CS39003 Compilers Laboratory, Autumn 2024–2025

Lab Test 2

04–November–2024 07:15pm–08:15pm Maximum marks: 40

Roll no: **Name:**

Write in the respective spaces provided. Write syntactically correct codes (no credits for pseudocodes).

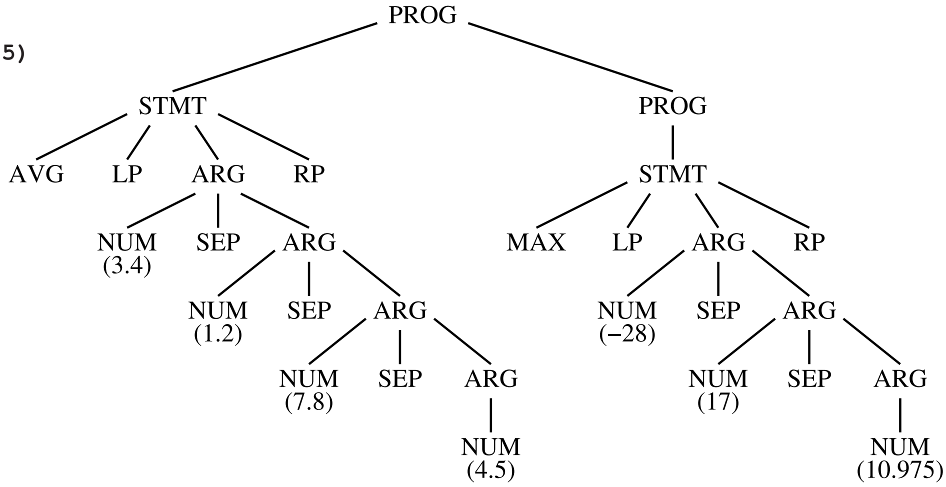
1. A database processing language supports statements for computing the maximum, minimum, and average of an arbitrary (but positive) number of arguments. The grammar for the language is given below. The start symbol is PROG. Other nonterminals are STMT and ARG. Lex returns the terminal tokens as MAX, MIN, AVG (keywords), NUM (double-valued), LP and RP (parentheses), and SEP (comma used as a separator between arguments).

$$\begin{array}{rcl} \text{PROG} & \rightarrow & \text{STMT} \mid \text{STMT PROG} \\ \text{STMT} & \rightarrow & \text{max (ARG)} \mid \text{min (ARG)} \mid \text{avg (ARG)} \\ \text{ARG} & \rightarrow & \text{NUM} \mid \text{NUM , ARG} \end{array}$$

The parse tree for the input

```
avg(3.4, 1.2, 7.8, 4.5)
max(−28, +17, 10.975)
```

is shown to the right.



Each node of the parse tree has the following data type. The attributes stored in each node are explained as comments.

```
typedef struct _node {
   int type;              /* Node type: PROG/STMT/ARG/MAX/MIN/AVG/NUM/LP/RP/SEP */
   int op;                /* Operation: MAX/MIN/AVG (inherited) */
   double value;          /* Numeric value of an operation (synthesized) */
   double partial;        /* Partial result computed so far (inherited) */
   int count;             /* Number of arguments (synthesized) */
   struct _node *C[4];    /* A node can have a maximum of four children */
} node;
```

Assume that the parse tree is provided to you. You do not have to write the Lex and Yacc files. The `type` fields at all the nodes are filled up by the parser as shown in the figure above. For NUM nodes, the `value` fields are populated too at parse time (the parenthesized numbers shown in the figure). The parser also creates the desired parent-child links using the entries of the child array `C`. If the number of children of a node is less than four, then the NULL pointer is stored as each absent child node. For example, `C[3]` is NULL at an ARG node with the production ARG → NUM, ARG, whereas `C[1]`, `C[2]`, and `C[3]` are all NULL at an ARG node with production ARG → NUM. Apart from the fields just mentioned, all other fields at the nodes of the parse tree are left uninitialized during parsing.

On the next two pages, fill out the details of a **recursive function** `eval()` for computing all max, min, and avg values from the parse tree. Use only the attributes in the `node` data type as specified above. The end result of an operation is to be printed by the corresponding `STMT` node. The root of the parse tree is stored in a (global) `node` pointer `root`. The `main()` function calls `eval(root)`. For the input in the above example, the output will be as follows. The output of a min operation will be similar.

```
Average of 4 items is 4.225000
Maximum of 3 items is 17.000000
```

The function `eval()` starts on Page 2, and continues to Page 3.

```
void eval ( node *p )
{
   switch (p -> type) {
      case PROG:                                                              (3)

        eval(p -> C[0]);
        if (p -> C[1] != NULL) eval(p -> C[1]);

      case STMT:
         p -> C[2] -> op = __________p -> C[0] -> type__________ ;           (1)
         if (p -> C[2] -> op == MAX) {                                        (4)

           p -> C[2] -> partial = -INFINITY;
           eval(p -> C[2]);
           p -> count = p -> C[2] -> count;
           p -> value = p -> C[2] -> value;
           printf("Maximum of %d items is %lf\n", p -> count, p -> value);

         } else if (p -> C[2] -> op == MIN) {                                 (4)

           p -> C[2] -> op = MIN;
           p -> C[2] -> partial = INFINITY;
           eval(p -> C[2]);
           p -> count = p -> C[2] -> count;
           p -> value = p -> C[2] -> value;
           printf("Minimum of %d items is %lf\n", p -> count, p -> value);

         } else if (p -> C[2] -> op == AVG) {                                 (4)

           p -> C[2] -> op = AVG;
           p -> C[2] -> partial = 0;
           eval(p -> C[2]);
           p -> count = p -> C[2] -> count;
           p -> value = p -> C[2] -> value / p -> count;
           printf("Average of %d items is %lf\n", p -> count, p -> value);

         }  /* End of if-else-else */
```

```
        case ARG:
```

(8)

```
          double newpartial;
          if (p -> op == MAX) {
             newpartial = (p->C[0]->value > p->partial) ? p->C[0]->value : p->partial;
          } else if (p -> op == MIN) {
             newpartial = (p->C[0]->value < p->partial) ? p->C[0]->value : p->partial;
          } else if (p -> op == AVG) {
             newpartial = p->partial + p->C[0]->value;
          }
          if (p -> C[2] == NULL) {
             p -> value = newpartial;
             p -> count = 1;
          } else {
             p -> C[2] -> op = p -> op;
             p -> C[2] -> partial = newpartial;
             eval(p -> C[2]);
             p -> value = p -> C[2] -> value;
             p -> count = 1 + p -> C[2] -> count;
          }
```

```
      }  /* End of switch */
   }
```

2. Consider expressions involving `*` (left-associative) and `^` (right-associative) with `^` having higher precedence than `*`. The following grammar for such expressions respects the associativity and precedence constraints. The start symbol is *E* (expression). Two other nonterminals are *F* (factor) and *B* (base). Lex returns the terminal tokens as NUM (signed integers), ID (same convention as in C), STAR and CARET (operators), and LP and RP (parentheses for grouping).

$$E \rightarrow F \mid E * F$$
$$F \rightarrow B \mid B \text{ ^ } F$$
$$B \rightarrow \text{NUM} \mid \text{ID} \mid (\, E \,)$$

Your task is to write a Yacc program for **printing** three-address codes for an input expression, where the temporaries are named as $1, $2, $3, and so on. An example is given below.

Input	Printed output

`123 * (a^(b*c)^(d*e^-12))^f * (x^y*z)`	`$1    =  b * c`
	`$2    =  e ^ -12`
	`$3    =  d * $2`
	`$4    =  $1 ^ $3`
	`$5    =  a ^ $4`
	`$6    =  $5 ^ f`
	`$7    =  123 * $6`
	`$8    =  x ^ y`
	`$9    =  $8 * z`
	`$10   =  $7 * $9`

Fill out the code of the Yacc program on the next page, for the given task. You do not have to write the Lex file. Assume that Lex prepares (as `yylval`) address pointers for the tokens NUM and ID. The Yacc program shows TMP as a token. Lex never generates a temporary, but this token is for getting a #define'd number for TMP.

```
%{
struct addr {
   int type;   /* NUM, ID, or TMP */
   int val;    /* Integer value for NUM, temporary number for TMP, 0 for ID */
   char *id;   /* Name of the variable for ID, NULL for NUM and TMP */
};

int tmpno = 0; /* Number of the temporary generated in the sequence 1, 2, 3, ...  */

/* Whenever a new temporary is to be created, call the following function.
   The parameters are address pointers for the arguments, and the operator (char). */
struct addr *gentmpaddr ( struct addr * , char , struct addr * ) ;
%}

%start E
%union { struct addr *ADDR; char SYMB; }
%token <ADDR> NUM ID TMP
%token <SYMB> STAR CARET LP RP
%type  <ADDR> E F B

%%

E       : F             { $$ = $1                       ; }   (1)

        | E STAR F      { $$ = gentmpaddr($1,$2,$3)     ; }   (1)
        ;

F       : B             { $$ = $1                       ; }   (1)

        | B CARET F     { $$ = gentmpaddr($1,$2,$3)     ; }   (1)
        ;

B       : NUM           { $$ = $1                       ; }   (1)

        | ID            { $$ = $1                       ; }   (1)

        | LP E RP       { $$ = $2                       ; }   (1)
        ;

%%

struct addr *gentmpaddr ( struct addr *A1, char op, struct addr *A2 )
{
   struct addr *A;

   /* Create the new temporary A */                                 (4)


   ++tmpno;
   A = (struct addr *)malloc(sizeof(struct addr));
   A->type = TMP;
   A->val = tmpno;
   A->id = NULL;


   /* Print the three-address instruction for A */                  (5)


   printf("$%-4d = ", tmpno);
   if (A1->type == NUM) printf("%d", A1->val);
   else if (A1->type == ID) printf("%s", A1->id);
   else if (A1->type == TMP) printf("$%d", A1->val);
   printf(" %c ", op);
   if (A2->type == NUM) printf("%d", A2->val);
   else if (A2->type == ID) printf("%s", A2->id);
   else if (A2->type == TMP) printf("$%d", A2->val);
   printf("\n");


   return A;
}
```