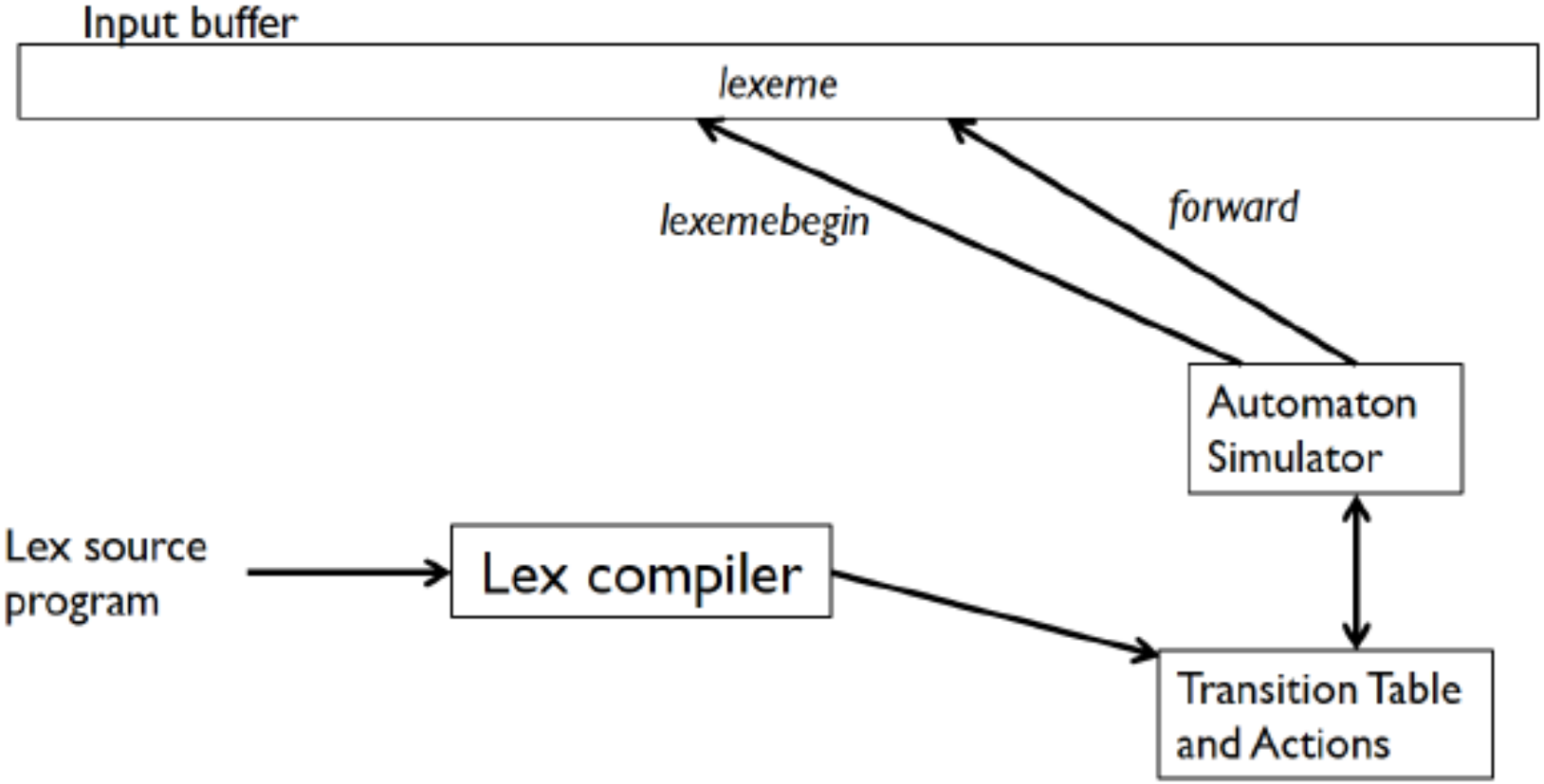
Compilers (CS31003)

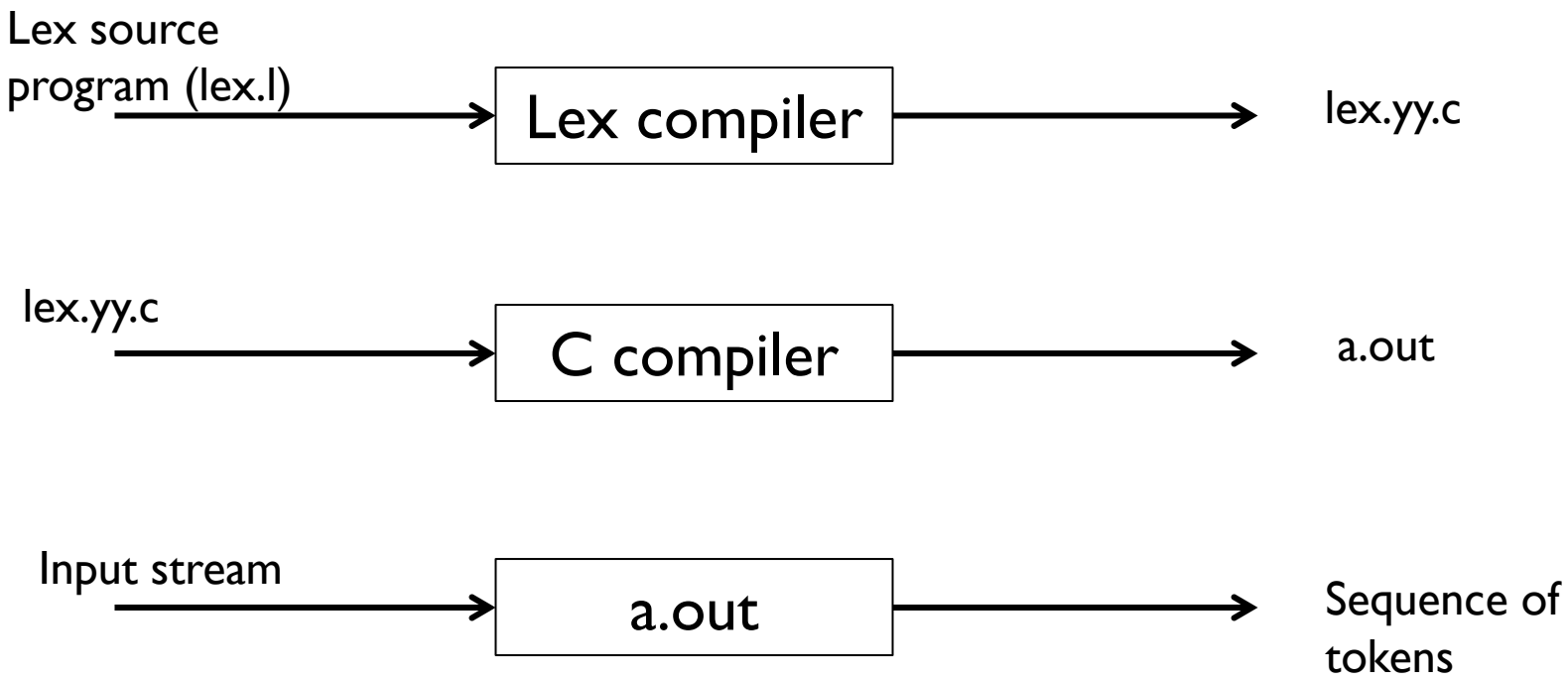
Lecture 04-05

Flex flow



Lex program → Transition table and actions → FA simulator

The Lexical Analyzer Generator



Structure of Flex Specs

Declarations

%%

Translation rule

%%

Auxiliary functions

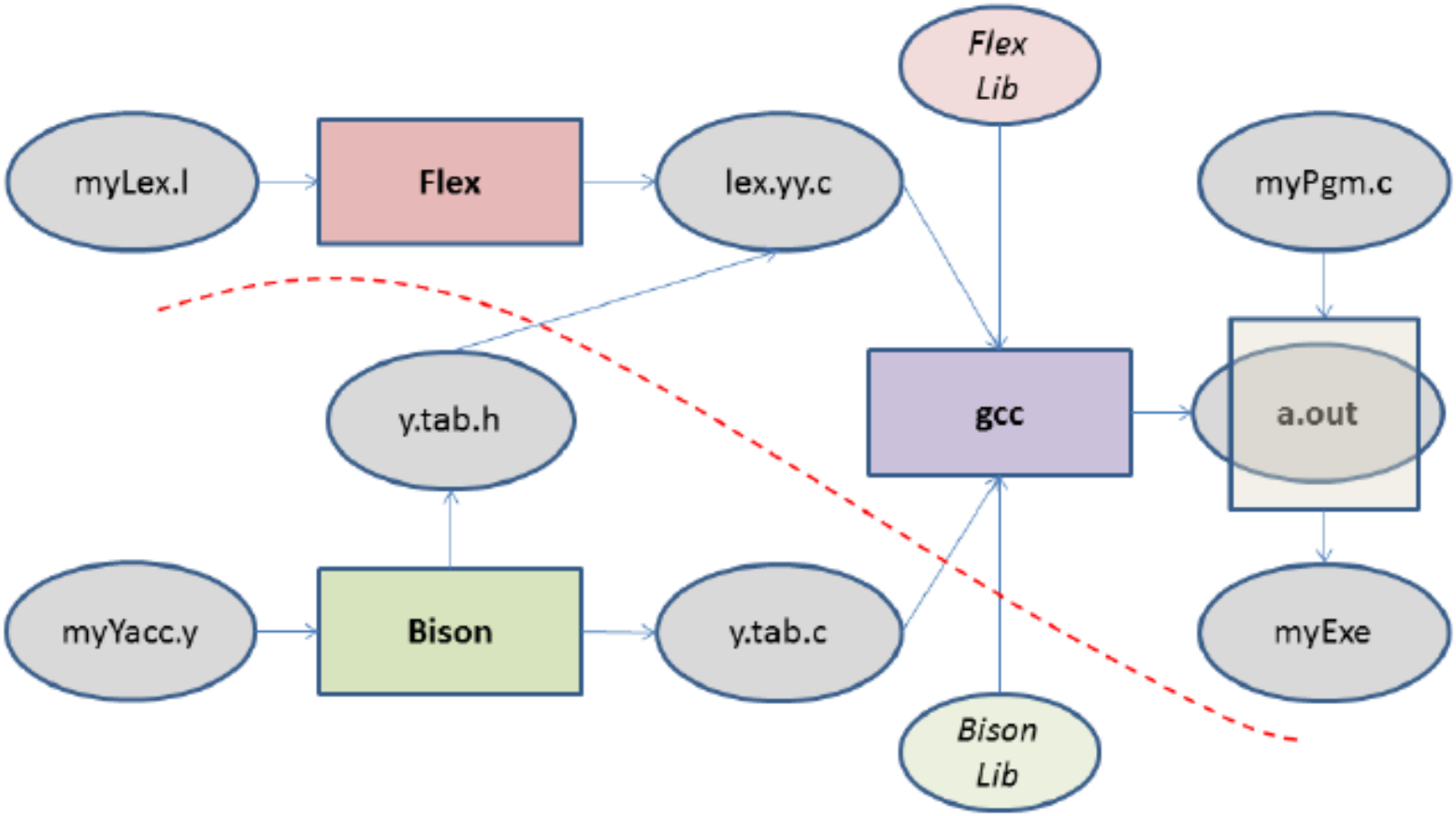
First Flex program

```
%{
    int chars=0;
    int words=0;
    int lines=0;
%}
%%
[a-zA-Z]+  {words++; chars+=strlen(yytext); }
\n                          {chars++; lines++;}
.                           {chars++;}
%%
main(int argc, char **argv)
{
    yylex();
    printf("%8d%8d%8d\n",lines,words,chars);
}
```

First Flex program

```
$ flex firstProg.l
$ cc lex.yy.c –lfl
$ ./a.out
….
$
```

Flex-Bison Flow



I/O in FLEX

```
main(int argc, char **argv)
{
    if(argc>1) {
            if(!(yyin=fopen(argv[1],"r"))  {
                    perror(argv[1]);
                    return (1);
            }
    }

    yylex();
    printf("%8d%8d%8d\n",linex,words,chars);
}
```

I/O in FLEX

```
for(i=1;i<argc;i++) {
      FILE *f=fopen(argv[i],”r”);
      if(!f) {
            perror(argv[i]);
            return (1);
       }
       yyrestart(f);
       yylex();
       fclose(f);
             /* More body */
}
```

Token recognizer

```
%%
“+”        { printf(“PLUS\n”); }
“-”        { printf(“MINUS\n”); }
“*”        { printf(“MULT\n”); }
“/”        { printf(“DIVIDE\n”); }
“|”        { printf(“ABS\n”); }
[0-9]+     { printf(“NUMBER %s\n”,yytext); }
\n         { printf(“NEWLINE\n”); }
[ \t]      { }
.          { printf(“UNKNOWN %s\n”,yytext); }
%%
```

12+34
9 9+34
9+99f

Token and values

- Token numbers are arbitrary (EOF is token 0).
- Bison assigns the token number starting at 258.

```
%%
“+”         { return ADD; }
“-”         { return SUB; }
“*”         { return MUL; }
“/”         { return DIV; }
“|”         { return ABS; }
[0-9]+      { yyval=atoi(yytext); return NUMBER; }
\n          { return EOL; }
[ \t]       { /* ignore whitespace */}
.           { printf(“UNKNOWN %s\n”,yytext); }
%%
```

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; }
%%
```

An example for Flex

- This is a simple block with declaration and expression statements
- We shall use this as a running example

```
{
    int x;
    int y;
    x = 2;
    y = 3;
    x = 5 + y * 4;
}
```

Structure of Flex Specs

Declarations

%%

Translation rule

%%

Auxiliary functions

Flex spec for our example

- C Declarations and definitions
- Definitions of Regular Expressions
- Definitions of Rules & Actions
- C functions

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT         "int"
ID          [a-z][a-zO-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]
/* Definitions of Rules \& Actions */
%%
{INT}       { printf("<KEYWORD, int>\n"); /* Keyword Rule */ }
{ID}        { printf("<ID, %s>\n", yytext); /* Identifier Rule & yytext points to lexeme */}
"+"         { 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 */ ;
%%
/* C functions */
main() { yylex(); /* Flex Engine */ }
```

Flex I/O for our example

- Every token is a doublet showing the token class and the specific token information
- The output is generated as one token per line. It has been rearranged here for better readability

I/P Character Stream

```
{
    int x;
    int y;
    x = 2;
    y = 3;
    x = 5 + y * 4;
}
```

O/P Token Stream

```
<SPECIAL SYMBOL, {>
<KEYWORD, int> <ID, x> <PUNCTUATION, ;>
<KEYWORD, int> <ID, y> <PUNCTUATION, ;>
<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
<ID, x> <OPERATOR, => <INTEGER CONSTANT, 5> <OPERATOR, +>
<ID, y> <OPERATOR, *> <INTEGER CONSTANT, 4> <PUNCTUATION, ;>
<SPECIAL SYMBOL, }>
```

Variables in Flex

yylex() Flex generated lexer driver
yyin File pointer to Flex input
yyout File pointer to Flex output
yytext Pointer to Lexeme
yyleng Length of the Lexeme

Regular Expressions - Basic

Expr.	Meaning
x	Character x
.	Any character except newline
[xyz]	Any characters amongst x, y or z.
[a-z]	Denotes any letter from a through z
[^0-9]	Stands for any character which is not a decimal digit, including new-line
\x	If x is an a, b, f, n, r, t, or v, then the ANSI-C interpretation of \x. Otherwise, a literal x (used to escape operators such as *)
\0	A NULL character
\num	Character with octal value num
\xnum	Character with hexadecimal value num
"string"	Match the literal string. For instance "/*" denotes the character / and then the character *, as opposed to /* denoting any number of slashes
<<EOF>>	Match the end-of-file

Regular Expressions - Operators

Expr.	Meaning
(r)	Match an r; parentheses are used to override precedence
rs	Match the regular expression r followed by the regular expression s. This is called *concatenation*
r|s	Match either an r or an s. This is called *alternation*
{*abbreviation*}	Match the expansion of the abbreviation definition. Instead of:

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

Use

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

Regular Expressions - Operators

Expr.	Meaning
	quantifiers
r*	zero or more r's
r+	one or more r's
r?	zero or one r's
r{[num]}	num times r
r{min,[max]}	Anywhere from min to max (defaulting to no bound) r's
r/s	Match an r but only if it is followed by an s. This type of pattern is called *trailing context*.
	For example: Distinguish DO1J=1,5 (a for loop where I runs from 1 to 5) from DO1J=1.5 (a definition/assignment of the floating variable DO1J to 1.5) in FORTRAN. Use
	DO/[A-Z0-9]*=[A-Z0-9]*
^r	Match an r at the beginning of a line
r$	Match an r at the end of a line

Wrong Flex specification

- Rules for ID and INT have been swapped.
- No keyword can be tokenized as keyword now.

```
%{
/* 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 */ ;
%%

main() {
    yylex(); /* Flex Engine */
}
```

Wrong Flex output

I/P Character Stream

```
{
    int x;
    int y;
    x = 2;
    y = 3;
    x = 5 + y * 4;
}
```

O/P Token Stream

```
<SPECIAL SYMBOL, {>
<ID, int> <ID, x> <PUNCTUATION, ;>
<ID, int> <ID, y> <PUNCTUATION, ;>
<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
<ID, x> <OPERATOR, => <INTEGER CONSTANT, 5> <OPERATOR, +>
<ID, y> <OPERATOR, *> <INTEGER CONSTANT, 4> <PUNCTUATION, ;>
<SPECIAL SYMBOL, }>
```

- Both `int`'s have been taken as ID!

Count Number of Lines - Flex Specs Another Example

```
/* C Declarations and definitions */
%{
    int charCount = 0, wordCount = 0, lineCount = 0;
%}

/* Definitions of Regular Expressions */
word   [^ \t\n]+                              /* A word is a seq. of char. w/o a white space */

/* Definitions of Rules \& Actions */
%%
{word}    { wordCount++; charCount += yyleng; /* Any character other than white space */ }
[\n]      { charCount++; lineCount++;          /* newline character */ }
.         { charCount++;                      /* space and tab characters */ }
%%

/* C functions */
main() {
    yylex();
    printf("Characters: %d Words: %d Lines %d\n",charCount, wordCount, lineCount);
}
```

Count Number of Lines - lex.yy.c

```
char *yytext;
int charCount = 0, wordCount = 0, lineCount = 0; /* C Declarations and definitions */
/* Definitions of Regular Expressions & Definitions of Rules & Actions */
int yylex (void) { /** The main scanner function which does all the work. */
// ...
    if ( ! (yy_start) ) (yy_start) = 1;    /* first start state */
    if ( ! yyin ) yyin = stdin;
    if ( ! yyout ) yyout = stdout;
// ...
    while ( 1 ) {        /* loops until end-of-file is reached */
// ..
     yy_current_state = (yy_start);
yy_match: // ...
yy_find_action: // ...
do_action:
        switch ( yy_act ) { /* beginning of action switch */
            case 0: /* must back up */ // ...
            case 1: { wordCount++; charCount += yyleng; } YY_BREAK
            case 2: { charCount++; lineCount++; } YY_BREAK
            case 3: { charCount++; } YY_BREAK
            case 4: ECHO; YY_BREAK
            case YY_STATE_EOF(INITIAL): yyterminate();
            case YY_END_OF_BUFFER:
            default: YY_FATAL_ERROR("fatal flex scanner internal error--no action found" );
        } /* end of action switch */
    } /* end of scanning one token */
} /* end of yylex */
main() { /* C functions */
    yylex();
    printf("Characters: %d Words: %d Lines %d\n",charCount, wordCount, lineCount);
}
```

Modes of Flex Operations

Flex can be used in two modes:

- Non-interactive: Call yylex() only once. It keeps spitting the tokens till the end-of-file is reached. So the actions on the rules do not have return and falls through in the switch in lex.yy.c.

  This is convenient for small specifications. But does not work well for large programs because:

  - Long stream of spitted tokens may need a further tokenization while processed by the parser
  - At times tokenization itself, or at least the information update in the actions for the rules, may need information from the parser (like pointer to the correctly scoped symbol table)

- Interactive: Repeatedly call yylex(). Every call returns one token (after taking the actions for the rule matched) that is consumed by the parser and yylex() is again called for the next token. This lets parser and lexer work hand-in-hand and also eases information interchange between the two.

Flex Specs (non-interactive) for our example

- C Declarations and definitions
- Definitions of Regular Expressions
- Definitions of Rules & Actions
- C functions

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT         "int"
ID          [a-z][a-z0-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]
/* Definitions of Rules \& Actions */
%%
{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 */ ;
%%
/* C functions */
main() { yylex(); /* Flex Engine */ }
```

Flex Specs (interactive) for our 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 */}
%%
```

Flex Specs (interactive) for our example

```
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;
        }
    }
}
```

– Input is taken from `stdin`. It can be changed by opening the file in `main()` and setting the file pointer to `yyin`.
– When the lexer will be integrated with the YACC generated parser, the `yyparse()` therein will call `yylex()` and the `main()` will call `yyparse()`.

Flex I/O (interactive) for our example

I/P Character Stream

```
{
    int x;
    int y;
    x = 2;
    y = 3;
    x = 5 + y * 4;
}
```

```
#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
```

O/P Token Stream

```
<SPECIAL SYMBOL, 15, {>
<KEYWORD, 10, int>
<IDENTIFIER, 11, x>
<PUNCTUATION, 18, ;>
<KEYWORD, 10, int>
<IDENTIFIER, 11, y>
<PUNCTUATION, 18, ;>
<IDENTIFIER, 11, x>
<OPERATOR, 14, =>
<INTEGER CONSTANT, 17, 2>
<PUNCTUATION, 18, ;>
<IDENTIFIER, 11, y>
<OPERATOR, 14, =>
<INTEGER CONSTANT, 17, 3>
<PUNCTUATION, 18, ;>
<IDENTIFIER, 11, x>
<OPERATOR, 14, =>
<INTEGER CONSTANT, 17, 5>
<OPERATOR, 12, +>
<IDENTIFIER, 11, y>
<OPERATOR, 13, *>
<INTEGER CONSTANT, 17, 4>
<PUNCTUATION, 18, ;>
<SPECIAL SYMBOL, 16, }>
```

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);
```