# CS39003 Compilers Laboratory
# Autumn 2026
# Assignment 11
# Date of posting: 03-Nov-2026

---

## Handling functions

This assignment continues to develop the MathC compiler tailored to Mathorola CPU's. As earlier (LA8 – LA10), we use the four basic data types `int`, `rtl`, `flt`, and `cpx` of respective widths 4, 8, 8, and 16 (bytes). For simplicity, we now do not use arrays and also branching and looping statements. We instead add constructs to declare and call functions. We also need to place all the variables in memory locations (absolute and relative) generated during compile time. To be more precise, a variable `x` is no longer written as `x` in the 3-address code, but by its memory location `[loc]` where `x` resides. Here, `[loc]` can be either a fixed address like `[1000456]`, or an offset in the control stack like `[SP(248)]` relative to the current control-stack pointer `SP`. The final target code usually does not have symbol tables, so this assignment is a step forward toward the generation of target codes.

In order to cater to the needs of explicit memory placements of variables, we can no longer let variables appear and/or change their types during runtime. In other words, all variables and functions must be declared or defined before they are used. The types of declared variables (and functions) are not permitted to change dynamically. We use the standard scoping rules as in C, namely, a function has access to its local variables (in addition to the universally accessible global variables). A variable name in a function is first searched in its local address space, and if not found in the global address space. Unlike C, we will not have a `main()` function. Instead, the global code will reside after the declaration of all global variables and functions. Function definitions are never nested (that is, you cannot declare a function inside another function). Function calls can however be nested.

We make another change in the old grammar. Instead of the `mkrtl` and `mkcpx` operations, we will specify rational and complex *constants* as `<intconstant>//<intconstant>` and as `<realconstant>@<realconstant>`. These constants are stored like `int` and `flt` constants (without calling the `mkrtl` and `mkcpx` operations).

The following grammar defines a program (PROG) in our language. The terminal symbols are shown in color red.

```
PROG      →   DECL SLIST                          // Program (the start symbol)

DECL      →   DECL VDEF | DECL FDEF | ϵ           // Declaration of variables and functions
VDECL     →   VDECL VDEF | ϵ                      // Declaration of only variables

VDEF      →   TYPE VLIST ;                        // Declaration of a set of variables of the same type
VLIST     →   VLIST , ID | ID                     // VLIST is non-empty comma-separated list of names

FDEF      →   TYPE ID ( PLIST ) { FBODY } |       // Definition of a function that returns a basic data type
              void ID ( PLIST ) { FBODY }         // Definition of a function that returns nothing (void)
FBODY     →   VDECL SLIST                         // No nested function declarations in a function body
PLIST     →   ϵ | NEPLIST                        // List of (formal) parameters (may be empty)
NEPLIST   →   NEPLIST , TYPE ID | TYPE ID         // Non-empty comma-separated parameter list

SLIST     →   SLIST STMT | ϵ                      // List of statements
STMT      →   ASGN | RTRN | FCALL                 // Assignment and return statement, and function call
ASGN      →   ID = EXPR ;                         // Standard assignment statement
RTRN      →   return ; | return EXPR ;            // Return statements without and with a return value

EXPR      →   EXPR + TERM | EXPR - TERM | TERM    // Usual grammar for expressions
TERM      →   TERM * FACTOR | TERM / FACTOR | TERM % FACTOR | FACTOR
FACTOR    →   NUM | ID | ( EXPR ) | ID ( ALIST ) // The last production stands for a function call
FCALL     →   ID ( ALIST ) ;                      // Function-call statement without an assignment

ALIST     →   ϵ | NEALIST                         // List of actual arguments (may be empty)
NEALIST   →   NEALIST , EXPR | EXPR               // Non-empty comma-separated list of arguments

NUM       →   INT_CONST | RTL_CONST | FLT_CONST | CPX_CONST   // Numeric constants
TYPE      →   int | rtl | flt | cpx                           // Keywords for basic types
```

## Part 1

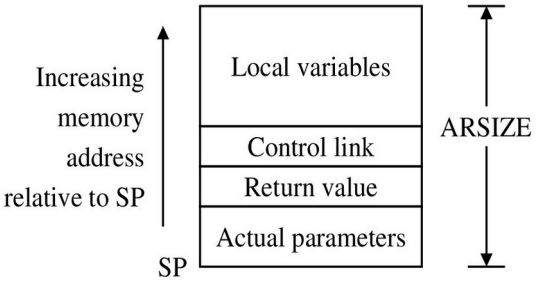
In this part, you handle the declaration of variables and functions in the input MathC program. In Part 2, you handle all the statements in the input MathC program, that may include function calls.

Start by updating your LA8/LA9 lex and yacc files to handle the modified grammar. Note that the 3-address code is to be stored in a table QT of quads (as in LA9 and LA10). Use the same typed Operation names as in LA9 and LA10 (like `fadd`, `cstr`, `rtl2int`). We use a copy operation `tcpy` (`t` is the type) from a memory location to a temporary. As stated earlier, all memory locations are presented as `[loc]`, where `loc` is an absolute address or an offset relative to `SP`.

Every function has its own name space stored in a private symbol table. The global symbol table stores information about the global variables and the functions. An entry in a symbol table (global or function-specific) stores (a) the name of the entity, (b) the category of the entity (variable or function), (c) the type (relevant for variables only), and (d) a reference (memory location for a variable, or a reference to the symbol table for a function).

All global variables are stored in the static area (assumed to begin from the location 1,000,000), and constitute the *global activation record*. In the absence of a `main()` function, the global activation record marks the beginning of the control stack, and will always reside there in this position throughout the execution of the MathC program. This means that the control-stack pointer SP (that marks the beginning of the topmost activation record) is initialized to 1,000,000. This is done in run time, so you do not need to generate any 3-address instruction for this.

For a function too, its symbol table stores the relevant information about its activation record AR. The organization of the activation record of a function is given in the following figure. Assume that there are sufficiently many registers, so we do not need to store temporaries in an activation record.




As an example, consider the following function.

```
rtl myfunc ( int n , cpx z, flt f )
{
    rtl r, s;
    int i;

    /* code of the function */
    . . .
}
```

For this function, the parameters `n`, `z`, and `f` reside at offsets 0, 4, and 20. The parameters take a total space of 4 + 16 + 8 = 28 bytes. The return value is stored at the offset 28, and has a size of 8 bytes. The control link is a quad number, and is assumed to be stored as an integer, so this is stored in bytes 36 – 39 in the `AR` of `f`. Then, the local variables `r`, `s`, and `i` have offsets 40, 48, and 56, respectively. The size of this AR (denoted by ARSIZE) is 60.

In this part, write the yacc actions for the productions under DECL (variable and function declarations). You do not need to worry about the yacc actions for the productions under SLIST. That means that you do not generate any intermediate code (quads) in this part.

You need to use several markers to aid you in doing the necessary bookkeeping. Functions are always added to the global symbol table, whereas a variable may be added to any symbol table. All parameters of a function are added to the beginning of its AR. You need to set a reference to the current symbol table where a name is added. By default, it is the global symbol table. As soon as you see the prefix TYPE in the global section of the input, you remember the type (of the variable or the value returned by a function). This can be stored in a global variable of your program by a marker (see the yacc slides). This type is followed by an ID. At this moment, you do not know whether a variable list

or a function declaration ensues. If the next token is ID, then it is a variable list, whereas if the next token is (, then this is a function definition. This decision is handled by yacc. You only want to add another marker after you see the token (. Store the name of the function in the global symbol table, create a new symbol table for the function being defined, and set the current symbol table to this newly created table. Add the formal parameters and the local variables to the new symbol table. At the end of the function declaration, restore the global symbol table as the current symbol table.

When you add a parameter or a variable to a symbol table ST, you also calculate and store its AR offset, in its ST entry. The entry of a function in the global ST does not have an offset, but stores a reference to its symbol table.

You may store the type of a function (its input-output types) in its ST entry. Another option (recommended) is to use a separate table FT for functions. An entry in FT for a function would store: (a) the name of the function (optional, because this name is already stored in the global ST), (b) the number of parameters, (c) the return type, (d) the offset of the return value, (e) the offset of the control link, (f) the size ARSIZE of the activation record, and (g) the start and the end quad numbers for the code of the function. The start quad number is needed for making a jump to the function as initiated by a call to the function. The symbol table of the function stores the parameter names and types, and treat them as local variables, so these types need not be stored in the FT entry of the function.

Using the global symbol table and the symbol tables of the defined functions, print how the activation records will be organized in run time (use the format as illustrated in the Sample Output for Part 1). Note that symbol tables exist only during compilation time, whereas activation records exist only during run time. Symbol tables have names and the offsets of the variables, whereas activation records are chunks of memory without any names. In Part 2, you will replace the names by explicit memory addresses in the output 3-address code. However, for the purpose of printing the activation records of this part, you still have access to the names, so print them along with the offsets.

Combine all your source files (and a mandatory *makefile*) in a tar/tgz/zip archive, and submit that archive.

## Part 2

In this part, you generate the 3-address code (stored in a table QT of quads), and print them. Handling arithmetic operations and assignment statements is similar to as in LA9. You only need to make a small change, because in our revised grammar, // and @ are no longer non-associative operators, so the expression grammar does not include them. They can be handled at the lex level. Moreover, the r-value is typecast to the type of the l-value during as assignment. This is unlike as in the assignments LA8 – LA10.

### Handling function calls

We may have standalone function calls or function calls within expressions. Both are treated in essentially the same manner. For any function call, the actual parameters (expressions calculated by the caller) are typecast (if necessary) to the corresponding parameter types as stored in the symbol table of the callee.

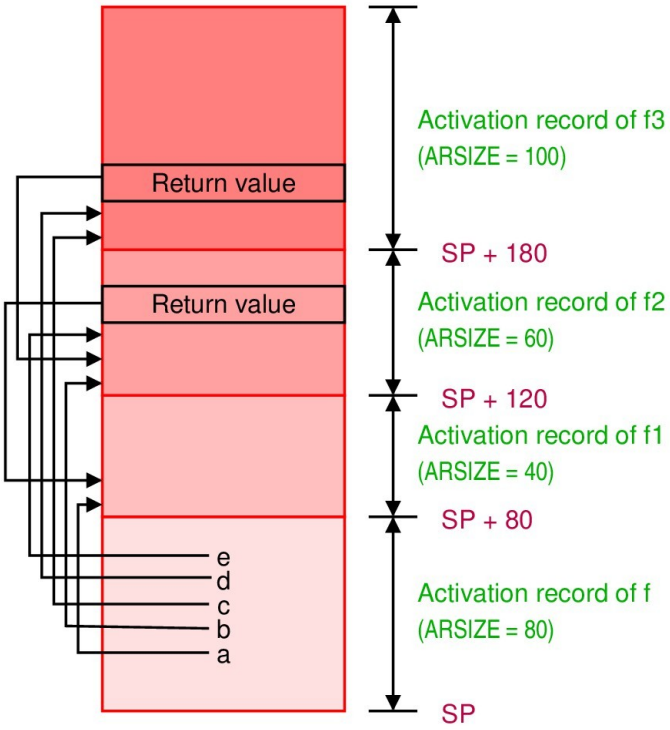
First, consider a simple call `g(`$e_1$`, `$e_2$`, . . . )` made by a function or global code `f`, where $e_1$, $e_2$, … are expressions that do not involve any function call. Currently, the activation record of `f` is at the top of the control stack, and its beginning is located at the current value of the pointer `SP`. Let ARSIZE be the size of the activation record of `f`. When `g` will be called, `SP` will be incremented by ARSIZE. But preparing the parameters $e_1$, $e_2$, … for `g` requires access to variables in the current activation record (of `f`) and/or to global variables. So `SP` is not incremented before the actual call of `g` is made. The evaluated expression $e_k$ (an address) is copied to the relative address ARSIZE + $\text{offset}_k$, where $\text{offset}_k$ is the AR offset of the `k`-th parameter of `g`, as stored in the symbol table for `g`. When all parameters are copied, the call of `g` can be initiated (the return value if any will be read from the address SP + control_link_offset_of_g). Before that, `SP` is incremented by ARSIZE, and an unconditional jump (`goto`) to the first quad number of the code for `g` is added to the code for `f`. A return from `g` goes to the next 3-address instruction of `f` (the decrement of `SP` by ARSIZE).

This implementation becomes complicated if we have nested calls. Consider the following function call inside a caller `f` (which may be the global code or another function).

```
f1(a, f2(b, f3(c,d), e))
```

Here, f1 and f3 are functions with two parameters, and f2 is a function with three parameters. In run time, f will call f3 first, and then f2, and finally f1. However, the input is processed from left-to-right, so the code of f will see the actual parameter a of f1 and the actual parameter b of f2 before encountering the call of f3. One way to handle this is to maintain an external list of computed parameters (their addresses actually) for every function call. Before a function call, the formal parameters are populated by the contents of the corresponding list, and then the call to that function is made. However, since our functions do not use dynamically sized variables, their ARSIZE's are fixed and known beforehand (already stored in FT). This allows us to take a (slightly) simplified approach to deal with nested function calls. Note also that at compile time, a, b, c, d, and e are to be resolved using the symbol table of f and the global symbol table. So change SP just before a function call, and restore it back immediately after the call returns.

In our approach, we store the computed parameters of all nested calls together in the control stack itself (instead of in external list(s)). This means that the activation records of multiple calls can coexist in the control stack, and are modified together before any call is made. For the above example, suppose that the ARSIZE of f, f1, f2, and f3 are respectively 80, 40, 60, and 100. The current value of SP marks the beginning of the activation record of f. The following figure demonstrates how the nested calls are made. Assume that a, b, c, d, and e are local variables found in the activation record for f.




The actual parameters for f1, f2, and f3 must be written in separate locations. In the call f1(a, f2(b, f3(c,d), e)), f1 is encountered in the input first, and the parameter a for f1 is copied relative to SP + 80 (the ARSIZE for f is 80). Then the call of f2 is encountered in the input. Since the ARSIZE for f1 is 40, copying of parameters of f2 will be relative to SP + 80 + 40 = SP + 120. Then the call for f3 is detected in the input as the second argument of f2. The ARSIZE for f2 is 60, so the parameters c and d for f3 are copied relative to SP + 120 + 60 = SP + 180.

In the input, the innermost call f3(c,d) appears first. A reduction will take place at this point. This reduction will increment SP by 180, and transfer control to the first quad of the code of f3. When this call returns, SP will be decremented by 180. The third argument of f2 will be relative to SP + 120. Then the call of f2 ends in the input. Now, SP will incremented by 120 before the call of f2, and decremented by 120 after the call. The outermost call of f1 ends last in the input. This time the increment and decrement of SP is by 80.

The offsets OFF 80, 120, 180 relative to SP can be maintained using a *compile-time* call stack CS. This stack is not the same as the control stack that will exist during *run time*. At the beginning of the code of f, CS is initialized by a reference to f. In each stack element, we also maintain the information about how many parameters are populated so far. Moreover, OFF is initialized to 0. Whenever a function call is encountered in the input, the reference to the callee is pushed to the top of CS, and OFF is incremented by the ARSIZE of the earlier top (the current caller) of CS. When an end-of-call is encountered in the input, the top of CS is popped, and OFF is decremented by the ARSIZE of the new top. With this mechanism, writing formal parameters will be at [PARAMOFFSET(SP)], where PARAMOFFSET will be the

current value of OFF plus the offset of the parameter as stored in the symbol table of the callee. The increment and decrement of SP will be by the current value of OFF during a function call or return.

Notice that OFF is a variable in your program. In the generated 3-address code, use explicit constants for OFF. Notice also that the nested calls (like `f3(c,d)`) may make function calls from their code. These function calls will work with the incremented value of SP as received by the call of `f3`. That is, the code for `f3` will use the control stack above its own activation record, and will not interfere with the space already allocated to the outer calls of `f2` and `f1`.

You need to use markers for initializing CS and OFF at appropriate places. Their updates can be in the actions of function calls and returns.

### Handling returns from functions

The work associated with a return from a function is jointly shared by the caller and the callee. There are two kinds of return statements: (a) `return expr;` (applicable for functions with a non-`void` return type) and (b) `return;` (applicable for functions with return type `void`).

If the return type of a function is not `void`, the callee handles a return of kind (a) by writing `expr` at the return offset relative to the SP that it gets. If the type of `expr` does not match the return type of the callee, then `expr` is first typecast to the return type before the write is performed. If the return type is `void`, then `expr` is ignored. For an expression-less return (the second kind), no copy is made (even if the return type is not `void`). Both the kinds of return statements then do a crucial thing for returning control to the caller. The control link in the current activation record is looked at, and an unconditional jump is made to the quad number written by the caller before the call was made. This jump is rather necessary, so if a function does not end with an explicit return statement, it should be implicitly added at the end of the function body. For the global code, the return code is called `halt` (this would initiate a system call to the OS).

After the return, the caller decrements SP (by OFF) and OFF (by an appropriate ARSIZE), as explained earlier. After that, the caller checks whether or not the callee's return type is `void`. If that is not `void`, then the caller copies (from the return offset of the callee) the returned value, to a temporary (whose type will be the return type of the callee). For a callee of return type `void`, no such copy is made. After this, the caller goes ahead with the rest of its body.

### Handling the quads

Store the 3-address instructions for the statement lists in the sequence as they are generated from the input. Note that a variable or a function must be defined earlier than it is used. This does not preclude the possibility of self-recursive functions (because its activation record is prepared before its statement list is translated). The only case of mutually recursive functions (like `egg()` and `chicken()`) cannot be handled without introducing the concept of function prototypes (not included in our grammar for simplicity).

After the entire input is parsed, print all the activation records (this was already done in Part 1). Then print the quads in the sequence as generated by your updated program for Part 2. Quad numbering starts from 1. Quad 0 is reserved for a special purpose, namely to store an unconditional jump to the start quad of the global code. Constants are specified by the prefix `#` (like `#123`, `#6.543210`, `#<5//7>`, and `#<1.23456-i2.345678>`). Memory references are enclosed within square brackets (like `[1000456]` and `[20(SP)]`). Recall that the static area begins at memory location 1,000,000. Finally, the contents of a memory address is referred to by a preceding `*` (like `*56(SP)`).

### Submission

Combine all your source files (and a mandatory *makefile*) in a tar/tgz/zip archive, and submit that archive. The submitted files must generate the outputs for Part 1 and Part 2 together.

## Sample Output

The output for the following sample input is supplied.

```
flt a, b, c;

int f1 ( rtl a )
{
   return a + b * c;
}

int i, j, k;

rtl f2 ( int n , flt f )
{
   int k, l, m;
   flt g;

   k = n;
   l = n + i;
   m = n + j;
   g = f * f;
   return f1((k + l) / m - g + c);
}

cpx f3 ( int n, rtl r, flt f, cpx c )
{
   cpx z1, z2;

   z1 = n * f1(r);
   z2 = f / f1(f2(n,f)/f1(r)-f3(i, f2(f1(r),f), f - 1, c * c));

   return z1 + z2;
}

rtl r, s, t;
cpx x, y, z;

void init ( )
{
   i = 123;
   j = 234;
   k = 345;

   r = 98 // 87;
   s = 87 // 76;
   t = r * s;

   a = 4.567;
   b = 5.678;
   c = b - a;

   x = 1;
   y = 0@1;
   z = 6.78@-7.89;
}

init();

z = f3(i+j+k, f2(21e1, f1(r)), b, y + 2.1 @ -1.2);
```

These functions do not perform anything meaningful. Self-recursive functions like `f3` do not terminate in run time. You need a condition check to stop recursion. That requires conditional statements and/or loops. You know how to handle branching and looping (as in LA10). You do not need to repeat the same tasks again in LA11. Functions are important here. Testing them requires some statements, so we use the simplest type of them (assignments).

In the sample output below, some (not all) 3-address instructions are annotated. These annotations are not part of the input file. Moreover, the global declarations of variables are moved forward before the function definitions. This is done for synchronizing the (rearranged) input with the output. Your compilation program should work so long as it does not encounter a name not already defined. Consult the output from Part 1 to verify the various offsets in the activation records.

## Annotated input

```
                                    /* Global variables */
                                    flt a, b, c;

                                    int i, j, k;

                                    rtl r, s, t;

                                    cpx x, y, z;



                                int f1 ( rtl a )
                                    {
                                return a + b * c;
                                    }


                            rtl f2 ( int n , flt f )
                                    {
                                int k, l, m;
                                      flt g;

                                      k = n;
                                  l = n + i;
                                  m = n + j;
                                  g = f * f;
                          return f1((k + l) / m - g + c);
                                    }

                        cpx f3 ( int n, rtl r, flt f, cpx c )
                                    {
                                     cpx z1, z2;

                                    z1 = n * f1(r);
z2 = f / f1(f2(n,f)/f1(r)-f3(i, f2(f1(r),f), f - 1, c * c));

                                    return z1 + z2;
                                    }
```

## Output

```
*********************************************************************************
***                                  PART 1                                   ***
*********************************************************************************

+++ Global activation record
    (All offsets are relative to 1000000)
    [Variable] a (flt) at location 0
    [Variable] b (flt) at location 8
    [Variable] c (flt) at location 16
    [Variable] i (int) at location 24
    [Variable] j (int) at location 28
    [Variable] k (int) at location 32
    [Variable] r (rtl) at location 36
    [Variable] s (rtl) at location 44
    [Variable] t (rtl) at location 52
    [Variable] x (cpx) at location 60
    [Variable] y (cpx) at location 76
    [Variable] z (cpx) at location 92
    Activation record size = 108

+++ Activation record for function f1()
    (All offsets are relative to SP)
    [Parameter] a (rtl) at offset 0
    [Return value] (int) at offset 8
    [Control link] at offset 12
    Activation record size = 16

+++ Activation record for function f2()
    (All offsets are relative to SP)
    [Parameter] n (int) at offset 0
    [Parameter] f (flt) at offset 4
    [Return value] (rtl) at offset 12
    [Control link] at offset 20
    [Variable] k (int) at location 24
    [Variable] l (int) at location 28
    [Variable] m (int) at location 32
    [Variable] g (flt) at location 36
    Activation record size = 44

+++ Activation record for function f3()
    (All offsets are relative to SP)
    [Parameter] n (int) at offset 0
    [Parameter] r (rtl) at offset 4
    [Parameter] f (flt) at offset 12
    [Parameter] c (cpx) at offset 20
    [Return value] (cpx) at offset 36
    [Control link] at offset 52
    [Variable] z1 (cpx) at location 56
    [Variable] z2 (cpx) at location 72
    Activation record size = 88

+++ Activation record for function init()
    (All offsets are relative to SP)
    [Return value] (void) at offset 0
    [Control link] at offset 0
    Activation record size = 4

*********************************************************************************
***                                  PART 2                                   ***
*********************************************************************************

                ------------------------------------------------------------------
                Operation       Argument 1              Argument 2              Result
                ------------------------------------------------------------------
          0:    goto                                                            110

Code for function f1()
          1:    fmul            [1000008]               [1000016]               t1
          2:    rtl2flt         [0(SP)]                                         t2
          3:    fadd            t2                      t1                      t3
          4:    flt2int         t3                                              t4
          5:    istr            t4                                              [8(SP)]
          6:    goto                                                            *12(SP)

Code for function f2()

          7:    istr            [0(SP)]                                         [24(SP)]

          8:    iadd            [0(SP)]                 [1000024]               t5
          9:    istr            t5                                              [28(SP)]

         10:    iadd            [0(SP)]                 [1000028]               t6
         11:    istr            t6                                              [32(SP)]

         12:    fmul            [4(SP)]                 [4(SP)]                 t7
         13:    fstr            t7                                              [36(SP)]

         14:    iadd            [24(SP)]                [28(SP)]                t8
         15:    idiv            t8                      [32(SP)]                t9
         16:    int2flt         t9                                              t10
         17:    fsub            t10                     [36(SP)]                t11
         18:    fadd            t11                     [1000016]               t12
         19:    flt2rtl         t12                                             t13
         20:    rstr            t13                                             [44(SP)]
         21:    iset            #24                                             [56(SP)]
         22:    iadd            SP                      #44                     SP
         23:    goto                                                            1
         24:    isub            SP                      #44                     SP
         25:    icpy            [52(SP)]                                        t14
         26:    int2rtl         t14                                             t15
         27:    rstr            t15                                             [12(SP)]
         28:    goto                                                            *20(SP)
```

```
Code for function f3()

      29:   rstr          [4(SP)]                                   [88(SP)]
      30:   iset          #33                                       [100(SP)]
      31:   iadd          SP                    #88                 SP
      32:   goto                                                    1
      33:   isub          SP                    #88                 SP
      34:   icpy          [96(SP)]                                  t16
      35:   imul          [0(SP)]               t16                 t17
      36:   int2cpx       t17                                       t18
      37:   cstr          t18                                       [56(SP)]

      38:   istr          [0(SP)]                                   [104(SP)]
      39:   fstr          [12(SP)]                                  [108(SP)]
      40:   iset          #43                                       [124(SP)]
      41:   iadd          SP                    #104                SP
      42:   goto                                                    7
      43:   isub          SP                    #104                SP
      44:   rcpy          [116(SP)]                                 t19
      45:   rstr          [4(SP)]                                   [104(SP)]
      46:   iset          #49                                       [116(SP)]
      47:   iadd          SP                    #104                SP
      48:   goto                                                    1
      49:   isub          SP                    #104                SP
      50:   icpy          [112(SP)]                                 t20
      51:   int2rtl       t20                                       t21
      52:   rdiv          t19                   t21                 t22
      53:   istr          [1000024]                                 [104(SP)]
      54:   rstr          [4(SP)]                                   [236(SP)]
      55:   iset          #58                                       [248(SP)]
      56:   iadd          SP                    #236                SP
      57:   goto                                                    1
      58:   isub          SP                    #236                SP
      59:   icpy          [244(SP)]                                 t23
      60:   istr          t23                                       [192(SP)]
      61:   fstr          [12(SP)]                                  [196(SP)]
      62:   iset          #65                                       [212(SP)]
      63:   iadd          SP                    #192                SP
      64:   goto                                                    7
      65:   isub          SP                    #192                SP
      66:   rcpy          [204(SP)]                                 t24
      67:   rstr          t24                                       [108(SP)]
      68:   int2flt       #1                                        t25
      69:   fsub          [12(SP)]              t25                 t26
      70:   fstr          t26                                       [116(SP)]
      71:   cmul          [20(SP)]              [20(SP)]            t27
      72:   cstr          t27                                       [124(SP)]
      73:   iset          #76                                       [156(SP)]
      74:   iadd          SP                    #104                SP
      75:   goto                                                    29
      76:   isub          SP                    #104                SP
      77:   ccpy          [140(SP)]                                 t28
      78:   rtl2cpx       t22                                       t29
      79:   csub          t29                   t28                 t30
      80:   cpx2rtl       t30                                       t31
      81:   rstr          t31                                       [88(SP)]
      82:   iset          #85                                       [100(SP)]
      83:   iadd          SP                    #88                 SP
      84:   goto                                                    1
      85:   isub          SP                    #88                 SP
      86:   icpy          [96(SP)]                                  t32
      87:   int2flt       t32                                       t33
      88:   fdiv          [12(SP)]              t33                 t34
      89:   flt2cpx       t34                                       t35
      90:   cstr          t35                                       [72(SP)]

      91:   cadd          [56(SP)]              [72(SP)]            t36
      92:   cstr          t36                                       [36(SP)]
      93:   goto                                                    *52(SP)

void init ( )
{
    i = 123;
    j = 234;
    k = 345;
r = 98 // 87;
s = 87 // 76;
t = r * s;
a = 4.567;
b = 5.678;
c = b - a;
x = 1;
y = 0@1;
z = 6.78@-7.89;
/* jump to the caller */
}
init();
/* no return value to copy */
z = f3(i+j+k, f2(21e1, f1(r)), b, y + 2.1 @ -1.2);
/* i + j */
/* (i + j) + k */
/* save first parameter of f3 */
/* 21e1 */
/* save as first parameter of f2 */
```

```
Code for function init()

      94:   istr          #123                                      [1000024]

      95:   istr          #234                                      [1000028]

      96:   istr          #345                                      [1000032]

      97:   rstr          #<98/87>                                  [1000036]

      98:   rstr          #<87/76>                                  [1000044]

      99:   rmul          [1000036]             [1000044]           t37
     100:   rstr          t37                                       [1000052]

     101:   fstr          #4.567000                                 [1000000]

     102:   fstr          #5.678000                                 [1000008]

     103:   fsub          [1000008]             [1000000]           t38
     104:   fstr          t38                                       [1000016]

     105:   int2cpx       #1                                        t39
     106:   cstr          t39                                       [1000060]

     107:   cstr          #<0.000000+i1.000000>                     [1000076]

     108:   cstr          #<6.780000-i7.890000>                     [1000092]

     109:   goto                                                    *0(SP)

Code for global statements
     110:   iset          #113                                      [108(SP)]
     111:   iadd          SP                    #108                SP
     112:   goto                                                    94
     113:   isub          SP                    #108                SP

     114:   iadd          [1000024]             [1000028]           t40
     115:   iadd          t40                   [1000032]           t41
     116:   istr          t41                                       [108(SP)]
     117:   flt2int       #210.000000                               t42
     118:   istr          t42                                       [196(SP)]
```

```
                          /* save as first parameter pf f1 */
                        /* store the return address for f1 */
                                              /* call f1(r) */


                  /* store the return value to a temporary */
            /* typecasting for the second parameter for f2 */
                      /* save the second parameter for f2 */
                                /* return address for f2 */
                                  /* call f2(21e1, f1(r))*/


                      /* copy return value to a temporary */
   /* save the second parameter (the return value) for f3 */
                   /* save the third parameter (b) for f3 */
            /* add y and the complex constant  2.1 @ -1.2 */
                  /* save the fourth parameter (b) for f3 */
                                /* return address for f3 */
    /* call f3(i+j+k, f2(21e1, f1(r)), b, y + 2.1 @ -1.2) */


                  /* copy the return value to a temporary */
                              /* store the temporary in z */

                                          /* return to OS */
```

```
119:   rstr      [1000036]                                  [240(SP)]
120:   iset      #123                                       [252(SP)]
121:   iadd      SP                  #240                   SP
122:   goto                                                 1
123:   isub      SP                  #240                   SP
124:   icpy      [248(SP)]                                  t43
125:   int2flt   t43                                        t44
126:   fstr      t44                                        [200(SP)]
127:   iset      #130                                       [216(SP)]
128:   iadd      SP                  #196                   SP
129:   goto                                                 7
130:   isub      SP                  #196                   SP
131:   rcpy      [208(SP)]                                  t45
132:   rstr      t45                                        [112(SP)]
133:   fstr      [1000008]                                  [120(SP)]
134:   cadd      [1000076]           #<2.100000-i1.200000>  t46
135:   cstr      t46                                        [128(SP)]
136:   iset      #139                                       [160(SP)]
137:   iadd      SP                  #108                   SP
138:   goto                                                 29
139:   isub      SP                  #108                   SP
140:   ccpy      [144(SP)]                                  t47
141:   cstr      t47                                        [1000092]

142:   halt
```