University of Moratuwa, Sri Lanka

Faculty of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING B. Sc Engineering Honors Degree

CS3062 Theory of Computing (2 credits)
Semester 5, 15 Intake (Jan – May 2018)

Assignment 2 Part 1 (Worth-> 8%, Due-> 28/05/2018 at 11->55PM)

Group Assignment
Scanner & Parser Generation

2 students per group. You are free to select your lab partner.

The assignment is to generate a lexical analyzer and syntax analyzer for the language specified in this document.

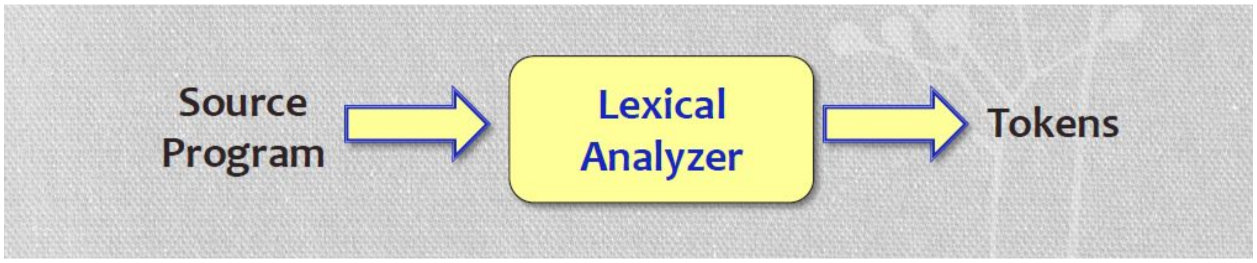
You can use a lexical analyzer generator and parser generator, or you may code the lexical analyzer on your own using a suitable programming language.

Submission - Your submission should include a single zip file which will contain all the software and the files required to demonstrate your project.

Evaluation - Each group has to demonstrate his/her project in an assigned slot. You will be assessed based on the completeness and quality of your work and your ability to answer the questions during the demonstration. In this demonstration, you are expected to explain how your scanner and parser work and for different inputs, how your parser will react. You may implement error handling or additional features for extra credit.

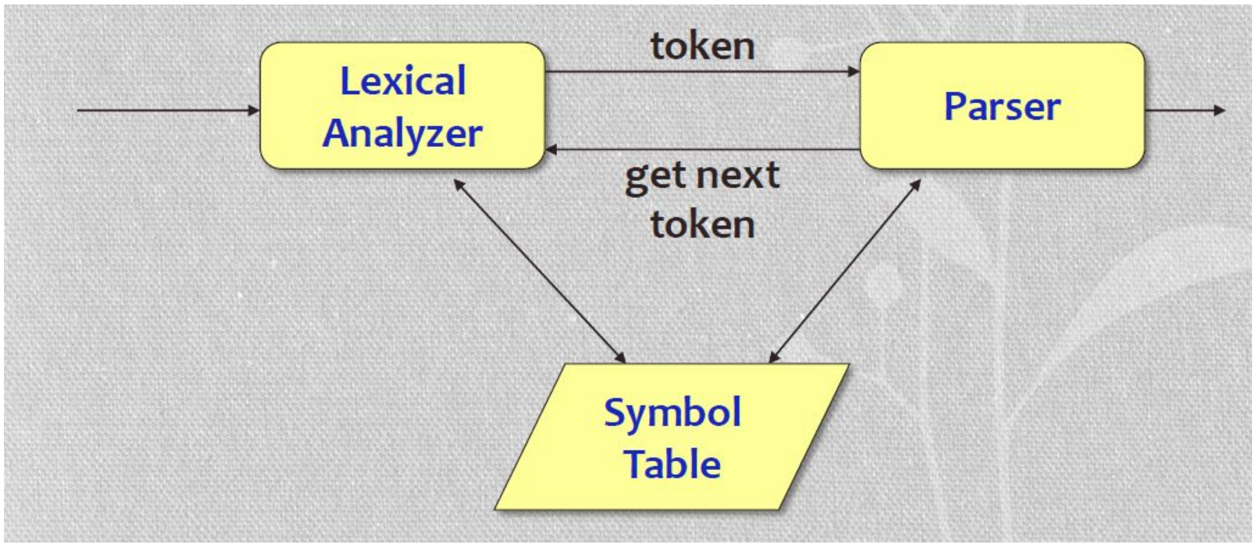
Reading Assignment: You are expected to go through how lexical analyzer and parser generator work and relate to regular expressions and context free grammars. Following is a brief introduction to lexical analyzer and parser generator.

The scanning process
Lexical analyzer reads input characters and produces a sequence of tokens.



In PLs, following are usually treated as tokens.
Keywords, Operators, Identifiers, Constants, Literal strings, Punctuation symbols

Regular expressions are used to produce tokens from a sequence of characters (program).

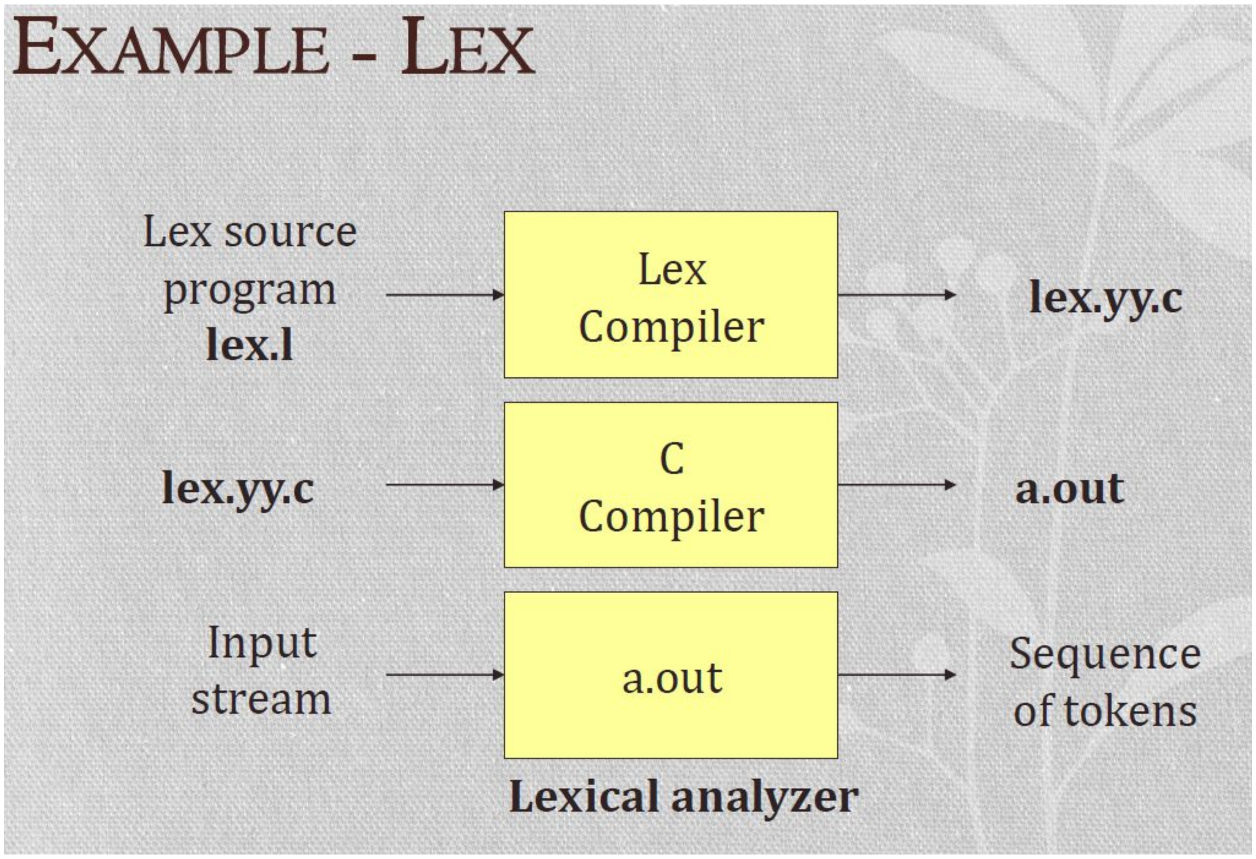


Software tools to generate lexical analyzer

- lex, flex, JLex, JavaCC
- Uses notations based on RE for specifying the tokens
- RE to FA conversion

Example - Lex

- Tool is also called "Lex compiler"
- Works with the "yacc" program (parser generator) in C environment
- Input is given in "Lex language"
  - Specify patterns using regular expressions
  - Specify actions for the lexical analyzer
    - E.g., make entries to symbol table, return token to parser



A lex program consists of 3 parts

- Declarations
    - Variable declaration, manifest constants, regular definitions
    - Ex: Letter[A-Za-z]
- Translation rules
    - P {action}
    - Ex:
        - {if}   {return (IF);}
        - {id}   {yylval= install_id(); return(ID);}
- Auxiliary procedures
    - Procedures needed by actions
        - These procedures may be compiled separately and loaded with the lexical analyzer generated using Lex.

Scanner is used in conjunction with parser. When activated by the parser,

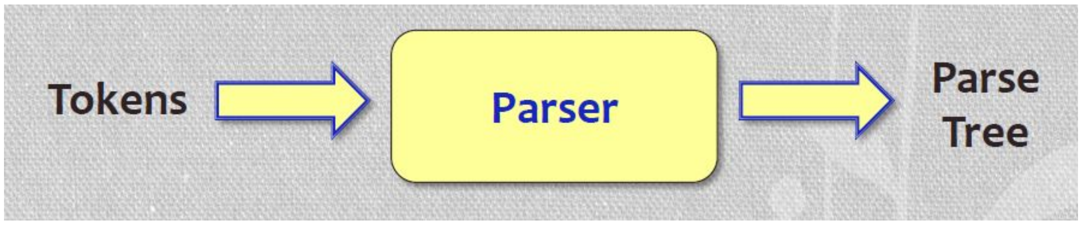
- Reads remaining inputs one character at a time
- When a token is found, the corresponding action will be executed
- This action will return control to parser
- Only one token passed to the parser in a single call

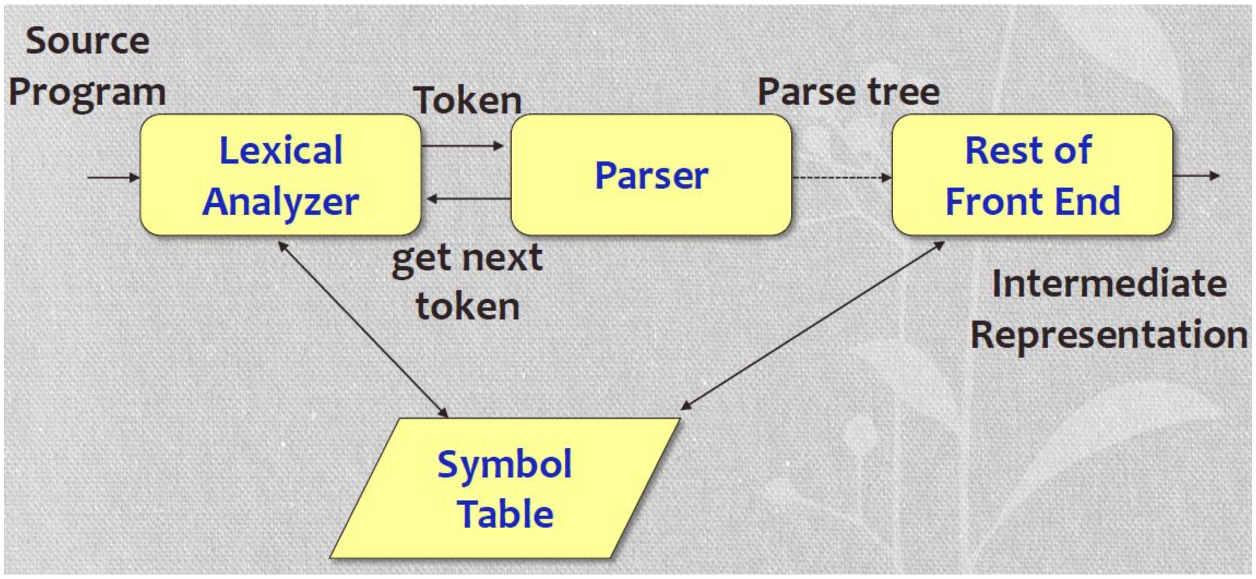
Resolving Conflicts
Two rules used

- Longest prefix match
- Rule priority
    - For a particular longest prefix, the 1st regular expression that can match determines its token type
    - Order of translation rules is important
    - Easy to reserve keywords by listing them first

The Role of the Parser

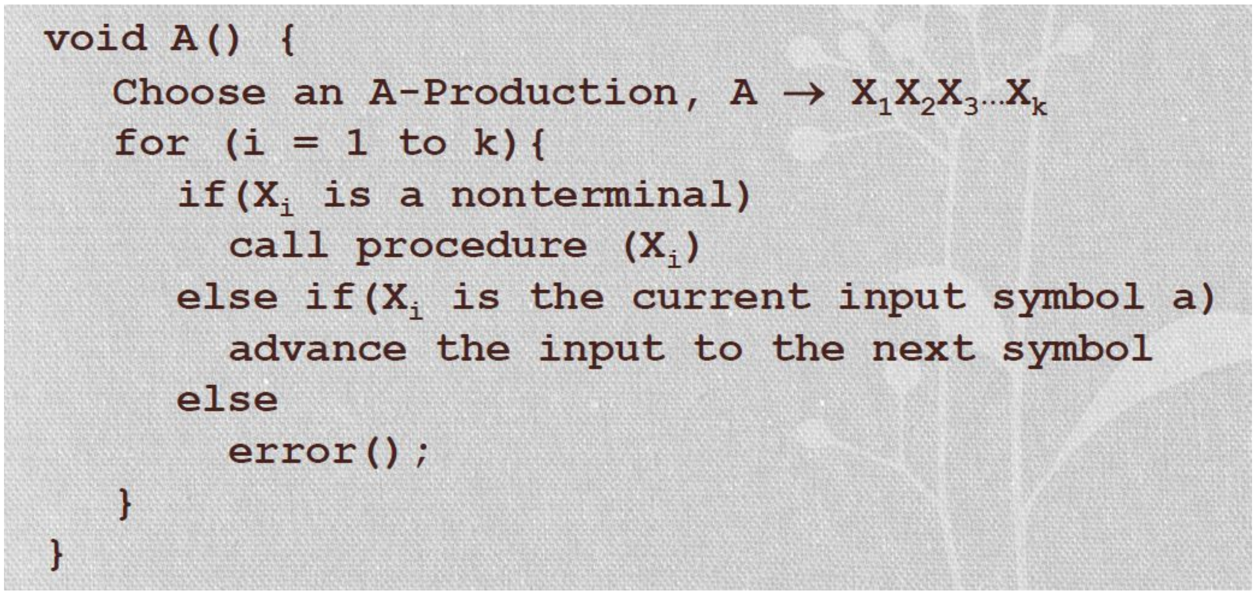
- Obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the language
- Produces the parse tree





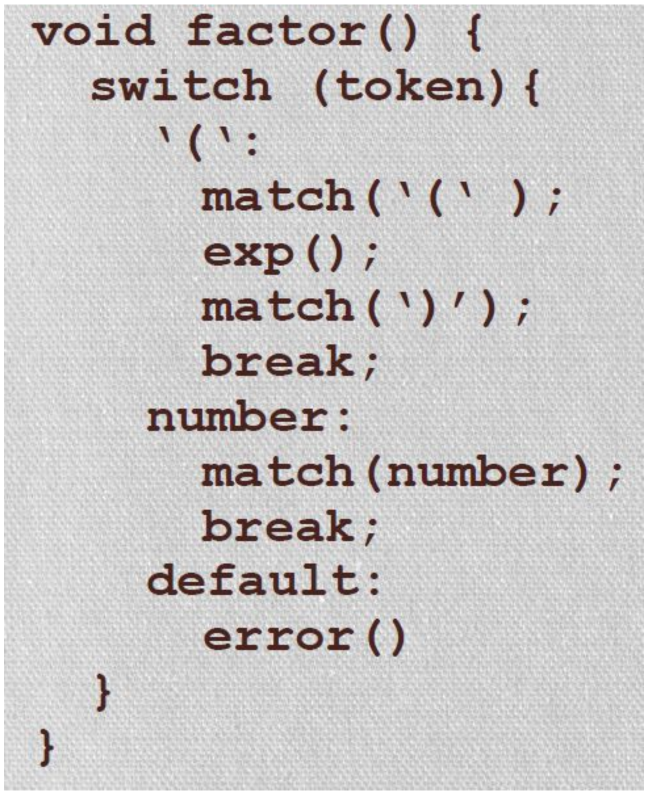
Recursive Descent Parsing

- Executes a set of recursive procedures to process input.
- A typical procedure for a non terminal in a top-down parser



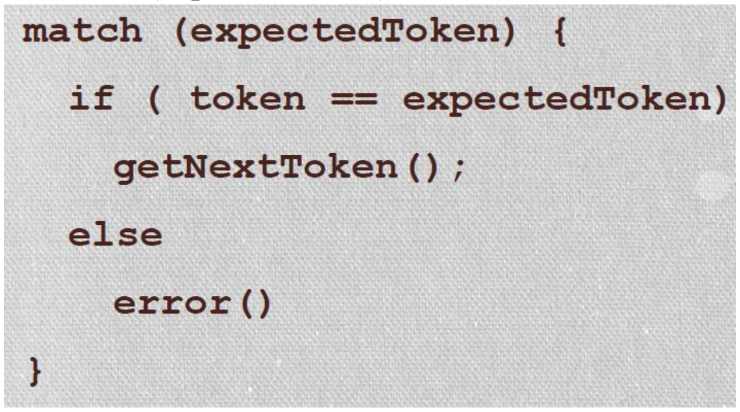
```
void A() {
   Choose an A-Production, $A \rightarrow X_1X_2X_3...X_k$
   for (i = 1 to k){
      if($X_i$ is a nonterminal)
        call procedure ($X_i$)
      else if($X_i$ is the current input symbol a)
        advance the input to the next symbol
      else
        error();
   }
}
```

- Example: Consider the rule; <factor > -> (<exp>) | <number>



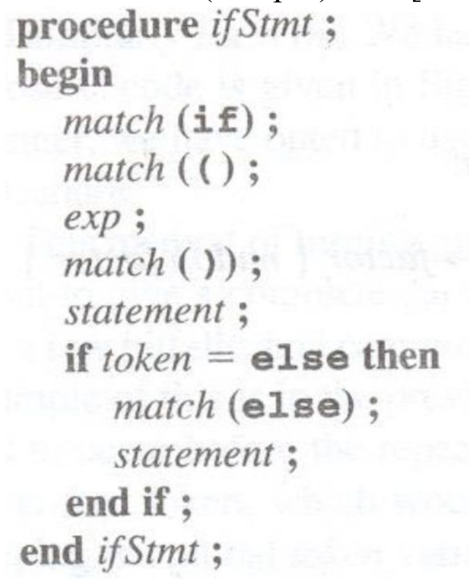
```
void factor() {
  switch (token){
    '(':
      match('(' );
      exp();
      match(')');
      break;
    number:
      match(number);
      break;
    default:
      error()
  }
}
```

- The match(expected Token) function



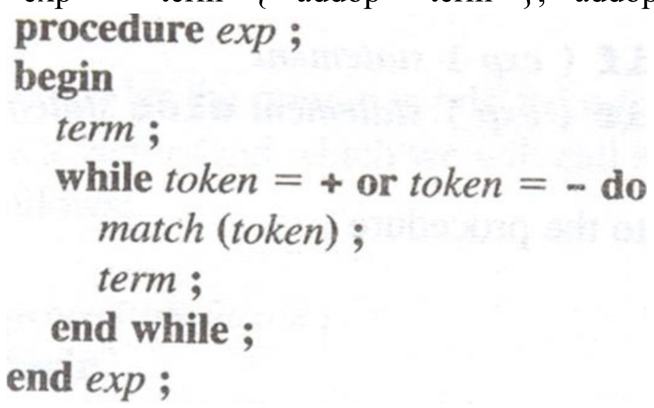
```
match (expectedToken) {
  if ( token == expectedToken)
    getNextToken();
  else
    error()
}
```

- **Examples**
  - <Ifstmt> -> if (<exp>) <st> [else <st>]



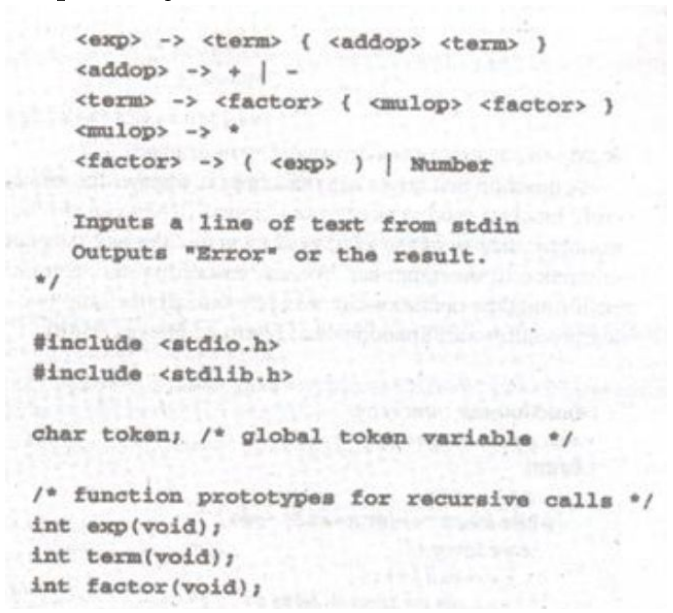
```
procedure ifStmt ;
begin
  match (if) ;
  match (() ;
  exp ;
  match ()) ;
  statement ;
  if token = else then
    match (else) ;
    statement ;
  end if ;
end ifStmt ;
```

  - <exp> -> <term> { <addop> <term> }, <addop> -> + | -



```
procedure exp ;
begin
  term ;
  while token = + or token = - do
    match (token) ;
    term ;
  end while ;
end exp ;
```

○ Simple integer arithmetic calculator



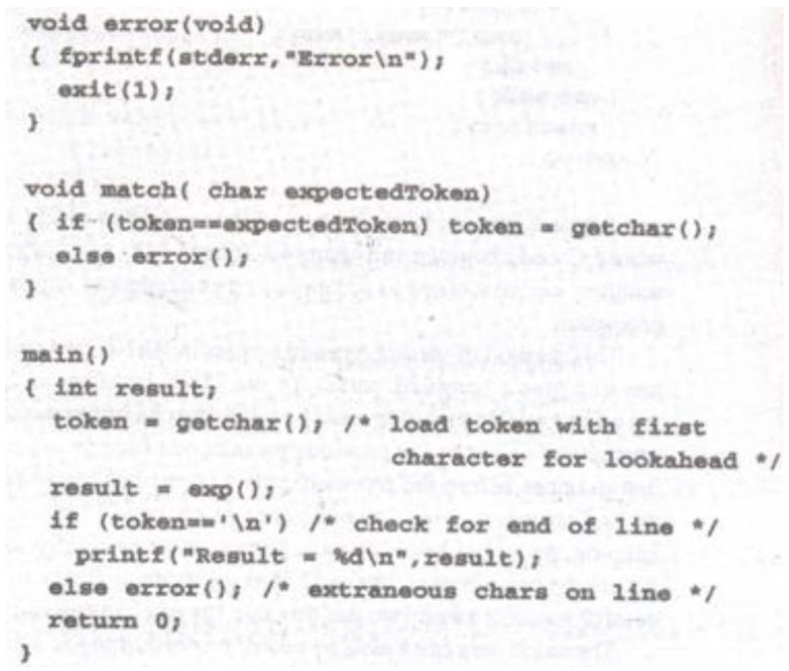
```
    <exp> -> <term> { <addop> <term> }
    <addop> -> + | -
    <term> -> <factor> { <mulop> <factor> }
    <mulop> -> *
    <factor> -> ( <exp> ) | Number

    Inputs a line of text from stdin
    Outputs "Error" or the result.
 */

 #include <stdio.h>
 #include <stdlib.h>

 char token; /* global token variable */

 /* function prototypes for recursive calls */
 int exp(void);
 int term(void);
 int factor(void);
```



```
void error(void)
{ fprintf(stderr,"Error\n");
  exit(1);
}

void match( char expectedToken)
{ if (token==expectedToken) token = getchar();
  else error();
}

main()
{ int result;
  token = getchar(); /* load token with first
                        character for lookahead */
  result = exp();
  if (token=='\n') /* check for end of line */
    printf("Result = %d\n",result);
  else error(); /* extraneous chars on line */
  return 0;
}
```

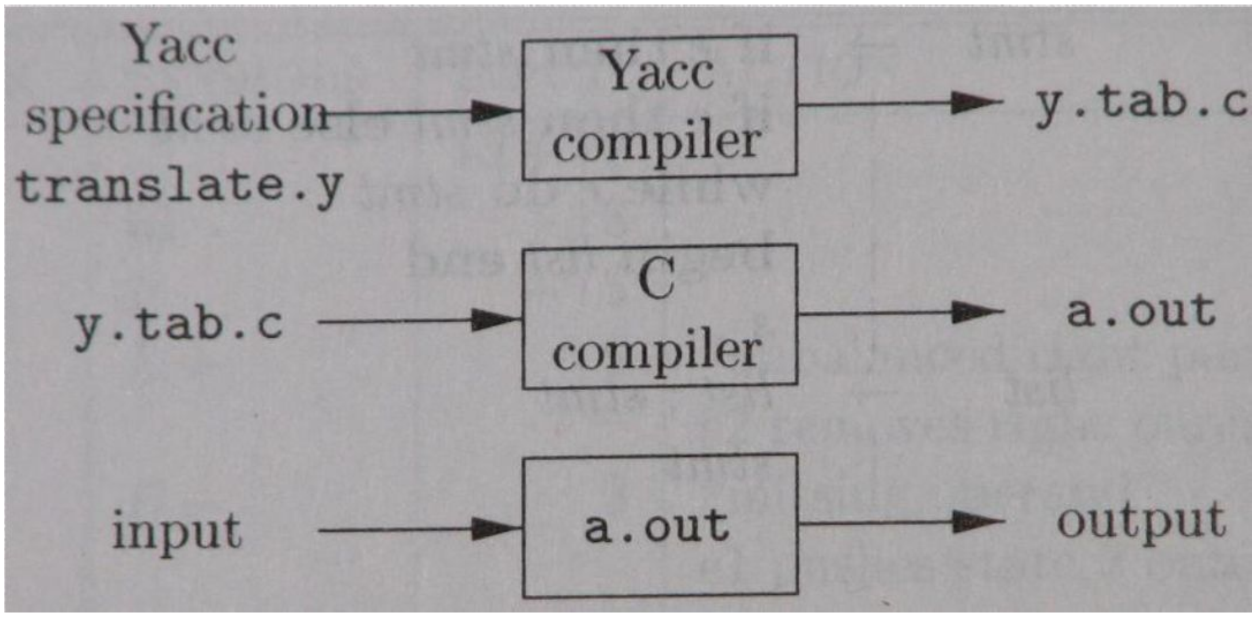
```
int exp(void)
{ int temp = term();
  while ((token=='+')||(token=='-'))
    switch (token) {
    case '+': match('+');
              temp+=term();
              break;
    case '-': match('-');
              temp-=term();
              break;
    }
  return temp;
}

int term(void)
{ int temp = factor();
  while (token=='*') {
    match('*');
    temp*=factor();
  }
  return temp;
}
```

```
int factor(void)
{ int temp;
  if (token=='(') {
    match('(');
    temp = exp();
    match(')');
  }
  else if (isdigit(token)) {
    ungetc(token,stdin);
    scanf("%d",&temp);
    token = getchar();
  }
  else error();
  return temp;
}
```

NOTE: You can use LALR parsing if you use YACC. If you are planning to code the parser generator yourself, you have to use recursive descent parsing, as mentioned above.

YACC



Project Description

Programming language C-Minus which is a suitable language for a compiler project is defined as follows. It is essentially a subset of C language but is missing some important pieces, hence its name.

This appendix consists of three sections. ln the first. the lexical conventions of the language, including the description of the tokens of the language are listed. In the second BNF description of each language construct, together with an English description of the associated semantics are given. In the third section, two sample C- programs are given.

Lexical Conventions of C-

1. The keywords of the language are the following.
   else, if, int, return, void, while
   All keywords are reserved and must be written in lowercase.

2. Special symbols are as following.
   + - * / < <= > >= == != = ; , { } ( ) [ ] /* */

3. Other tokens are ID and NUM, defined by the following regular expressions.
   ID = letter letter*
   NUM = digit digit*
   letter = a|..|z|A|..|Z
   digit = 0|..|9

Lowercase and uppercase letters are distinct.

4. White space consists of blanks, newlines and tabs. White space is ignored except that it must separate IDs and NUMs and keywords.

5. Comments are surrounded by the usual C notations /* */. Comments can be placed anywhere white space can appear and comments cannot be placed within tokens. Comments may include more than one line, but comments may not be nested.

Syntax and Semantics of C-

1. program -> declaration-list

2. declaration-list -> declaration-list declaration | declaration

3. declaration -> var-declaration | fun-declaration

4. var-declaration-> type-specifier ID ; | type-specifier ID [NUM] ;

5. type-specifier-> int | void

6. fun-declaration-> type-specifier ID (params) compound-stmt

7. params-> param-list | VOID

8. param-list-> param-list , param| param

9. param-> type-specifier ID | type-specifier ID [ ]

10. compound-stmt-> { local-declarations statement-list }

11. local-declarations-> local-declarations var-declaration | empty

12. statement-list-> statement-list statement | empty

13. statement-> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

14. expression-stmt-> expression ; | ;

15. selection-stmt-> if(expression) statement | if(expression) statement else statement

16. iteration-stmt-> while (expression) statement

17. return-stmt-> return ; | return expression ;

18. expression-> var = expression | simple-expression

19. var-> ID | ID [expression]

20. simple-expression -> additive-expression relop additive-expression| additive-expression

21. relop-> < | <= | > | >= | == | !=

22. additive-expression-> additive-expression addop term | term

23. addop-> + | -

24. term-> term mulop factor | factor

25. mulop-> * | /

26. factor-> (expression) | var | call | NUM

27. call-> ID (args)

28. args-> arg-list | empty

29. arg-list-> arg-list , expression | expression

Example Program 1

```
/* A program to perform Euclid's
   Algorithm to compute gcd. */

int gcd (int u, int v){
        if(v==0) return u ;
        else return gcd(v, u-u/v*v);
        /* u-u/v*v == u mod v*/
}

void main(void) {
        int x;  int y;
        x = input(); y = input();
        output(gcd(x, y));
}
```

Example Program 2

```
/* A program to perform selection sort on a 10   element array. */

int x[10];

int minloc ( int a[], int low, int high ){
 int i; int x; int k;
 k = low;
 x = a[low];
 i = low + 1;
 while(i< high)
          {if (a[i] < x)
                    { x = a[i];
                     k = i; }
          i = i + 1;
          }
 return k;
}
void sort( int a[], int low, int high)
 { int i; int k;
   i = low;
   while (i < high - 1)
    {
     int t;
     k = minloc(a, i, high);
     t = a[k];
     a[k] = a[i];
     a[i] = t;
     i = i + 1;
    }
 }
void main (void)
 { int i;
   i = 0;
   while (i < 10){
     x[i] = input();
     i = i + 1; }
   sort(x, 0, 10);
   i = 0;
   while (i < 10)
    { output (x[i]);
      i = i + 1; }
}
```