CSE 504 Compiler Design
Lex & Yacc

YoungMin Kwon

Lex

- A lexical scanner tool
- Lex program is comprised of 3 sections separated by %%
  - Definition Section:
    - Any initial C program code, like header files, comes here.
    - The C code needs to be surrounded by %{ and %} delimiters.
  - Rules Section:
    - Each rule is a pair of a pattern (a regular expression) and an action.
    - When a pattern is recognized, the corresponding action is executed.
    - The rules are evaluated from the first to the last and when there are multiple matches in a pattern the longest one is chosen.
  - User subroutine section:
    - Any legal C code can come here.

Regular Expressions in Lex

- . matches any single character except for \n.
- * matches zero or more copies of the preceding expression.
- + matches one or more copies of the preceding expression.
- ? matches zero or one occurrence of the preceding expression.
- {} if 1 ~ 2 numbers or a name is contained
  - how many time the previous pattern is allowed if it contains 1 ~ 2 numbers.
    - A{1,3}: one to three occurrences of A.
  - substitution of a name if it contains a name.
- \ to escape metacharacters
  - \n for the newline character, * for the literal asterisk character.

Regular Expressions in Lex

- ^ matches the beginning of a line
- $ matches the end of a line
- [] character class
  - any characters inside the brackets.
  - if the first character is ^, any characters except for the ones in the brackets.
  - - can be used to indicate the range like a-z, 0-9.
  - - or] at the first character position is interpreted literally.
- | matches either the preceding expression or the following expression
  - e.g. cow | pig | sheep

Regular Expressions in Lex

- “…” matches everything within the quotation marks literally except for the C escape sequence.
- / matches the preceding expression only if followed by the following expression.
  - e.g. 0 / 1 matches 0 in the string “01” but not in the string “02”
- () group a series of regular expressions together

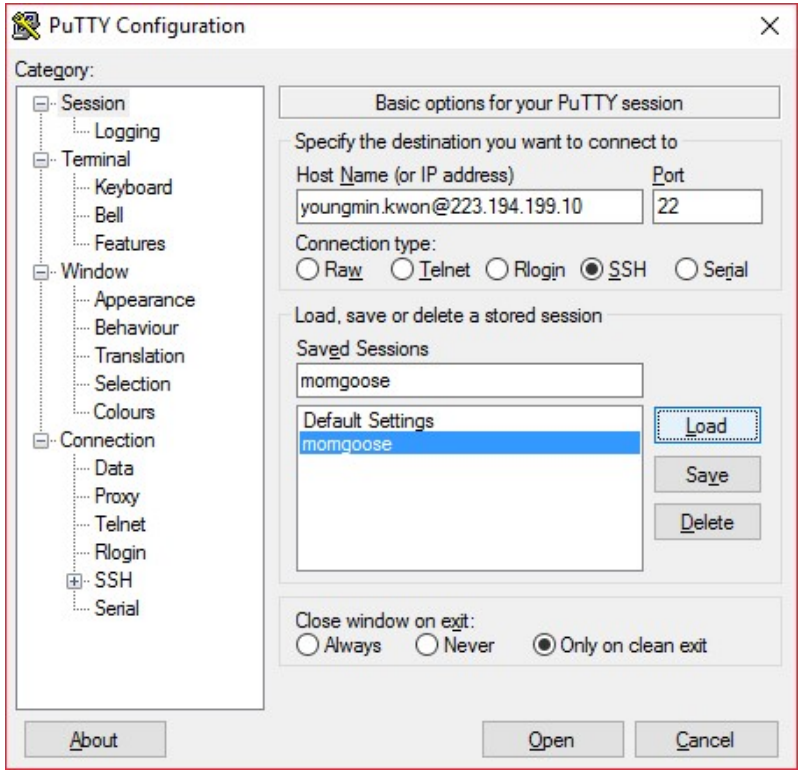
Regular Expressions (Examples)

- [0-9] matches a digit
- [0-9]+ matches a number
- [a-zA-Z_][a-zA-Z_0-9]* matches an identifier
- [\t\n\r] matches a whitespace
- #.* matches the remainder of a line from the # character (a useful expression for comments)

```
%{
    /*lex1.l: Lex example program*/
    #include <stdio.h>
%}
%%
[ \t\n\r] ;                                                    /*semicolon means no action*/
“exit”    { return 1; }                                  /*returns 1 to the caller of yylex()*/
[a-zA-Z]+ { printf(“found a word: %s\n”, yytext); } /*yytext contains the matching text*/
[0-9]+    { printf(“found a number: %s\n”, yytext); }
.         { printf(“found a special char: %s\n”, yytext); }
%%
int yywrap() { return 1; } /*ignore this function for now*/
int main(int argc, char** argv)
{
    yylex(); /*yylex tries to match the rule section*/
}
```

Compile the lex program (Windows example)

- Install an SSH program like putty (`http://www.putty.org/`)
- SSH to momgoose (223.194.199.10)
  - You can use the GUI tool on the right
  - Or, change directory to where the putty is installed
  - `putty 223.194.199.10` or `putty your_login_id@223.194.199.10`
- Create a file, say `lex1.l`, with the program in the previous page
- Compile lex1.lprogram
  - `/usr/bin/flex lex1.l` (it will create `lex.yy.c`)
  - `/usr/bin/gcc lex.yy.c` (it will create `a.out`)
- Run the compiled program
  - `./a.out`



Yacc

- A parser generator tool
- Like Lex, Yacc program is comprised of 3 sections separated by %%
- Definition Section:
  - Any initial C code, like header files, comes here. It needs to be surrounded by `%{` and `%}` delimiters.
  - Tokens (terminals) are defined here after %token keyword (single character tokens don't need definitions)
    - e.g. `%token NUMBER IDENTIFIER`
  - Token associativity (`%left, %right, %nonassoc`) and precedence (by their order of definitions from low to high) are defined here
    - Example precedence and associativity: UMINUS has the highest priority and '+', '-' have the lowest priority

```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS    (unary minus)
```

Yacc

- Definition section (continued)
  - Symbols (terminals and nonterminals) can have types defined in `%union` keyword.

```
%union {
    double dbl;
    char* str;
}
```

  - With the types we can define

```
%token<dbl> NUMBER
%token<str> IDENTIFIER
%type<dbl> expr term factor
```

  - It is customary to use all upper case names for terminals and all or mostly lower case names for others.
- User subroutine section (after the second %%)
  - Any legal C code can come here.

Yacc (Rule Section)

- A program area in between the first %% and the second %%
- A list of productions (rules) are defined in the rule section
  - Each rule defines a production
    - Arrow (->) is replaced by `':'`
    - The end of a rule is marked by `';'`
  - The Left Hand Side (LHS) of the first rule is the start symbol (the root of the parse tree) unless overridden by `%start` declaration.
- Actions, C codes wrapped in { and }, can be added to the rules.
  - As soon as a rule matches, the corresponding action is executed.
  - The values of Right Hand Side (RHS) symbols are `$1, $2, …`
  - The value of the LHS symbol is `$$`
- Example

```
expr : expr '+' term { $$ = $1 + $3; }
     | expr '-' term { $$ = $1 - $3; }
     ;
```

Working with Lex

- yyparse() is the function that starts the parsing.
- yyparse calls yylex() when it needs a token.
- The tokens defined in a yacc program will be added to y.tab.h file and a lex program can include this header file to use the symbols.

```
%union {
    double dbl;
    char* str;
}
%token<dbl> NUMBER
%token<str> IDENTIFIER
```

will be converted to

```
#define NUMBER 257
#define IDENTIFIER 258
typedef union {
    double dbl;
    char* str;
} YYSTYPE;
extern YYSTYPE yylval;
```

```
%{
    /*file name: calc.y*/
    #include <stdio.h>
    #include <stdlib.h>

    int yylex();
    int yyerror(char*);
%}

%union {
    double dval;
};

%token <dval> NUMBER
%type <dval> expr term factor

%%

Rule section is on the right

%%

int main(int argc, char**argv)
{
    yyparse();
}
```

```
calc : expr ‘\n’ { printf("ans = %lf\n", $1); }
     ;
expr : expr '+' term   { $$ = $1 + $3; }
     | expr '-' term   { $$ = $1 - $3; }
     | term            { $$ = $1; }
     ;
term : term '*' factor { $$ = $1 * $3; }
     | term '/' factor { $$ = $1 / $3; }
     | factor          { $$ = $1; }
     ;
factor : '(' expr ')'  { $$ = $2; }
       | NUMBER        { $$ = $1; }
       ;
```

```
%{
    /*file name: calc.l*/
    #include <string.h>
    #include “calc.tab.h”
%}
%%
[ \t\r]+  ;                          /*ignore white spaces*/
([0-9]+(\.[0-9]+)?)|(\.[0-9]+) {
    yylval.dval = atof(yytext);      /*set the value of the token*/
    return NUMBER;                   /*return the token NUMBER*/
}
.|\n {
    return yytext[0];                /*return the single character tokens*/
}
%%
int yywrap() { return 1; }           /*ignore this function for now*/
int yyerror(char*) { return 1; }     /*ignore this function for now*/
```

Compile with Make

- SSH to momgoogse and create calc.l and calc.y in the previous slides
- Create a file with the following contents and name it as Makefile

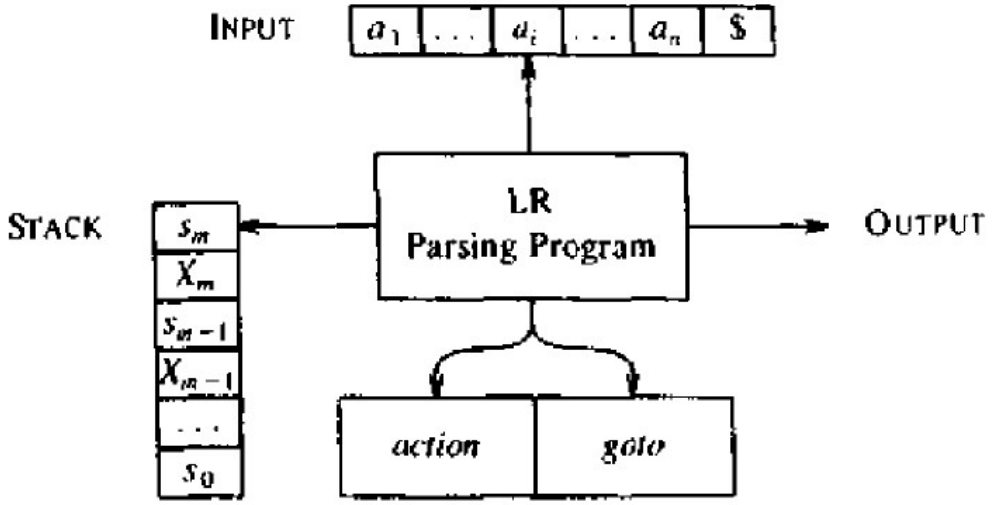
```
LEX = /usr/bin/flex                 #define the variables LEX, YACC, and CC
YACC = /usr/bin/bison
CC = /usr/bin/gcc

a.out: lex.yy.c calc.tab.c          #LHS of ':' is the output, RHS is the input
    $(CC) lex.yy.c calc.tab.c       #the blank at the beginning should be a tab
lex.yy.c: calc.l calc.tab.h
    $(LEX) calc.l
calc.y.c calc.y.h: calc.y
    $(YACC) calc.y -d               #-d option creates calc.y.h
```

- Run the command make
- Try run the program a.out

How the parser works

- The parser created by yacc is LALR(1) parser, an LR parser with 1 look ahead. We will learn LR parsers later.
- LR parsers use a stack, an action table, and a goto table to parse the input.
- The parsing algorithm can be described by actions on configuration
- Configuration
  - A stack of states and symbols (terminals and nonterminals), a delimiter |, and unhandled input tokens
  - (S_1, X_1, S_2, X_2 … S_n | T_1, T_2, …), where S_i is a state, X_i is a symbol, T_i is a token.

How LR parsers work



- 4 Actions of LR parser
  - Shift and go to state S
    - (… S_1 | T_1 T_2…) -> (… S_1 T_1 S | T_2 …)
  - Reduce X -> X_1 … X_n
    - (… S_0 X_1 S_1 … X_n S_n | T_1 …) -> (… S_0 X S | T_1 …), where S is the goto target of S_0 for symbol X.
  - Accept: finish with success
  - Error: found an error

How the parser works

- To see how the parser works, let's create a yacc program (phrase.y)

```
phrase: cart_animal CART
      | work_animal PLOW
      ;
cart_animal: HORSE
           | GOAT
           ;
work_animal: HORSE
           | OX
           ;
```

- Run /usr/bin/bison phrase.y –v will produce phrase.output and other files
- Next slide shows some of the states, actions (shift, reduce, accept), and goto tables of each state in phrase.y.
- The dots '.' in the first lines of each state are where in the productions the state represents.
- The last part shows how the configuration changes for the input HORSE CART.

phrase.output file (add -v option to bison)

```
0 $accept: phrase $end (yacc added this rule)
1 phrase: cart_animal CART
2       | work_animal PLOW
3 cart_animal: HORSE
4            | GOAT
5 work_animal: HORSE
6            | OX
```

States

```
state 0
    0 $accept: . phrase $end
    HORSE  shift, and go to state 1
    GOAT   shift, and go to state 2
    OX     shift, and go to state 3
    phrase       go to state 4
    cart_animal  go to state 5
    work_animal  go to state 6
state 1
    3 cart_animal: HORSE .
    5 work_animal: HORSE .
    PLOW      reduce using rule 5 (work_animal)
    $default  reduce using rule 3 (cart_animal)
state 2
    4 cart_animal: GOAT .
    $default  reduce using rule 4 (cart_...)
state 4
    0 $accept: phrase . $end
    $end  shift, and go to state 7
state 5
    1 phrase: cart_animal . CART
    CART  shift, and go to state 8
state 7
    0 $accept: phrase $end .
    $default  accept
state 8
    1 phrase: cart_animal CART .
    $default  reduce using rule 1 (phrase)
```

Configurations for the input "HORSE CART"

```
(0 | HORSE CART $end)
(0 HORSE 1 | CART $end)
(0 cart_animal 5 | CART $end)
(0 cart_animal 5 CART 8 | $end)
(0 phrase 4 | $end)
(0 phrase 4 $end 7 | )
accept
```

Shift/Reduce Conflict

- Shift/Reduce conflict occurs when both shift and reduce actions are possible for an input string.
- Example

```
e : 'X'
  | e '+' e
  ;
```

  - X + X ↑ + X have two possible actions at the position ↑
    - After reducing the string to e + e ↑ + X
    - Shift + and reduce X to e later
    - Reduce e + e to e

Reduce/Reduce Conflict

- Reduce/Reduce conflict occurs when two reduce actions are possible for an input string.
- Example

```
e : e1 | e2 ;
e1 : 'X' ;
e2 : 'X' ;
```

  - An input X can be reduced to both e1 and e2.