CS39003 Compilers Laboratory
Autumn 2026, Lab Test 2
29-Oct-2026, 07:15pm – 08:45pm
Maximum marks: 75

Roll No: _____

Name: _____

In this test, you work with a mini version of the MathC compiler for Mathorola CPU's. The grammar and the storage of three-address instructions are modified as follows. As in the lab assignments, variables are not declared anywhere. An undefined reference in an r-value of an assignment is substituted by the integer 0.

- Assume that there are only two basic data types (scalars): integer (INT) and rational (RTL) with the type hierarchy INT < RTL. As in the lab assignments, the widths of INT and RTL are 4 and 8 (bytes), respectively.
- All arithmetic operations (including the rational constructor *//*) are left-to-right associative. The additive operators **+** and **–** have the same precedence, and the multiplicative operators *****, ***/***, **%**, and ***//*** have the same precedence too. The multiplicative operators have higher precedence than the additive operators. For an operation a **op** b, the following conventions are adopted.

op	**Type-conversion rules for operands a and b**
+, **–**, *****, ***/***	If both a and b are of type INT, then an integer operation is carried out (`iadd`, `isub`, `imul`, `idiv`). If both a and b are of type RTL, then a rational operation is carried out (`radd`, `rsub`, `rmul`, `rdiv`). If one of a and b is INT and the other RTL, then the INT operand is converted to RTL using `int2rtl`, and then a rational operation is carried out.
%	Both operands of **%** should be of type INT. Rational operands (if any) are typecast to INT.
// (denoted by RDIV)	If both a and b are of type INT, then the rational constructor `mkrtl` is used. If only one of the operands is INT, it is first typecast to RTL, and then rational division (`rdiv`) is carried out. If both the operands are rational, then a rational division (`rdiv`) is carried out.

- Array elements `A[e]` may appear both in the l-value and as the r-value of an assignment statement. There will be no band-copy operation. If `A[e]` appears in the r-value, then the symbol table is looked at for the existence of an array named `A`. If no such entry is found (this includes the case that `A` is defined as a scalar), then the integer 0 is used for `A[e]` (an entry for `A` is <u>not</u> added to the symbol table). If `A` exists as an array of element type `t`, then `A[e]` is of type `t`, and is available at the offset `e` × `w` relative to (the base of) `A`, where `w` is the width of `t`.
- If `A[e]` is the l-value of an assignment statement, the name `A` is searched in the symbol table. If `A` is not found or is of a scalar type, then an ARR-type entry for `A` is added to the symbol table (replacing the old definition of `A` if any). The element type for `A` is set to the type of the r-value. If `A` already exists as an array of element type `t`, then the r-value is typecast to the type `t` (if it is not already `t`). The r-value is then stored at the offset `e` × `w` of `A`.
- Instead of printing the 3-address instructions, you store them in a table of ***<u>triples</u>*** (<u>not</u> quads). Recall that a triple consists of two operands and the operation. The result is assumed to be a temporary residing at the triple number *n*, and is denoted by (*n*). This is acceptable, since most intermediate results are temporaries. For an arithmetic operation a **op** b, the triple stores **op**, `a`, and `b` only. In particular, every 3-address arithmetic operation is stored in a triple (standing for a temporary). A store operation `var = e` of a scalar is stored as the triple (`=`, `var`, `e`). Here, `var` is an address referring to its symbol-table entry, and `e` can be an INT constant or a reference to a symbol-table entry or a triple storing the result of some computation. An array element `A[e]` is stored in two triples (*****, `e`, `w`) (`t` is the type and `w` is the width of an element of `A`), and (`tarr`, `A`, (reference to the last triple)). The second triple is used in r-values containing the element `A[e]`. If `A[e]` is an l-value as in `A[e] = r`, then the store instruction is stored as the triple (`=`, (second triple for `A[e]`), `r`). Use appropriate operation codes for the operator `=` (introduced later).

The triples are stored in a table TT (*to be implemented as an <u>array</u>*). After the entire input is parsed (and all triples are generated), the triples are printed in the same sequence as they are generated. Triple numbering starts from 100.

Input	**Output (triples generated)**
	`----------`
	`Operation  Argument 1  Argument 2`
	`----------`
`a = 8;`	`100 :  istr  a  8`
`A[0] = a // 7 // 3;`	`101 :  mkrtl  a  7`

	102 :	int2rtl	3	
	103 :	rdiv	(101)	(102)
	104 :	imul	0	8
	105 :	rarr	A	(104)
	106 :	rstrarr	(105)	(103)
A[5 % 4] = ((a + 6) * A[0] - 4) % 5;	107 :	irem	5	4
	108 :	iadd	a	6
	109 :	imul	0	8
	110 :	rarr	A	(109)
	111 :	int2rtl	(108)	
	112 :	rmul	(111)	(110)
	113 :	int2rtl	4	
	114 :	rsub	(112)	(113)
	115 :	rtl2int	(114)	
	116 :	irem	(115)	5
	117 :	int2rtl	(116)	
	118 :	imul	(107)	8
	119 :	rarr	A	(118)
	120 :	rstrarr	(119)	(117)

Part 1: Lexical analysis

Write a lex file to return the tokens. The tokens are **ID** (a C-style variable name), **NUM** (a signed integer), the single-letter operators **=**, **+**, **-**, *****, **/**, and **%**, a token **RDIV** for the operator **//**, and the single-letter punctuation symbols **;**, **(**, **)**, **[**, and **]**. Ignore all white spaces (spaces, tabs, and new lines). Note that a single assignment statement may span across multiple lines. For **NUM**, set the yacc union component `val`, and for **ID**, set the yacc union component `str`. `#include` the appropriate file for getting yacc's token numbers. Do not define them in the lex file. **[10]**

```
%{
#include "y.tab.h"
%}

ws      [ \t\n]+
id      [a-zA-Z_][a-zA-Z0-9_]*
num     [-+]?[0-9]+
ops     [-+*/%=]
rdiv    \/\/
punc    [;()\[\]]

%%

{ws}    { }
{id}    { yylval.str = strdup(yytext); return ID; }
{num}   { yylval.val = atoi(yytext); return NUM; }
{ops}   { return yytext[0]; }
{rdiv}  { return RDIV; }
{punc}  { return yytext[0]; }
.       { fprintf(stderr, "*** unrecognized character: '%c'\n", yytext[0]); }

%%

int yywrap ( ) { return 1; }
```

Part 2: Triple generation functions

We start with the definitions. ID, NUM, and RDIV are non-character tokens to be defined in the yacc file. We need some other values not used by yacc: INT (integer data type), RTL (rational data type), ARR (array data type), TRIPLE (an address of a temporary stored in a triple), INTWD (width of an INT), RTLWD (width of an RTL), and INVALID (an invalid return value). Assume that these are appropriately defined, so make no attempt to define them yourself.

Yacc uses C-style data types, so we define two C structures for addresses and triples. The address structure is used as types of several yacc non-terminals (like EXPR, TERM, FACTOR). The triple structure is not directly used by yacc.

```
struct _addr {
   int cat;             /* Category: can be ID, NUM, or TRIPLE */
   int type;            /* Data type: can be INT or RTL */
   int val;             /* stref for ID, lexval for NUM, triple number for TRIPLE */
};

typedef struct {
   char op[64];         /* The operation like idiv, rdiv, mkrtl, rarr, istr, rstrarr */
   struct _addr *arg1;  /* Pointer to the first argument (operand) */
   struct _addr *arg2;  /* Pointer to the second argument (operand), NULL if no second operand */
} triple;

triple TT[1000];        /* Global table of triples */
int nexttriple = 100;   /* Global variable: triple numbering starts from 100 */
```

We need some data types and functions for handling the symbol table ST. You do not have to implement the ST-related utilities. Assume the existence of the following functions on the global ST.

`stref`	A data type for a reference to an ST entry.
`stref stfind (char *varname);`	Returns an ST reference if `varname` is defined, or INVALID if not.
`stref stadd (char *varname, int t);`	Add a <u>new</u> entry to ST of scalar type `t`, and return a reference to this entry.
`stref staddarr (char *var , int t);`	Add a <u>new</u> array entry of element type `t` to ST, and return a reference to this entry.
`void stchtype (stref r, int t);`	Change to type `t` the type of the <u>valid</u> ST entry at reference `r`.
`int stgettype (stref r);`	Get the type (INT, RTL, or ARR) for a <u>valid</u> reference `r` to ST.
`int stgetelttype (stref r);`	Get the element type (INT or RTL) for a <u>valid</u> ARR-type entry `r` in ST.

The following function is needed for basic-type conversions. It is not directly called by any yacc action. The source type is `s` (stored in the address `a`) and the target type is `t`. If `s` = `t`, then the original `a` is returned. Otherwise, `int2rtl` or `rtl2int` (as the case is) is used to generate a new triple and a new address storing a reference to that triple. The new triple is stored in TT, and the new address is returned. This function demonstrates how addresses and triples are to be handled in the rest of this part.

```
struct _addr *typeconv ( struct _addr *a , int t )
{
   struct _addr *b;

   if (a -> type == t) return a;

   strcpy(TT[nexttriple].op, (t == INT) ? "rtl2int" : "int2rtl");
   TT[nexttriple].arg1 = a;
   TT[nexttriple].arg2 = NULL;

   b = (struct _addr *)malloc(sizeof(struct _addr));
   b -> cat = TRIPLE;
   b -> type = t;
   b -> val = nexttriple;
   ++nexttriple;

   return b;
}
```

Write the other address- and code-generation functions in the above format, as directed in the following parts **(a)**–**(f)**. <u>Use C syntax only.</u>

(a) The following function converts a **NUM** (an integer token `n`) to an address (no triples are generated). **[4]**

```
struct _addr *num2addr ( int n )
{




     struct _addr *a;

     a = (struct _addr *)malloc(sizeof(struct _addr));
     a -> cat = NUM;
     a -> type = INT;
     a -> val = n;

     return a;


}
```

(b) The following function converts an **ID** (a string token `var` for a scalar) to an address (no triples are generated). If the variable name is invalid (that is, undefined), then the integer 0 is to be returned as an address structure. **[4]**

```
struct _addr *id2addr ( char *var )
{



        stref r;
        struct _addr *a;

        a = (struct _addr *)malloc(sizeof(struct _addr));

        r = stfind(var);
        if (r == INVALID) {
           a -> cat = NUM;
           a -> type = INT;
           a -> val = 0;
        } else {
           a -> cat = ID;
           a -> type = stgettype(r);
           a -> val = r;
        }
        return a;



}
```

(c) The following function generates an array reference of the form `A [ offset ]` in a triple with operation `iarr` or `rarr`. A reference to this triple is returned. Before generating this triple, some other triples need to be generated. First, the array index `i` supplied to the function is the address of an expression. If its type is RTL, then it should be typecast to INT. Second, the index `i` is converted to the offset (a triple is needed to store the result of the offset calculation). If however `A[i]` is an invalid reference, then an address structure storing the integer 0 is to be returned. **[8]**

```
struct _addr *arr2addr ( char *var , struct _addr *i )
{

        stref r;
        struct _addr *a;
        int t, w;

        i = typeconv(i, INT);

        a = (struct _addr *)malloc(sizeof(struct _addr));

        r = stfind(A);
        if ( (r == INVALID) || (stgettype(r) != ARR) ) {
           a -> cat = NUM; a -> type = INT; a -> val = 0;
        } else {
           t = stgetelttype(r);
           w = (t == INT) ? INTWD : RTLWD;
           strcpy(TT[nexttriple].op, "imul");
           TT[nexttriple].arg1 = i;
           TT[nexttriple].arg2 = num2addr(w);
```

```
        a -> cat = TRIPLE; a -> type = INT; a -> val = nexttriple;
        ++nexttriple;

        strcpy(TT[nexttriple].op, (t == INT) ? "iarr" : "rarr");
        TT[nexttriple].arg1 = id2addr(A);
        TT[nexttriple].arg2 = a;

        a = (struct _addr *)malloc(sizeof(struct _addr));
        a -> cat = TRIPLE; a -> type = t; a -> val = nexttriple;
        ++nexttriple;
     }
     return a;




}
```

(d) The function below generates codes for binary arithmetic operations. It takes as input two operands (as addresses) and an operator. The result is stored in a temporary as a triple, and a reference to this triple is returned as an address pointer. As described on Page 1, the operations are of three categories. You need to handle these three cases separately. Notice that type conversions of the input operands may be performed before the operation is carried out. **[15]**

```
struct _addr *operation ( struct _addr *a , struct _addr *b , int op )
{
   struct _addr *c; /* to be returned */
   int atype = a -> type, btype = b -> type, ctype;

   switch (op) {  /* typecast when necessary, and determine ctype and the next operation */
      case '+':
      case '-':
      case '*':
      case '/':







       if (atype != btype) {
          ctype = RTL;
          if (atype == INT) a = typeconv(a, RTL); else b = typeconv(b, RTL);
       } else {
          ctype = atype;
       }
       TT[nexttriple].op[0] = (ctype == INT) ? 'i' : 'r';
       if (op == '+') strcpy((TT[nexttriple].op) + 1, "add");
       else if (op == '-') strcpy((TT[nexttriple].op) + 1, "sub");
       else if (op == '*') strcpy((TT[nexttriple].op) + 1, "mul");
       else if (op == '/') strcpy((TT[nexttriple].op) + 1, "div");
       else strcpy((TT[nexttriple].op) + 1, "???");







      break;
```

```
   case '%':


      ctype = INT;
      a = typeconv(a, INT);
      b = typeconv(b, INT);
      strcpy(TT[nexttriple].op, "irem");


      break;
  case RDIV:


      ctype = RTL;
      if ( (atype == RTL) || (btype == RTL) ) {
         a = typeconv(a, RTL);
         b = typeconv(b, RTL);
         strcpy(TT[nexttriple].op, "rdiv");
      } else {
         strcpy(TT[nexttriple].op, "mkrtl");
      }


      break;
      default: ctype = INVALID; strcpy(TT[nexttriple].op, "???"); break;
   } /* End of switch */
   /* Complete the final triple and the address structure c for returning */


      TT[nexttriple].arg1 = a;
      TT[nexttriple].arg2 = b;

      c = (struct _addr *)malloc(sizeof(struct _addr));
      c -> cat = TRIPLE;
      c -> type = ctype;
      c -> val = nexttriple;

      ++nexttriple;


   return c;
}
```

(e) The following function stores in ST the scalar referred to by `a`, with the name `var`. A new triple (with operation `istr` or `rstr`) is generated, but no address structure is needed to be returned. The ST entry (new or updated) will get the type of `a`. Use the symbol-table utilities as explained on Page 3. **[6]**

```
void asgnscalar ( char *var , struct _addr *a )
{


   stref r;

   r = stfind(var);

   if (r == INVALID) r = stadd(var, a -> type); else stchtype(r, a -> type);
```

```
    strcpy(TT[nexttriple].op, (a -> type == INT) ? "istr" : "rstr");
    TT[nexttriple].arg1 = (struct _addr *)malloc(sizeof(struct _addr));
    (TT[nexttriple].arg1) -> cat = ID;
    (TT[nexttriple].arg1) -> type = a -> type;
    (TT[nexttriple].arg1) -> val = r;
    TT[nexttriple].arg2 = a;

    ++nexttriple;


}
```

(f) The following function is used for storing the r-value referred to by `a`, to the array element `A[i]`. The details are explained in the last bulleted paragraph on Page 1. The final storage operation will be `istrarr` or `rstrarr`. **[8]**

```
void asgnarray ( char *A , struct _addr *i, struct _addr *a )
{









     stref r;
     int t;
     struct _addr *itriple;

     i = typeconv(i, INT);

     r = stfind(A);

     if ((r == INVALID) || (stgettype(r) != ARR)) {
        t = a -> type;
        staddarr(A, t);
     } else {
        t = stgetelttype(r);
        a = typeconv(a, t);
     }

     itriple = arr2addr(A, i);

     strcpy(TT[nexttriple].op, (t == INT) ? "istrarr" : "rstrarr");
     TT[nexttriple].arg1 = itriple;
     TT[nexttriple].arg2 = a;

     ++nexttriple;









}
```

Part 3: Syntax analysis

The grammar for mini MathC programs is given on the next page. PROG is the start symbol. The terminal symbols are in bold face. Write a yacc file for this grammar. The yacc actions would call the functions developed in Part 2. Specify the yacc stack type (a union), tokens, and the types of terminal and non-terminal symbols (whenever applicable). The grammar already conforms to the associativity and precedence of operators, so no yacc directives are necessary to set these. No need to write the `main()` function, the function(s) for printing TT, and the `#include`'s. **[20]**

PROG → PROG ASGN | ASGN
ASGN → **ID** = EXPR ; | **ID** [EXPR] = EXPR ;
EXPR → EXPR + TERM | EXPR – TERM | TERM
TERM → TERM * FACTOR | TERM / FACTOR | TERM % FACTOR | TERM // FACTOR | FACTOR
FACTOR → **NUM** | **ID** | **ID** [EXPR] | (EXPR)

```
%{
void yyerror ( char *msg ) { fprintf(stderr, "Error: %s\n", msg); exit(100); }
%}

%start PROG

%union {
   int val;
   char *str;
   struct _addr *addr;
}

%token RDIV
%token <str> ID
%token <val> NUM

%type <addr> EXPR TERM FACTOR

%%

PROG            : PROG ASGN                     { }
                | ASGN                          { }
                ;

ASGN            : ID '=' EXPR ';'               { asgnscalar($1, $3); }
                | ID '[' EXPR ']' '=' EXPR ';'  { asgnarray($1, $3, $6); }
                ;

EXPR            : EXPR '+' TERM                 { $$ = operation($1, $3, '+'); }
                | EXPR '-' TERM                 { $$ = operation($1, $3, '-'); }
                | TERM                          { $$ = $1; }
                ;

TERM            : TERM '*' FACTOR               { $$ = operation($1, $3, '*'); }
                | TERM '/' FACTOR               { $$ = operation($1, $3, '/'); }
                | TERM '%' FACTOR               { $$ = operation($1, $3, '%'); }
                | TERM RDIV FACTOR              { $$ = operation($1, $3, RDIV); }
                | FACTOR                        { $$ = $1; }
                ;

FACTOR          : NUM                           { $$ = num2addr($1); }
                | ID                            { $$ = id2addr($1); }
                | ID '[' EXPR ']'               { $$ = arr2addr($1, $3); }
                | '(' EXPR ')'                  { $$ = $2; }
                ;

%%
```