COMPILER DESIGN

Code generation

Variable declarations and value access/assignment

- Integer variable declaration:

```
int x;
```

```
x      res 4
```

where x is the address of x, which is a (unique) label generated during the parse and stored in the symbol table.

- To load or change the content of an integer variable:

```
lw r1,x(r0)
sw x(r0),r1
```

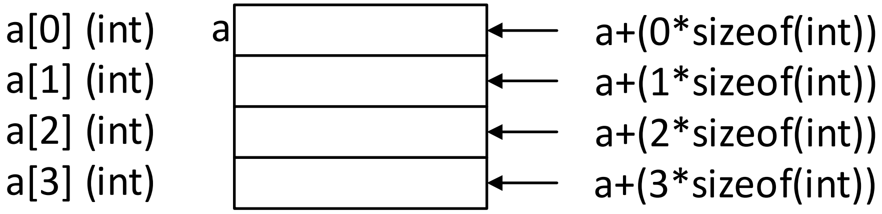
where x is the label of variable `x`, `r1` is the register containing the value of variable x and `r0` is assumed to contain 0 (offset).

Variable declarations and access

- Array of integers variable declaration:

```
int a[4];
```

```
a     res 16
```



a[0] (int)
a[1] (int)
a[2] (int)
a[3] (int)

a

a+(0*sizeof(int))
a+(1*sizeof(int))
a+(2*sizeof(int))
a+(3*sizeof(int))

- Accessing elements of an array of integers, using offsets:

```
x = a[2];
```

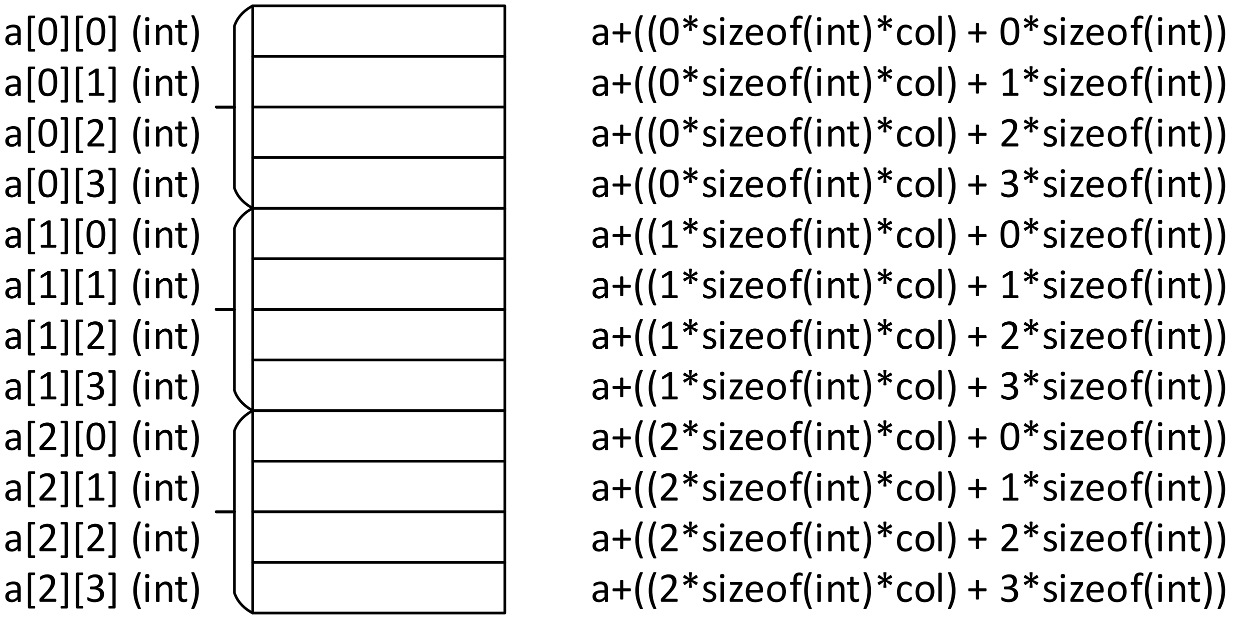
```
addi r1,r0,8
lw r2,a(r1)
sw x(r0),r2
```

Variable declarations and access

- Multidimensional arrays of integers:

```
int a[3][4];
```

```
a       res 48
```

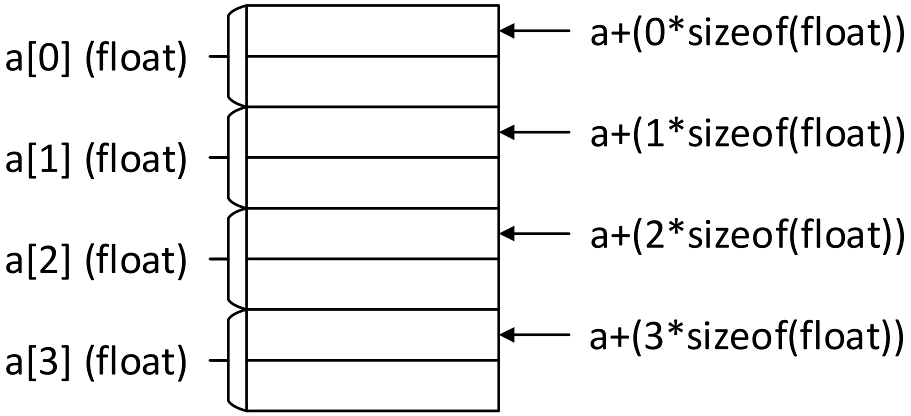


a[0][0] (int)	a+((0*sizeof(int)*col) + 0*sizeof(int))
a[0][1] (int)	a+((0*sizeof(int)*col) + 1*sizeof(int))
a[0][2] (int)	a+((0*sizeof(int)*col) + 2*sizeof(int))
a[0][3] (int)	a+((0*sizeof(int)*col) + 3*sizeof(int))
a[1][0] (int)	a+((1*sizeof(int)*col) + 0*sizeof(int))
a[1][1] (int)	a+((1*sizeof(int)*col) + 1*sizeof(int))
a[1][2] (int)	a+((1*sizeof(int)*col) + 2*sizeof(int))
a[1][3] (int)	a+((1*sizeof(int)*col) + 3*sizeof(int))
a[2][0] (int)	a+((2*sizeof(int)*col) + 0*sizeof(int))
a[2][1] (int)	a+((2*sizeof(int)*col) + 1*sizeof(int))
a[2][2] (int)	a+((2*sizeof(int)*col) + 2*sizeof(int))
a[2][3] (int)	a+((2*sizeof(int)*col) + 3*sizeof(int))

- To access specific elements, a more elaborated offset calculation needs to be implemented, and the offset value be put in a register before accessing.

Variable declarations and access

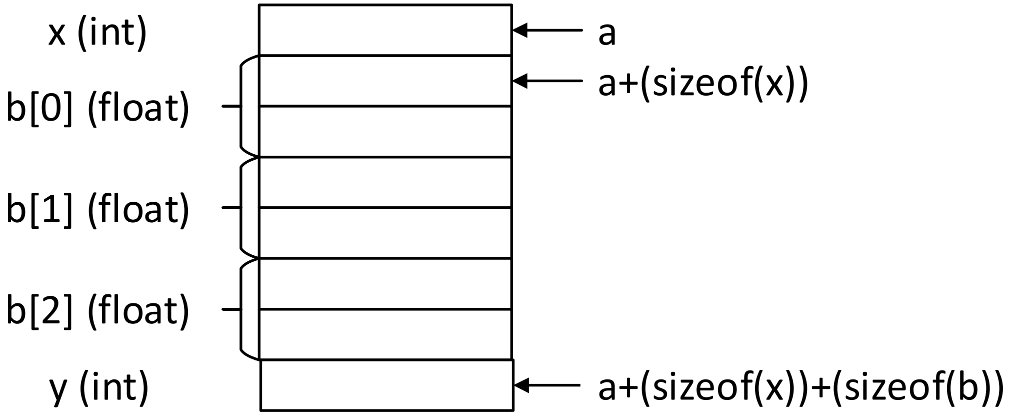
- For arrays of elements of aggregate type, each element takes more than one memory cell.
- The offset calculation needs to take into account to size of each element.
- For example, assuming a float takes 8 bytes (2 words):



Variable declarations and access

- For an object variable declaration, each data member is stored contiguously in the order in which it is declared.

```
class MyClass{
  int x;
  float b[3]
  int y;
}
```



```
Myclass a;
```

```
a         res 32
```

- The offsets are calculated according to the total size of the data members preceding the member to access.

```
x = a.b[2]…
a + (offset of x) + (offset of b[2])
a + (sizeof(x))   + sizeof(float)*2)
```

```
addi r1,r0,4
addi r1,r1,16
lw r2,a(r1)
sw x(r0),r2
```

Arithmetic operations

a+b

```
     lw r1,a(r0)
     lw r2,b(r0)
     add r3,r1,r2
t1   res 4
     sw t1(r0),r3
```

a+8

```
     lw r1,a(r0)
     addi r2,r1,8
t2   res 4
     sw t2(r0),r2
```

a*b

```
     lw r1,a(r0)
     lw r2,b(r0)
     mul r3,r1,r2
t3   res 4
     sw t3(r0),r3
```

a*8

```
     lw r1,a(r0)
     muli r2,r1,8
t4   res 4
     sw t4(r0),r2
```

Relational operators

a==b

```
     lw r1,a(r0)
     lw r2,b(r0)
     ceq r3,r1,r2
t5   res 4
     sw t5(r0),r3
```

a==8

```
     lw r1,a(r0)
     ceqi r2,r1,8
t6   res 4
     sw t6(r0),r2
```

Logical operators

- The Moon machine's **and**, **or** and **not** operators are bitwise operators.
- In order to have a logical operators, we need to code them with the assumption that false is 0 and anything else is true.

a and b

```
            lw r1,a(r0)
            lw r2,b(r0)
t7          res 4
            bz r1, zero1
            bz r2, zero1
            addi r1,r0,1
            j endand1
zero1       addi r3,r0,0
endand1     sw t7(r0),r0
```

not a

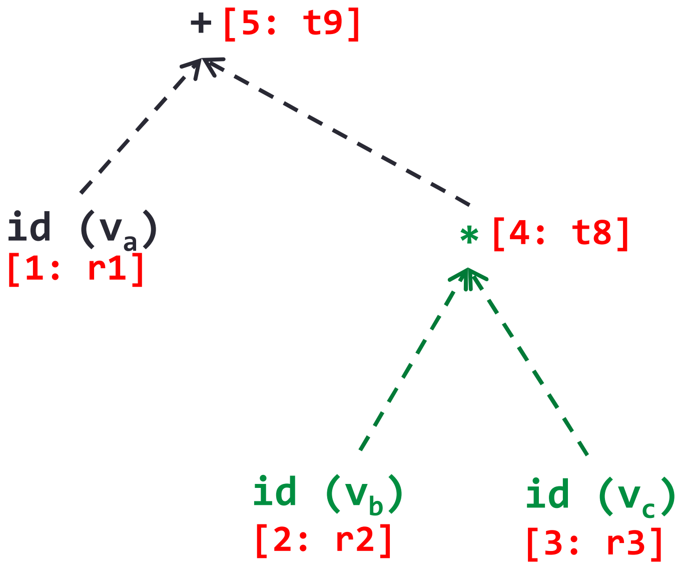
```
            lw r1,a(r0)
            not r2,r1
t8          res 4
            bz r2,zero2
            addi r1,r0,1
            sw t8(r0),r1
            j endnot1
zero2       sw t8(r0),r0
endnot1
```

Expressions

- Each operator’s code generation in the previous examples is the result of translating a subtree with two leaves as the operands and one intermediate node as the operator.
- For composite expressions, the temporary results become operands of operators higher in the tree.

a+b*c

```
          lw r1,a(r0)    [1]
          lw r2,b(r0)    [2]
          lw r3,c(r0)    [3]
          mul r4,r2,r3   [4]
t8        res 4          [4]
          sw t8(r0),r4   [4]
          lw r5,t8(r0)   [5]
          add r6,r1,r5   [5]
t9        res 4          [5]
          sw t9(r0),r6   [5]
```



Assignment operation

`a := b;`

```
lw r1,b(r0)
sw a(r0),r1
```

`a := 8;`

```
sub r1,r1,r1
addi r1,r1,8
sw a(r0),r1
```

`a := b+c;`

```
{code for b+c. yields tn as a result}
lw r1,tn(r0)
sw a(r0),r1
```

Conditional statements

```
if a>b then a:=b; else a:=0;
   [1] [2]  [3]  [4]  [5] [6]
```

```
             {code for "a>b“, yields tn as a result}[1]
             lw r1,tn(r0)                          [2]
             bz r1,else1                           [2]
             {code for "a:=b”}                     [3]
             j endif1                              [4]
else1 [4]    {code for "a:=0”}                     [5]
endif1[6]    {code continuation}
```

Loop statements

```
while a<b  do  a:=a+1;
[1]   [2]  [3] [4]   [5]
```

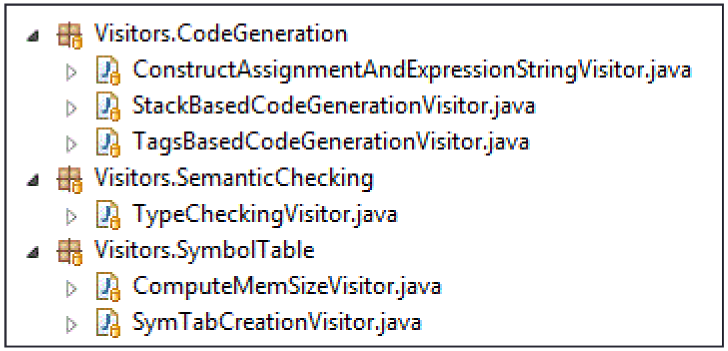
```
gowhile1 [1] {code for "a<b". yields tn as a result}[2]
             lw r1,tn(r0)                           [3]
             bz r1,endwhile1                        [3]
             {code for statblock (a:=a+1)}          [4]
             j gowhile1                             [5]
endwhile1[5] {code continuation}
```

Translating functions

- There are two essential parts in translating programs that use functions:
  - translating function definitions.
  - translating function calls.
- First, the compiler encounters a function header. It can either be a function prototype (if the language has them) or the header of a full function definition.
- In both cases, a record can be created in the appropriate symbol table, and a local symbol table can be created if it is a new function.
- In the case of a full definition, the code is generated for the variable declarations and statements inside the body of the function, which is preceded by parameter-passing instructions, and followed by return value passing instructions.

Translating functions

- The address field in the symbol table entry of the function contains the address (or label) of the first memory cell assigned to the function.
- This address/label will be used to jump to the function when a function call is encountered and translated.
- Function calls raise the need for semantic checking. The number and type of actual parameters must match with the information stored in the symbol table for that function.
- Once the semantic check is successful, semantic translation can occur for the function call.
- For modularity purposes, it is better to have all semantic checks in a separate phase that runs prior to the code generation.



- Visitors.CodeGeneration
  - ConstructAssignmentAndExpressionStringVisitor.java
  - StackBasedCodeGenerationVisitor.java
  - TagsBasedCodeGenerationVisitor.java
- Visitors.SemanticChecking
  - TypeCheckingVisitor.java
- Visitors.SymbolTable
  - ComputeMemSizeVisitor.java
  - SymTabCreationVisitor.java

Function declarations

```
int fn ( int a, int b ){ statlist };
     [1]    [2]      [3]    [4]    [5]
```

```
fnres  res 4                               [1]
fnp1   res 4                               [2]
fn     sw fnp1(r0),r2                      [2]
fnp2   res 4                               [3]
       sw fnp2(r0),r3                      [3]
       {code for var. decl. & statement list} [4]
       {assuming tn contains return value}    [4]
       lw r1,tn(r0)
       sw fnres(r0),r1
       jr r15                              [5]
```

- The above code assumes that the parameters are passed using registers, and that they are eventually stored in memory cells identified with a tag name.
  - Dependent on number of registers available.
  - Can only pass a value that fits into a register (or pass an address).
  - This is a simple solution, but with severe limitations.

Function declarations

- `fn` corresponds to the first instruction in the function.
- `fnres` contains the return value of `fn`.
- Parameters are copied to registers at function call and copied in the local variables when the function execution begins.
- This limits the possible number of parameters to the number of registers available.
- `r15` is reserved for linking back to the instruction following the jump at function call (see the following slides for function calls).

Function calls

- For languages not allowing recursive function calls, only one occurrence of any function can be running at any given time if we are using this model.
- In this case, all variables local to the function are statically allocated at compile time. The only things there are to manage are:
    - the jump to the function code.
    - the passing of parameters upon calling and return value upon completion.
    - the jump back to the instruction following the function call.

Function calls: simples case: no parameters

`fn()`

```
        ...
        {code for calling function}
        jl r15,fn
        {code continuation in the calling function}
        ...
fn      {code for called function}
        ...
        jr r15
        ...
```

Function calls: passing parameters

- Parameters may be passed using registers.
- In this case, the number of parameters passed cannot exceed the total number of registers.
- The return value can also be passed using a register, typically `r1`.
- Simplistic parameter passing method. Works only in restricted cases.

Function calls: passing parameters (registers)

```
x = fn(p1,p2);
```

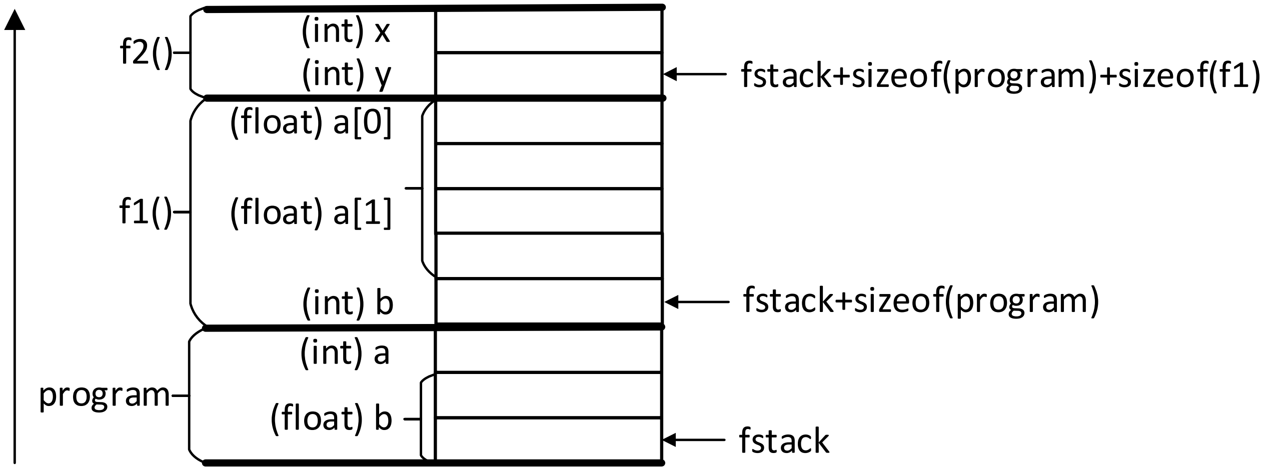
```
       ...
       {code for the calling function}
       lw r2,p1(r0)
       lw r3,p2(r0)
       jl r15,fn
       {assignment: assumes r1 contains return value}
       sw x(r0),r1
       {code continuation in the calling function}
       ...
fn     {refer to param[i] as ri+1 in fn code}
       sw fnp2(r0),r3
       sw fnp1(r0),r2
       ...
       {assuming tn contains return value}
       lw r1,tn(r0)
       jr r15
```

Function calls: passing parameters: multiple function call instances

- To avoid the limitation of the allowed number of parameters to the number of registers, parameters can be stored statically in a tagged memory cell (one for each parameter).
- These methods are only usable for languages where recursive function calls are not allowed.
- With recursive function calls, the problem is that several instances of the same function can be running at the same time, hence there is a need to store the state of each function invocation of the same function.
- To enable more than one function instance to run at the same time, all the variables and parameters of a running function are stored in a stack frame which is dynamically allocated on the function call stack.
- This involves the elaboration of a primitive run-time system as part of the compiled code.
- Another problem with multiple function instances is that `r15` is used to store the return address after a call. If there is more than one consecutive call (i.e. `prog` calls `f1`, then `f1` calls `f2`), then the return address needs to be stored in the function call stack frame.

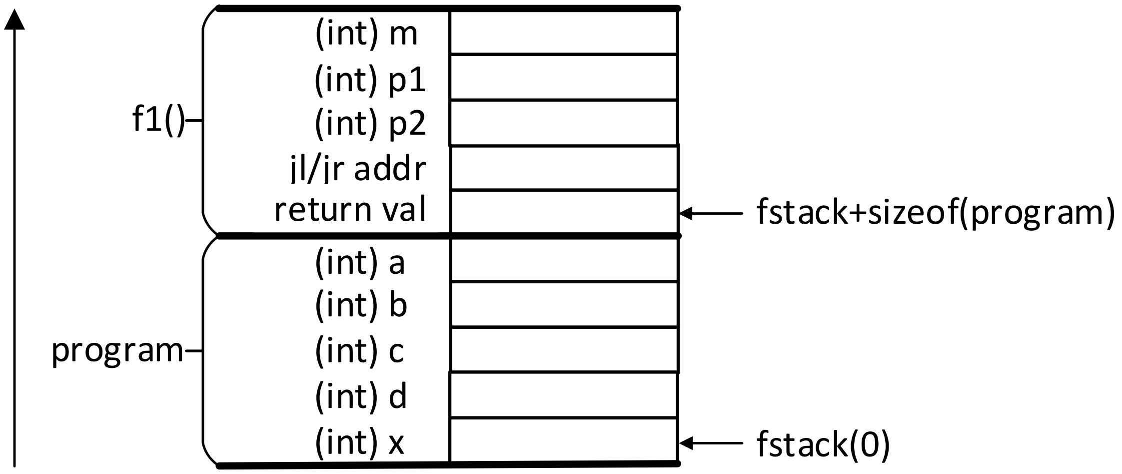
Function calls: function call stack and stack frames

- If multiple call instances is allowed, a **function call stack** is required:
  - The function call stack is a fixed-size memory area statically reserved.
  - For each function call, a **stack frame** is created on the function call stack.
  - The stack frame contains the values of all the local variables declared in the function.
  - The size of a stack frame is the sum of the sizes of all the function’s local variables.
  - The location of the top of the stack is managed by adding/subtracting stack frame sizes as an accumulated offset from the initial address of the stack.
  - Then, when the functions’ code uses its local variables, it refers to them as stored on the current function’s stack frame.
  - When the function returns, its stack frame is “removed”, i.e. the function call stack offset is decremented by its function call stack frame size.



Function calls: function call stack

- Function stack frames also need to contain space necessary to store values used in the function call mechanism, i.e. not only the local variables, but also:
  - The address stored in r15 by jl as the function is called.
  - The return value in a place predictable by the calling function. From the perspective of the calling function, the return value is always stored at:
    - `fstack + sizeof(myblock) + sizeof(typeof(return value))`
  - The parameters in a place predictable by the calling function.
    - e.g. for f1's parameters:
    - `fstack + sizeof(myblock) + sizeof(typeof(return value))`
      `+ 4 + sizeof(typeof(parameter1)) + sizeof(typeof(parameter1))`
    - `fstack + sizeof(myblock) + sizeof(typeof(return value))`
      `+ 4 + sizeof(typeof(parameter1))`
      `+ sizeof(typeof(parameter1))`



```
int f1(int p1, int p2){
  int m1;
  m1 = 5;
  p1 = p1 * m1;
  p2 = p2 * m1;
  return(p1 + p2);
}
program{
  int a;
  int b;
  int c;
  int d;
  int x;
  a = 1;
  b = 2;
  c = 3;
  d = 4;
  x = a + f1(b,c) * d;
  put(x);
} // result = 101
```

Function calls: function call stack: compute variables/block sizes and offsets

- For code generation the most important thing is to proceed in stages. Do not try to resolve all code generation in a single batch.
  - The first step is to compute the size of all variables involved in the compiled program.
  - These can be stored in the symbol tables.
  - Memory also needs to be reserved for intermediate results, and literal values used in the compiled program.
  - Then you can compute the offset of each element in a reserved block.

```
program{
  int a;
  int b;
  int c;
  a = 1;
  put(a);
  b = 2;
  put(b);
  c = 3;
  put(c);
  a = a + b c;
  put(a + 6);
} // result = 13
```

```
======================================================
| table: global           scope size: 0            |
======================================================
| func      | program   | void                     |
|    ======================================================
|    | table: program         scope size: 40        |
|    ======================================================
|    | var       | a         | int       | 4     | 0      |
|    | var       | b         | int       | 4     | 4      |
|    | var       | c         | int       | 4     | 8      |
|    | litval    | t1        | int       | 4     | 12     |
|    | litval    | t2        | int       | 4     | 16     |
|    | litval    | t3        | int       | 4     | 20     |
|    | tempvar   | t4        | int       | 4     | 24     |
|    | tempvar   | t5        | int       | 4     | 28     |
|    | litval    | t6        | int       | 4     | 32     |
|    | tempvar   | t7        | int       | 4     | 36     |
|    ======================================================
======================================================
```

Function calls: function call stack: compute variables/block sizes and offsets

```
class class1{
   float float1;
   int int1;
}

int func1(int int235[2][3][5], float float4[10]){
   float float7;
   a=a+b*3;
 }

 program{
   int int532[5][3][2];
   class1 class110[10];
   float float3;
   int int3;
   a=a+b*c;
   x=a+b*c;
   a=x+z*y
  }
```

```
=====================================================
| class     | class1                                |
|    =====================================================
|    | table: class1          scope size: 12            |
|    =====================================================
|    | var       | float1    | float     | 8     | 0     |
|    | var       | int1      | int       | 4     | 8     |
|    =====================================================
| | func      | int       | func1                     |
|    =====================================================
|    | table: func1           scope size: 216           |
|    =====================================================
|    | param     | int235    | int       | 120   | 0     |
|    | param     | float4    | float     | 80    | 120   |
|    | var       | float7    | float     | 8     | 200   |
|    | tempvar   | t1        | int       | 4     | 208   |
|    | tempvar   | t2        | int       | 4     | 212   |
|    | litval    | t3        | int       | 4     | 216   |
|    =====================================================
| | func      | program   | void                      |
|    =====================================================
|    | table: program         scope size: 856           |
|    =====================================================
|    | var       | int532    | int       | 120   | 0     |
|    | var       | float101  | class1    | 120   | 120   |
|    | var       | float3    | float     | 8     | 240   |
|    | var       | int3      | int       | 4     | 248   |
|    | tempvar   | t7        | int       | 4     | 252   |
|    | litval    | t8        | int       | 4     | 256   |
|    | tempvar   | t9        | int       | 4     | 260   |
|    | tempvar   | t10       | int       | 4     | 264   |
|    | tempvar   | t11       | int       | 4     | 268   |
|    | tempvar   | t12       | int       | 4     | 272   |
|    | tempvar   | t13       | int       | 4     | 276   |
|    =====================================================
=====================================================
```

get and put: calling the operating system

- Some function calls interact with the operating system, e.g. when a program does input/output
- In these cases, there are several possibilities depending on the resources offered by the operating system, e.g.:
  - treatment via special predefined ASM operations/subroutines
  - access to the OS via calls or traps
- In the Moon processor, we have two special operators: **`putc`** and **`getc`**
- They respectively output some data to the screen and input data from the keyboard
- They are used to directly translate **`get()`** and **`put()`** statements (see the Moon manual)
- There are also a variety of libraries provided with the Moon code:
  - `lib.m`: read/write strings to console/from keyboard, string/integer conversion, string operations
  - **`util.m`**: read/write integer to console/from keyboard, string operations.

Code generation: suggested sequence

- Suggested sequence:
  - variable declarations (integers first)
  - expressions (one operator at a time)
  - assignment statement
  - put and get statements
  - conditional statement
  - loop statement
- Tricky parts:
  - function calls
  - expressions involving arrays (offset calculation)
  - floating point numbers
  - recursive function calls
  - expressions involving access to object members (offset calculations)
  - calls to member functions (access to object’s data members)

Hints for final stages leading to the project demonstration

- You will not fail the project if you did not implement code generation for all aspects of the language.
- But, you might fail if your compiler is not working at all.
- This is why you should proceed in stages and make sure each successive stage is correct before going further.
- Be careful to not break what was previously working.
- This is the main reason why you should have numerous tests in place, ideally organized in automated regression testing. Unit testing is a good way to achieve that.
- Make sure you have a compiler that works properly for a subset of the problem.
- For the parts that you did not implement, think of a solution. You may get some marks if you are able to clearly explain how to do what is missing during your project demonstration.