

CS39003 Compilers Laboratory
Autumn 2026
Assignment 8
Date of posting: 06-Oct-2026

Intermediate Code Generation: Expressions, Assignments, and Type Checking

Mathorola Semiconductors Inc. builds a series of CPU chips meant for mathematicians. These chips support not only integer and floating-point operations but also operations on rational and complex numbers. You do not need to worry in which format rational or complex numbers are actually stored (you also do not need to look into the storage format for integers and floating-point numbers). You only need to understand that arithmetic operations are permitted only on the same types of numbers. For operands of different types, a type conversion is to be applied to bring both the operators to the same type. But that is the job of the compiler.

The **MathC** compiler written for Mathorola CPU's makes the following conventions.

- The widening hierarchy is integer $\rightarrow$ rational $\rightarrow$ floating-point $\rightarrow$ complex.
- The compiler uses the short-cut names `int`, `rtl`, `flt`, and `cpx` for these four basic types. A type conversion is implemented by the instruction `src2dst`, where `src` and `dst` are of the above four types. For example, `rtl2flt` converts a number in the rational format to the floating-point format, whereas `cpx2int` converts a complex number to the integer format. You do not have to implement any of these type-conversion primitives, so you do not need to bother about how these type conversions are done.
- For the operators `+`, `-`, `*`, and `/`, type widening is carried out whenever necessary. For example, when the user wants to add a rational number `r` with a complex number `z`, the rational number `r` is first converted to a complex number (using `rtl2cpx`), and then the addition is carried out.
- The operator `%` calculates integer remainders. It requires both the operands to be integers. If any other type of number is presented as an operand, it is first converted to the `int` type. For example, during the computation `r % z`, both the rational operand `r` and the complex operand `z` are converted to `int` type, and then the remainder is calculated. This is unlike C where it is illegal to use `%` on floating-point operands.
- Signed integer and real (floating-point) constants are used as in C. Examples: `-56`, `3.1415926535`, `9.11e-31`.
- For constructing rational numbers, a non-associative operator `//` is used. The construction expression `a // b` requires both `a` and `b` to be integers. If `a` and/or `b` is/are of other type(s), type conversion to `int` is first carried out. Here, `a` and `b` can be integer or floating-point constants, a variable name, or a parenthesized expression. Note that `//` is non-associative, so expressions like `a // b // c` are not allowed, whereas `(a // b) // c` and `a // (b // c)` are allowed.
- For constructing complex numbers, a non-associative operator `@` is used. The construction expression `a @ b` requires both `a` and `b` to be floating-point values. If `a` and/or `b` is/are of other type(s), type conversion to `flt` is first carried out. Here, `a` and `b` can be integer or floating-point constants, a variable name, or a parenthesized expression. Note again that `@` is non-associative, so expressions like `a @ b @ c` are not allowed, whereas `(a @ b) @ c` and `a @ (b @ c)` are allowed.
- The combination `a @ b // c` is illegal too. You can use `(a @ b) // c` or `a @ (b // c)` instead (whichever you mean).
- Unparenthesized rational and complex constructors can be used one at a time in expressions (these are treated as factors). For example, `5 // 7 / 4 @ -3 / 1.2345` is allowed without any parentheses.
- The **MathC** compiler does not use variable declarations. Depending on the types of the operands, the type of the result is determined.
- An assignment to a variable stores the name and the type of the variable in the symbol table. An undefined variable may be used in an expression. Any undefined variable is treated as the integer 0.

In view of the above specifications, **MathC** designers use the following grammar for a (simple) program.

```
PROG      →   PROG ASGN | ASGN
ASGN      →   ID = EXPR ;
EXPR      →   EXPR + TERM | EXPR - TERM | TERM
TERM      →   TERM * FACTOR | TERM / FACTOR | TERM % FACTOR | FACTOR
FACTOR    →   ID | ( EXPR ) | REAL | COMPLEX
REAL      →   REALCONST | RATIONAL
REALCONST →   INTCONST | FLTCONST
RATIONAL  →   RCOMP // RCOMP
COMPLEX   →   RCOMP @ RCOMP
RCOMP     →   REALCONST | ID | ( EXPR )
```

You need to implement an intermediate code generator for programs generated according to the above grammar.

- Use the temporaries **t1**, **t2**, **t3**, ... in that sequence.
- Binary operations generate codes as follows.

```
t20 = 5 + a
t59 = t47 * t58
```

- Any operation assumes that the operands are of the same type. If not, one or both of them is/are converted to the same type. The type-conversion three-address expressions are exemplified below.

```
t35 = int2cpx n
t46 = cpx2rat t31
```

- For constructing rational and complex numbers, use the following special three-address instructions.

```
t81 = mkrtl 50 t77
t92 = mkcpx 3.1416 f123
```

- If an undefined variable **v** is referenced inside a formula, use the integer **0** for **v**. For example, the complex construct **z = v @ 0.12345** using the undefined variable **v** generates the following three-address instructions. Note that an entry for **v** is not introduced in the symbol table in this case. An entry is created in the symbol table only for an l-value of an assignment.

```
t61 = int2flt 0
t62 = mkcpx t61 0.12345
z = t63
```

- Keep the current type of each defined variable in the symbol table. During an assignment operation, the type of the l-value is stored (or overwritten if the variable was already defined) as the type of the r-value.

Prepare the following files.

- A C/C++ file for implementing the symbol-table utilities.
- A C/C++ file for the functions on addresses. You need to store the category, the type, and a reference for each address. The category refers to what kind of storage is used: a variable name (a symbol-table reference), a constant, or a temporary. For an address of the variable-name category, the type is determined by the symbol-table entry for the variable. For a constant, the type is either `int` or `flt`. Moreover, its value must be stored in the address itself (another alternative is to use a constant table akin to the symbol table; in that case, a reference to the constant-table entry is to be stored in the address). Finally, an address of the temporary category should store the number of the temporary. Its type is determined by the three-address instruction that generates the temporary.

Notice that you need to use a pointer to an address in the yacc file. So an address should be implemented as a C structure, and the types of several non-terminals would be pointers to such structures. In view of these, it is expedient to write several functions: one that converts a variable name (in an r-value) to a variable-category address, a function to generate constant-category addresses, a function to generate a temporary-category

addresses as a result of a three-address instruction (a binary operation, a type-conversion operation, or a rational/complex construction), a function to handle an assignment, and so on.

Alongside the address-related operations, this file should generate the three-address instructions. These should be printed as the output of your code. You do not need to store the three-address instructions in a table of quads or triples. In your three-address instructions, use variable names for variable-category addresses. This is for promoting readability. Moreover, variables do not have fixed data types in this assignment, so any fixed amount of storage cannot be used for a variable. That is, you do not need to compute or print the offsets of variables.

- You may have a separate header file storing all the data-type declarations, and function prototypes.
- A lex file that keeps on returning tokens (variable names, numeric constants, operators, and punctuation symbols). In earlier assignments, you used the new-line character as a statement terminator. In this assignment, you use semi-colon as the statement terminator. This means that a single assignment statement may span across multiple lines, so `\n` is to be ignored as a white space.

You do not need to handle the problem associated with signed numbers, as pointed out in an earlier assignment. For example, the expression `5-6+7.8*9` will let lex identify `-6` and `+7.8` as signed numbers. This problem can be fixed, but you do not have to do so. Assume that the user will always supply enough spaces so that this confusion cannot bug lex. Example: `5 - 6 + 7.8 * 9`.

- A yacc file for parsing under the given grammar. In the actions, only call the address-related functions written in other files. The grammar itself handles precedence and associativity constraints of standard arithmetic, so your yacc file does not have to specify those again.
- The quintessential makefile with `compile`, `run` (on `input.txt`), and `clean` targets.

Combine all your files (do not include the machine-generated files) in a tar/tgz/zip archive. Submit only that archive.

Sample Output

Input	Output
<pre>a = ((1 + 2 * 3) * 4 + (5 * 6 % 7) - 8) / 9 ;</pre>	<pre>t1 = 2 * 3 t2 = 1 + t1 t3 = t2 * 4 t4 = 5 * 6 t5 = t4 % 7 t6 = t3 + t5 t7 = t6 - 8 t8 = t7 / 9 a = t8</pre> // ST[0] refers to variable a of type int
<pre>f = ((1 + 2 * 3) * 4 + (5. * 6 % 7) - 8) / 9 ;</pre>	<pre>t9 = 2 * 3 t10 = 1 + t9 t11 = t10 * 4 t12 = int2flt 6 t13 = 5.000000 * t12 t14 = flt2int t13 t15 = t14 % 7 t16 = t11 + t15 t17 = t16 - 8 t18 = t17 / 9 f = t18</pre> // ST[1] refers to variable f of type int
<pre>a = ((1 + 2 * 3) * 4 + (5 * 6 % 7) - 8.) / 9 ;</pre>	<pre>t19 = 2 * 3 t20 = 1 + t19 t21 = t20 * 4 t22 = 5 * 6 t23 = t22 % 7 t24 = t21 + t23 t25 = int2flt t24 t26 = t25 - 8.000000 t27 = int2flt 9 t28 = t26 / t27 a = t28</pre> // ST[0] refers to variable a of type flt
<pre>r = ((1 + 2 * 3) * 4 + (5 * 6 % 7) - 8.) // 9 ;</pre>	<pre>t29 = 2 * 3 t30 = 1 + t29 t31 = t30 * 4 t32 = 5 * 6 t33 = t32 % 7 t34 = t31 + t33 t35 = int2flt t34 t36 = t35 - 8.000000 t37 = flt2int t36 t38 = mkrtl t37 9 r = t38</pre> // ST[2] refers to variable r of type rtl
<pre>z = ((1 + 2 * 3) * 4 + (5 * 6 % 7) - 8.) @ 9 ;</pre>	<pre>t39 = 2 * 3 t40 = 1 + t39 t41 = t40 * 4 t42 = 5 * 6 t43 = t42 % 7 t44 = t41 + t43 t45 = int2flt t44 t46 = t45 - 8.000000 t47 = int2flt 9 t48 = mkcpx t46 t47 z = t48</pre> // ST[3] refers to variable z of type cpx
<pre>a = (a + 10.) * (f + r - a * z) - b - z ;</pre>	<pre>t49 = a + 10.000000 t50 = int2rtl f t51 = t50 + r t52 = flt2cpx a t53 = t52 * z t54 = rtl2cpx t51 t55 = t54 - t53 t56 = flt2cpx t49 t57 = t56 * t55 t58 = int2cpx 0 t59 = t57 - t58 t60 = t59 - z a = t60</pre> // ST[0] refers to variable a of type cpx