CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

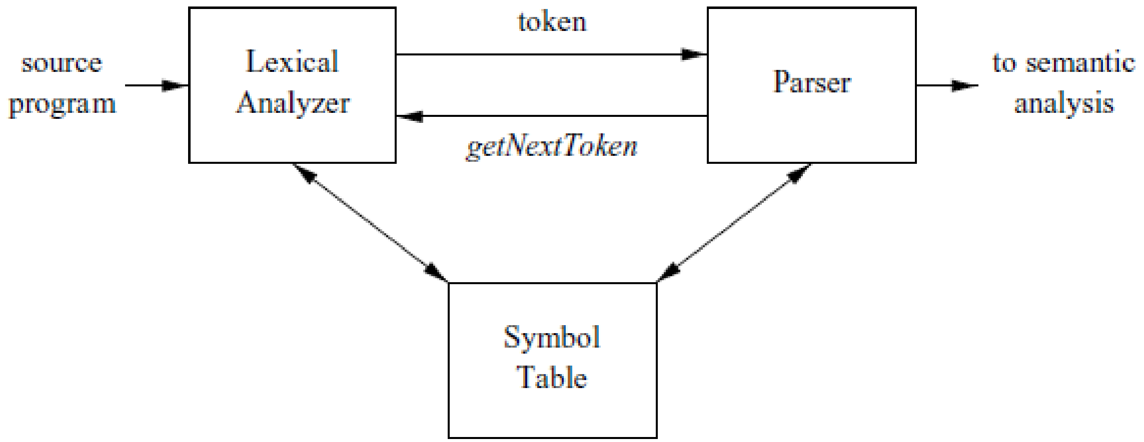
- **Recognition of Tokens (Ch 3.4)**
  - **Architecture of a Transition-Diagram-Based Lexical Analyzer (3.4.4)**
- **The Lexical-Analyzer Generator Lex (Ch 3.5)**
  - **Use of Lex (3.5.1)**
  - **Structure of Lex Programs (3.5.2)**
  - **Conflict Resolution in Lex (3.5.3) (bypassed)**
  - **The Lookahead Operator (3.5.4) (bypassed)**
  - **Finite Automata (3.6) (TBD, in Part6)**

Architecture of a Transition-Diagram-Based Lexical Analyzer

• Remember we first convert patterns into stylized flowcharts, called **transition diagrams**, in the construction of a Lexical Analyzer.

• Each state is represented by a piece of code

• A variable state holding the **number** of the current state for a transition diagram



• A switch based on the value of state takes us to code for each of the possible states

Architecture of a Transition-Diagram-Based Lexical Analyzer

- Let’s check the book, Fig. 3-13 and 3-18.
- The spec of a token in this case:
- <relop, attribute>

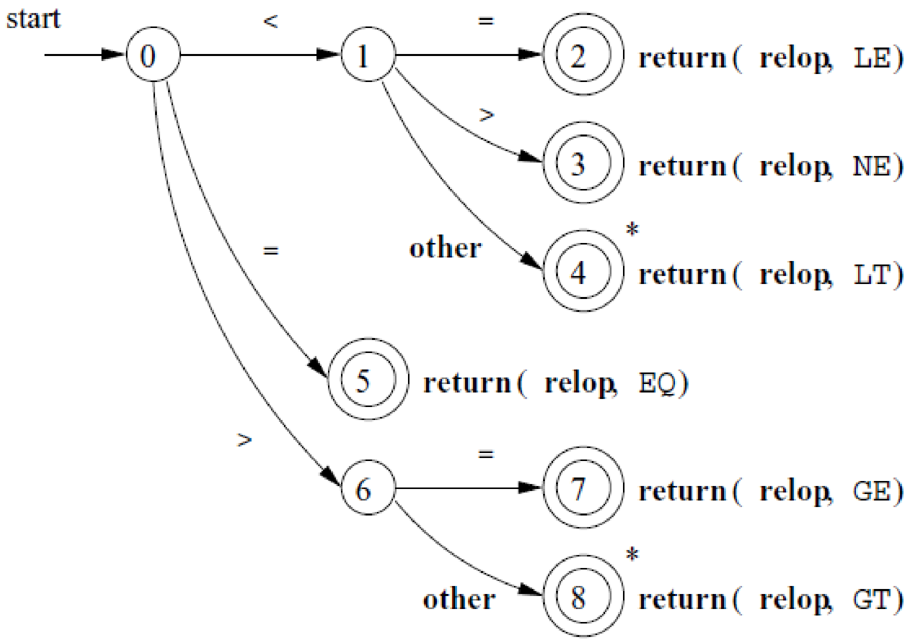


Figure 3.13: Transition diagram for **relop**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

Architecture of a Transition-Diagram-Based Lexical Analyzer

- Note that if the next input character is not one that can begin a comparison operator, then a function fail() is called
- If the fail() is called, it should **reset** the forward pointer to **lexemeBegin**, in order to allow another transition diagram to be applied to the true beginning of the unprocessed input.

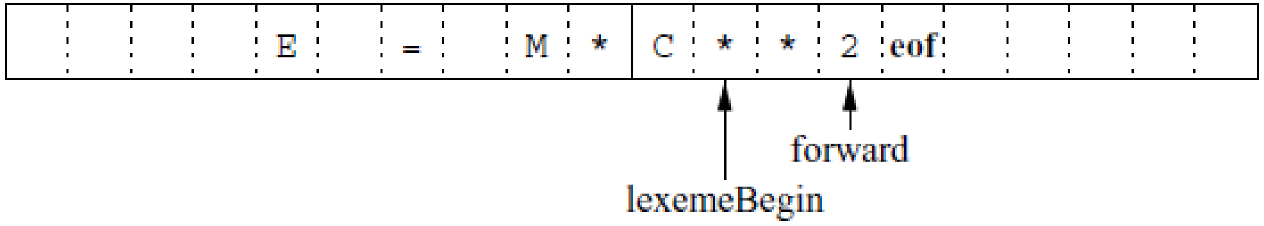


Figure 3.3: Using a pair of input buffers

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

Architecture of a Transition-Diagram-Based Lexical Analyzer

- When the fail() is called, it might then **change the value of state to be the start state** for another transition diagram, which will search for **another** (next) token.
- The state 8 bears a *****, we must **retract the input pointer one position** (i.e., **put c back on the input stream**).

The Lexical-Analyzer Generator Lex

- A tool called **Lex**, or in a more recent implementation **Flex**, that allows one to specify a lexical analyzer by **specifying regular expressions** to describe patterns for tokens.
- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler
- The Lex compiler can transform the input patterns into a transition diagram and generates code

Use of Lex

- An input file, which we call “lex.l”, is written in the Lex language and describes the lexical analyzer to be generated
- Lex compiler can:

lex.l→lex.yy.c

- lex.yy.c is compiled by C

Compiler into “a.out”

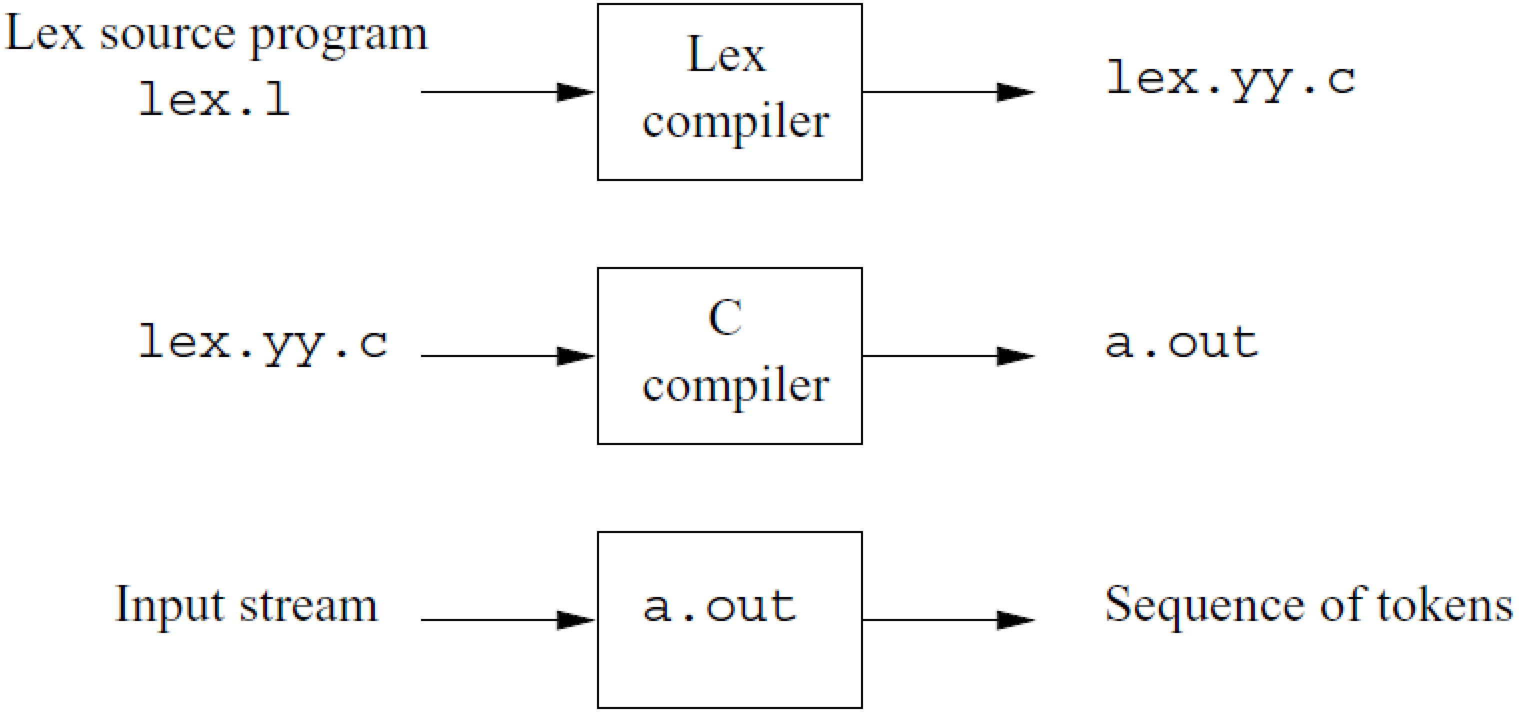


Figure 3.22: Creating a lexical analyzer with Lex

Structure of Lex Programs

```
declarations
%%
translation rules
%%
auxiliary functions
```

- A Lex program has the following form: (on RHS)
- 1st part: The declarations section includes:

declarations of variables, manifest constants (ID declared to stand for a constant. i.e. name of a token), or regular definitions as we had seen in Section 3.3.4

- 2nd part: translation rule: Pattern {Action}

Each of the pattern in this part is a regular expression.

The actions are fragments of code, typically written in C

Structure of Lex Programs

- The 3rd section holds whatever additional functions are used in the actions.
- Those functions can be compiled separately and loaded with the lexical analyzer
- (See the next page for a completed example)

```
declarations
%%
translation rules
%%
auxiliary functions
```

Structure of Lex Programs

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Figure 3.23: Lex program for the tokens of Fig. 3.12

- One thing I want to point out is the “yylval”, see the next page for detail

Structure of Lex Programs

- The lexical analyzer returns a single value, the token name, to the parser, but uses the **shared**, **integer variable** *yylval* to pass additional information about the lexeme found, if needed.
- The attribute value it could be placed in a global variable *yylval* which is shared between the lexical analyzer (LA) and parser, thereby making it simple to return both the **name** and an **attribute** value of a token.