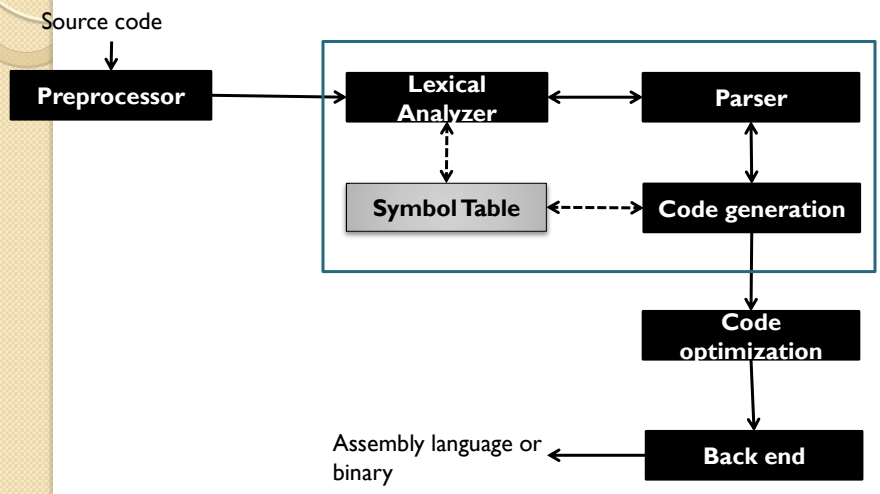
Compilers (CS30003)

Lecture 05

Pralay Mitra

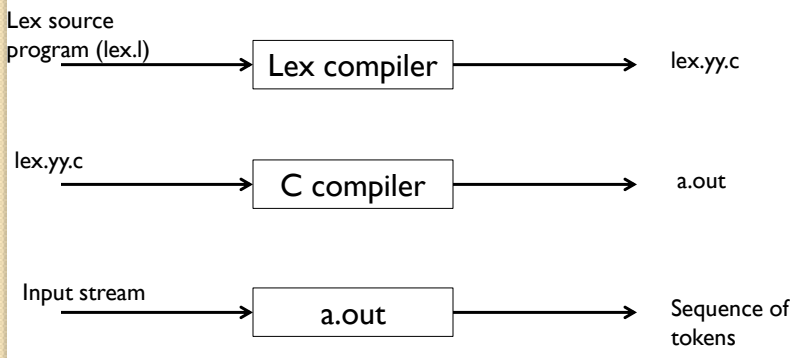
Autumn 2023-24

Four pass compiler



Autumn 2023-24

The Lexical Analyzer Generator



Structure of a Lex program

Declarations

%%

Translation rule

%%

Auxiliary functions

First flex program

$ flex firstProg.l

$ cc lex.yy.c –lfl

$ ./a.out

….

$

Ambiguous patterns

- Match the longest possible string every time the scanner matches input.
- Break the tie in favor of the pattern appears first in the program.

```
%%
"+"       { return ADD; }
"="       { return ASSIGN; }
"+="      { return ASSIGNADD; }
"<"       { return LT; }
"<="      { return LE; }
"if"      { return KEYWORDIF; }
"else"    { return KEYWORDELSE; }

[a-zA-Z_][a-zA-Z0-9_]*      { return IDENTIFIER; }
%%
```

I/P Character Stream	O/P Token Stream
{	<SPECIAL SYMBOL, {>
int x;	<KEYWORD, int> <ID, x> <PUNCTUATION, ;>
int y;	<KEYWORD, int> <ID, y> <PUNCTUATION, ;>
x = 2;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
y = 3;	<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
x = 5 + y * 4;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 5> <OPERATOR, +>
}	<ID, y> <OPERATOR, *> <INTEGER CONSTANT, 4> <PUNCTUATION, ;>
	<SPECIAL SYMBOL, }>

An example

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT         "int"
ID          [a-z][a-z0-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]

%%
{INT}       { printf("<KEYWORD, int>\n"); /* Keyword Rule */ }
{ID}        { printf("<ID, %s>\n", yytext); /* Identifier Rule */}
"+"         { printf("<OPERATOR, +>\n"); /* Operator Rule */ }
"*"         { printf("<OPERATOR, *>\n"); /* Operator Rule */ }
"="         { printf("<OPERATOR, =>\n"); /* Operator Rule */ }
"{"         { printf("<SPECIAL SYMBOL, {>\n"); /* Scope Rule */ }
"}"         { printf("<SPECIAL SYMBOL, }>\n"); /* Scope Rule */ }
{PUNC}      { printf("<PUNCTUATION, ;>\n"); /* Statement Rule */ }
{CONST}     { printf("<INTEGER CONSTANT, %s>\n",yytext); /* Literal Rule */ }
{WS}        /* White-space Rule */ ;
%%
```

I/P Character Stream	O/P Token Stream
{	<SPECIAL SYMBOL, {>
int x;	<ID, int> <ID, x> <PUNCTUATION, ;>
int y;	<ID, int> <ID, y> <PUNCTUATION, ;>
x = 2;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
y = 3;	<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
x = 5 + y * 4;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 5> <OPERATOR, +>
}	<ID, y> <OPERATOR, *> <INTEGER CONSTANT, 4> <PUNCTUATION, ;>
	<SPECIAL SYMBOL, }>

An example???

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT         "int"
ID          [a-z][a-z0-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]

%%
{ID}        { printf("<ID, %s>\n", yytext); /* Identifier Rule */}
{INT}       { printf("<KEYWORD, "int">\n"); /* Keyword Rule */ }
"+"         { printf("<OPERATOR, +>\n"); /* Operator Rule */ }
"*"         { printf("<OPERATOR, *>\n"); /* Operator Rule */ }
"="         { printf("<OPERATOR, =>\n"); /* Operator Rule */ }
"{"         { printf("<SPECIAL SYMBOL, {>\n"); /* Scope Rule */ }
"}"         { printf("<SPECIAL SYMBOL, }>\n"); /* Scope Rule */ }
{PUNC}      { printf("<PUNCTUATION, ;>\n"); /* Statement Rule */ }
{CONST}     { printf("<INTEGER CONSTANT, %s>\n",yytext); /* Literal Rule */ }
{WS}        /* White-space Rule */ ;
%%
```

Complete example

```
%{
#define INT         10
#define ID          11
#define PLUS        12
#define MULT        13
#define ASSIGN      14
#define LBRACE      15
#define RBRACE      16
#define CONST       17
#define SEMICOLON   18
%}

INT       "int"
ID        [a-z][a-z0-9]*
PUNC      [;]
CONST     [0-9]+
WS        [ \t\n]

%%
{INT}   { return INT; }
{ID}    { return ID; }
"+"     { return PLUS; }
"*"     { return MULT; }
"="     { return ASSIGN; }
"{"     { return LBRACE; }
"}"     { return RBRACE; }
{PUNC}  { return SEMICOLON; }
{CONST} { return CONST; }
{WS}    {/* Ignore
           whitespace */}
%%
```

```
main() { int token;
    while (token = yylex()) {
        switch (token) {
            case INT: printf("<KEYWORD, %d, %s>\n",
                token, yytext); break;
            case ID: printf("<IDENTIFIER, %d, %s>\n",
                token, yytext); break;
            case PLUS: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case MULT: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case ASSIGN: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case LBRACE: printf("<SPECIAL SYMBOL, %d, %s>\n",
                token, yytext); break;
            case RBRACE: printf("<SPECIAL SYMBOL, %d, %s>\n",
                token, yytext); break;
            case SEMICOLON: printf("<PUNCTUATION, %d, %s>\n",
                token, yytext); break;
            case CONST: printf("<INTEGER CONSTANT, %d, %s>\n",
                token, yytext); break;
        }
    }
}
```

Managing Symbol Table

```
%{
    struct symbol {
     char *name;
     struct ref *reflist;
    };
    struct ref {
     struct ref *next;
     char *filename;
     int flags;
     int lineno;
    };

    #define NHASH 100
    struct symbol symtab[NHASH];
    struct symbol *lookup(char *);
    void addref(int, char*, char*, int);
%}
```

Start condition in FLEX

Flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with <sc> will only be active when the scanner is in the start condition named sc. For example,

```
<STRING>[^"]*     { /* my comment */
                        …..
                  }
```

Will be active only when the scanner is in the STRING start condition, and

```
<INITIAL, STRING, QUOTE>\.        { /* handle an escape */
                                        ….
                                  }
```

Will be active only when the current start condition is either INITIAL, STRING, or QUOTE.

Start condition in FLEX

- **Declaration:** Declared in the definitions section of the input
- **BEGIN Action:** A start condition is activated using the BEGIN action. Until the next BEGIN action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive.
- **Inclusive Start Conditions:** Use unindented lines beginning with '\%s' followed by a list of names.
- **Exclusive Start Conditions:** Use unindented lines beginning with '\%x' followed by a list of names.

- A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the flex input. Because of this, exclusive start conditions make it easy to specify mini-scanners which scan portions of the input that are syntactically different from the rest (for example, comments).

Start condition in FLEX

The set of rules:

```
%s example
%%
<example>foo   do_something();
bar            something_else();
```

is equivalent to

```
%x example
%%
<example>foo             do_something();
<INITIAL,example>bar     something_else();
```

Without the <INITIAL,example> qualifier, the bar pattern in the second example wouldn't be active (that is, couldn't match) when in start condition example. If we just used <example> to qualify bar, though, then it would only be active in example and not in *INITIAL*, while in the first example it's active in both, because in the first example the example start condition is an inclusive (\%s) start condition.

Handling Comments

```
%x   comment
%%
int lines = 1;

"/*"        BEGIN(comment);

<comment>[^*\n]*          /* my comment */
<comment>"*"+[^*/\n]*     /* my comment */
<comment>\n               lines++;
<comment>"*"+"/"          BEGIN(INITIAL);
```