.3
float no fraction: 675e-5
float with optional integer part : 523.4e+2
float with optional integer part : 98.1e5
float with optional integer part : 3.4
float with optional fraction: 345.
float with optional integer part : .234E+09L
float no fraction: 987E-6F
float with optional fraction: 5432.E71
number \[0-9]+\.?|\[0-9\]*\.\[0-9\]+\n
name \[A-Za-z]\[A-Za-z0-9\]*

[ ] {\/* skip blanks */}

{number} \{sscanf(yytext, "%lf", &yylval.dval); return NUMBER;\}

{name} \{struct symtab *sp = symlook(yytext); yylval.symp = sp; return NAME;\}

"++" \{return POSTPLUS;\}

"--" \{return POSTMINUS;\}

"$" \{return 0;\}

\n|. \{return yytext[0];\}