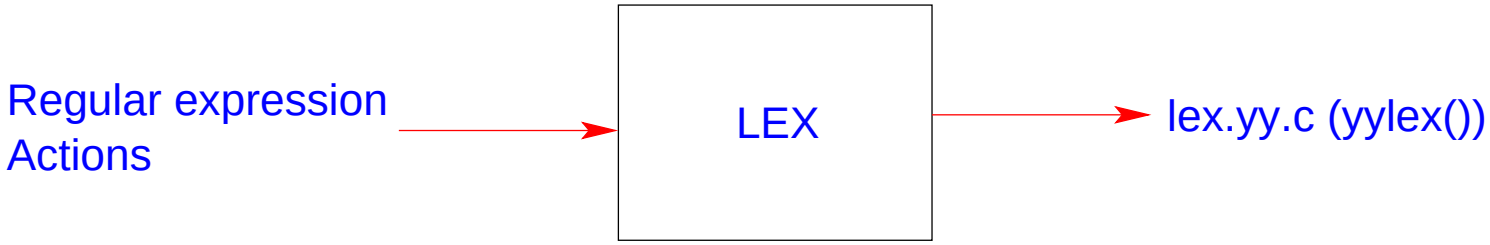
Lex and Yacc

Arijit Mondal

Lex: Lexical Analysis

Introduction

- Dividing the input into meaningful units
- Control flow is directed by instance of regular expressions in the input stream
- Matches input stream against the table of regular expressions supplied
- Carries out the associated action when a match is found



Structure of Lex file

```
%{
definitions
%}
%%

rules

%%

user subroutines
```

- Rules: `reg_exp {action}`
- `reg_exp` — Starts at the beginning of line and continue till first unescaped white space
- `action` — C statement

Example

```
%{
/* simple example */
%}
%%
[\t ]+              {}
is | are           {printf("%s:  verb\n",yytext);}
simple | easy       {printf("%s:  adjective\n",yytext);}
to | from         {printf("%s:  preposition\n",yytext);}
[a-zA-Z]+           {printf("%s:  may be noun\n",yytext);}
[0-9]+           {printf("%s:  number\n",yytext); return atoi(yytext);}
.|\n               {}
%%
main(){
    yylex();
}
```

Compilation

```
lex file_name.l

gcc lex.yy.c -o a.out -ll
```

Rules for matching

- The longest match is preferred.
- Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

Left context sensitivity

- Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input.
- Declare a set of start condition, `s - inclusive, x - exclusive`
- Example:

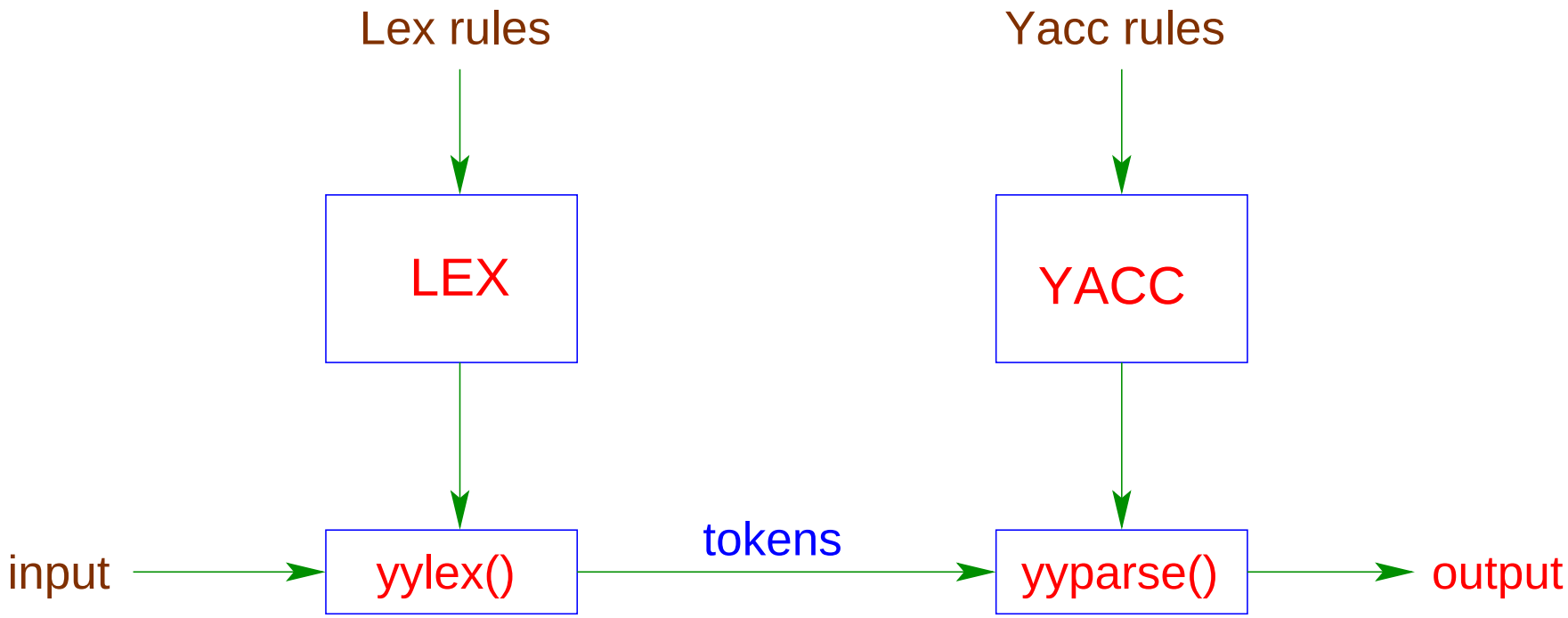
```
%{
%}
%x COMMENT
%%
"/*"                    {BEGIN COMMENT;}
"<COMMENT>[^*/]         {}
"<COMMENT>"*/"          {BEGIN 0;}
```

Communicating with user program

- `yytext` — A character array that contains the string that matched a pattern
- `yyleng` — The number of characters matched

Yacc: Yet Another Compiler Compiler

Lex & Yacc: Combined flow



Structure of YACC file

```
declarations

%%

grammar rules

%%

user subroutines
```

- Grammar rules: Production rules eg. $A \rightarrow B_1 B_2$, $B_1 \rightarrow C_1 C_2$
- Consists of terminal and non-terminal symbols
- Terminal symbols are to be defined as token in the definition section

Example

Input:

```
a = 5 + 6 ;
b = 5 + 6 + 9 - 6;
```

Output:

```
a = 11
b = 14
```

Example

```
1. %{ ...  %}
2.
3. %union { int ival; char *sval; }
4. %token <ival> INT
5. %token <sval> STRING
6. %start statements
7. %type <ival> mathexp
8. %left '+' '-'
9.
10. %%
11. statements:
12.       | statements endexpression {} ;
13. endexpression:  STRING '=' mathexp ';' {printf("%s = %d\n",$1,$3);} ;
14. mathexp:  mathexp '+' mathexp {$$=$1+$3;}
15.       | mathexp '-' mathexp {$$=$1-$3;}
16.       | INT {$$=$1;} ;
17. %%
18.
19. main(){ yyparse(); }
20. yyerror(char *s){ printf("Error:  %s\n",s); }
```

Example

```
 1. %{
 2. #include "y.tab.h"
 3. %}
 4.
 5. %%
 6. [\t ]+          {}
 7. [0-9]+        {yylval.ival=atoi(yytext); return INT;}
 8. [a-zA-Z]+    {yylval.sval=(char *)malloc(yyleng+1);
 9.                    strcpy(yylval.sval,yytext); return STRING;}
10. "="          {return yytext[0];}
11. "+"          {return yytext[0];}
12. "-"          {return yytext[0];}
13. ";"          {return yytext[0];}
14. "\n"          {}
15. %%
16.
```

Compilation

```
yacc -d -v file_name.y

lex file_name.l

gcc lex.yy.c y.tab.c -o a.out -ll
```

Conflicts

- Reduce/Reduce conflict

```
start:  a Y
     | b y ;
a :  X
b :  X
```

- Shift/Reduce conflict

```
start:  x
     | y R ;
x :  A R
y :  A
```