Indian Institute of Technology Guwahati
End Semester Examination
Compilers (CS 346)
(January-April Semester 2020)

Name : Rashi Singh
Roll No:170101052
Full Marks: 80
Time: 25 Hours

Name and roll no. should be written clearly in the first page of the answer answer booklet. Roll no. should be written in the each page of the question answer booklet.

Q. No.	Division of Marks	Marks Obtained
1		
2		
3		
4		
5		
6		
7		

Attempt all questions.

No explanation about any questions will be given during examination; all questions are seemed to be self-explanatory. In case of any discrepancy, write your assumption clearly before answering the question.

Consider the following code segment of a hypothetical language:

```
integer :: n,num,sum,i;
scanf("%d",&n);
do i=1,n
   ! Print Sum upto each digit    (Comment line)
   num:=1, sum:=0;
   while num<=i do
   begin
         sum:=sum+num;
         num:=num+1;
   end
   printf("%d\n",sum)
end do
```

Code Segment for Q.1

1. Answer the following questions based on above code segment.

   (a) Write a pseudo code for tokenization for keywords, identifier, operators, comments etc. [10]

   (b) Write a Regular Expression (RE) for validating the tokens for the above code. [10]

   (c) Write a pseudo code for validating these tokens using your RE. [15]

2. Write a pseudocode for the following grammar using Recursive Descent Parsing.

   E -> TE'
   E'->+TE/@ where @ is null character
   T->FT'
   T'->*FT'/@
   F->(E)/ID

   (a) Write a pseudo code to read the input string. [5]

   (b) Write a pseudo code for RD parser by matching terminals and non-terminals and backtracking. [10+5=15]

   Hints: Enter the string to be checked: (a+b)*c
   String is accepted

   Enter the string to be checked: a/c+d
   String is not accepted

3. Design an attributed grammar for type checking using both inherited and synthesized attributes for the following PASCAL declaration for identifiers:

D → L : T

T → integer | char

L → L , id | id

(a) Write a C like program/ pseudocode to implement Program semantic rules. [10]

(b) Restructure the grammar so that the type checking can be performed using only synthesized attribute. Provide the corresponding attributed grammar. [15]

ANSWERS

Ans 1 -

Part a)

```
bool isKeyword ( int token ) {
        return ( token == BEGIN || token == END || token == WHILE ||
                token == DO || token == PRINTF || token == SCANF ||
                token == INT );
}

bool isId ( int token ) {
        return token == NUM_OR_ID;
}

bool isOperator ( int token) {
        return ( token == AMP || token == ASSIGN || token == LE ||
                token == GE || token == LESS || token == MORE ||
                token == EQUAL || token == PLUS || token == MINUS ||
                token == MUL || token == DIV );
}

bool isSymbol ( int token ) {
        return ( token == PERC || token == OPEN || token == CLOSE ||
                token == SEMI || token == BSLASH || token == SCOPE ||
                token == COMMA || token == QUOTE );
}

bool isComment ( int token) {
        return token == COMMENT;
}

bool isConst ( int token ) {
        char temp[1000] = "";
        strcpy(temp,next,tokenLength);
        for ( int i = 0 ; i < tokenLength ; i++){
                char c = temp[i];
                if ( !( c <= '9' && c >= '0' )) return false;
        }
        return true;
```

```
}

char* tokenClass ( int token ){
        if( isKeyword ( token )) return “kw”;
        else if ( isOperator ( token )) return “op”;
        else if ( isId ( token )) {
                if ( isConst ( token )) return “const”;
                return “id”
        }
        else if ( isComment ( token )) return “comment”;
        else if ( isSymbol ( token )) return “sm”;
        else return “err”;
}
```

Where file lex.h can contain the following macros -

```
#define BEGIN 1     #define END 13      #define WHILE 25
#define DO 2        #define PRINTF 14   #define SCANF 26
#define INT 3       #define SEMI 15     #define PLUS 27
#define MUL 4       #define MINUS 16    #define DIV 28
#define OPEN 5      #define CLOSE 17    #define MORE 29
#define LESS 6      #define ASSIGN 18   #define AMP 30
#define EOI 7       #define MINUS 19    #define DIV 31
#define OPEN 8      #define CLOSE 20    #define MORE 32
#define LESS 9      #define ASSIGN 21   #define NUM_OR_ID 33
#define LE 10       #define GE 22       #define EQUAL 34
#define PERC 11     #define BSLASH 23   #define COMMA 35
#define SCOPE 12    #define ERR 24      #define COMMENT 36
```

Please note the token generation of string source is mentioned in part c of Q1

Part b) RE for the given language -

Digit	-	[0-9]
Alphabet	-	[A-Za-z]
Alphanum	-	{**Digit** | **Alphabet**}*
Id	-	**Alphabet Alphanum**
Keywords	-	begin | end | while | do | printf | scanf | integer
Operators	-	& | := | <= | >= | < | > | = | + | - | / | *
SYMBOLS	-	% | (| ; |) | \ | , | :: | “
White_Space	-	\t | | \n

Comment - ! {**Keywords** | **Operators** | \t | | **SYMBOLS** | **Alphanum** | ! }*

Therefore, the complete set of tokens is -

Tokens - **Comment** | **White_Space** | **Operators** | **Keywords** | **Id** | **Alphanum** | **SYMBOLS**

Part c) Validating tokens using RE

```
// Some global variables
TOKEN* next = null;
int nextIndex = 0;
int tokenLength = 0;

int lex (){
    while( nextIndex < source.size() ){
        tokenLength = 1;
        switch(next = source[nextIndex]){
            case ‘+’ : return PLUS;
            case ‘-’  : return MINUS;
            case ‘/’  : return DIV;
            case ‘%’ : return PERC;
            case ‘&’ : return AMP;
            case ‘=’ : return EQUAL;
            case ‘*’  : return MUL;
            case ‘(‘  : return OPEN;
            case ‘)’  : return CLOSE;
            case ‘;’  : return SEMI;
            case ‘”’  : return QUOTE;
            case ‘!’  : tokenLength = 1;
                        next = &(source[nextIndex]);
                        while( nextIndex < source.size() &&
                            Source[nextIndex] != ‘\n’ ){
                            tokenLength++;
                            nextIndex++;
                        }
                        return COMMENT;
```

```
case ':'  : tokenLength = 2;
            if ( nextIndex < source.size() - 1 ){
                if(source[nextIndex + 1] == ':')
                        return SCOPE;
                else if ( source[nextIndex + 1] == '=')
                        return ASSIGN;
                else return ERR;
            }
            else return ERR;
case '<' :
            if ( nextIndex < source.size() - 1 ){
                if ( source[nextIndex + 1] == '=')
                        tokenLength = 2;
                        return LE;
            }
            return LESS;
case '>' :
            if ( nextIndex < source.size() - 1 ){
                if ( source[nextIndex + 1] == '=')
                        tokenLength = 2;
                        return GE;
            }
            return MORE;
case '\n' :
case '\t' :
case ' ' :
        tokenlength = 0;
        break;
default :
        tokenLength = 0;
        next = &(source[nextIndex]);
        while( nextIndex < source.size() &&
                isAlphanum(source[nextIndex]) ){
                tokenLength++;
                nextIndex++;
        }
        char temp[1000] = "";
        strcpy(temp,next,tokenLength);
        if ( !strcmp ( temp , "begin"  )) return BEGIN;
```

```
                                        else if ( !strcmp ( temp , “end” )) return END;
                                        else if ( !strcmp ( temp , “while” )) return WHILE;
                                        else if ( !strcmp ( temp , “scanf” )) return SCANF;
                                        else if ( !strcmp ( temp, “printf” )) return PRINTF;
                                        else if ( !strcmp ( temp , “do”)) return DO;
                                        else if ( !strcmp ( temp , “integer” )) return INT;
                                        else return NUM_OR_ID;
                                }
                        }
                }
```

Ans 2 -

Given grammar is -

E -> T E’

E’ -> + T E | @

T -> F T’

T’ -> * F T’ | @

F -> (E) | id

where @ is null character

Part a)

Pseudo code to read the input string

```
bool term(TOKEN expected){
        return getNextToken() == expected;
}

TOKEN getNextToken() {
        while( nextIndex < source.size() ){
                tokenLength = 1;
                switch(next = source[nextIndex]){
                        case ‘+’ : return PLUS;
                        case ‘*’  : return MULTIPLY;
                        case ‘(‘  : return OPEN;
                        case ‘)’  : return CLOSE;
                        case ‘\n’ :
                        case ‘\t’ :
                        case ‘ ’ :
                                tokenlength = 0;
                                break;
```

```
                default :
                        tokenLength = 0;
                        next = &(source[nextIndex]);
                        while( nextIndex < source.size() &&
                                isAlphanum(source[nextIndex]) ){
                                tokenLength++;
                                nextIndex++;
                        }
                        char temp[1000] = "";
                        strcpy(temp,next,tokenLength);
                        return (TOKEN)temp;
            }
        }
}
```

Part b)

Pseudo code to match terminals and non-terminals and use backtracking

```
TOKEN* next = null;
int nextIndex = 0;
int tokenLength = 0;

bool E1()  { return T() && E'(); }
bool E () { return E1(); }

bool E'1() { return term(PLUS) && T() && E(); }
bool E'2() { return true; }
bool E' () {
        TOKEN* save = next;
        return (next = save, E'1()) || (next = save, E'2());
}

bool T1()  { return F() && T'(); }
bool T () { return T1(); }

bool T'1() { return term(MULTIPLY) && F() && T'(); }
bool T'2() { return true; }
bool T' () {
        TOKEN* save = next;
        return (next = save, T'1()) || (next = save, T'2());
```

```
}

bool F1()  { return term(OPEN) && E() && term(CLOSE); }
bool F2()  { return term(ID); }
bool F () {
        TOKEN* save = next;
        return (next = save, F1()) || (next = save, F2());
}
```

Ans 3 -

Part a)

The semantic rules for the given grammar are -

D -> L : T	{ L.type = T.type }
T -> *integer*	{ T.type = INTEGER }
| *char*	{ T.type = CHAR }
L -> L_2 , **id**	{
	L_2.type = L.type
	Insert (**id**.name , L.type)
	}
| **id**	{ Insert (**id**.name , L.type) }

Where

Insert function stores the data **type** associated with the given **id**

Part b)

Please note that

D -> L : T { L.type = T.type } ...(1)

has L.type as an inherited attribute, however, it depends on the value of its right sibling T, hence this is not an L attributed grammar

To solve this problem, the following steps can be taken -

- We need to delay the use of non-terminal **T** in production rule (1), thus giving us

  D -> L

- To re-introduce the non-terminal **T** in the grammar, we need to find when the non-terminal **L** is converted to a string of only terminals
- Hence remove the left recursion from the following production rule

  L -> L_2 , **id** | **id**

- This gives us -

L -> **id** L’

L’ -> , **id** L’ | *epsilon*

- The non-terminal **T** can be reintroduced by replacing *epsilon* by **: T**

The final grammar (currently without semantic rules) is as follows -

D -> L

T -> *integer* | *char*

L -> **id** L’

L’ -> , **id** L’ | : T

The non-terminal L can further be removed to give the following restructured grammar containing only synthetic attributes

D -> **id** L’ { Insert (**id**.name , L’.type) }

T -> *integer* { T.type = INTEGER }

| *char* { T.type = CHAR }

L’ -> , **id** L_2’ {

L’.type = L_2’.type

Insert (**id**.name , L_2’.type)

}

| : T { L’.type = T.type }