COMP 442 /6421 Compiler Design

Code Generation II

Instructor:	Dr. Joey Paquet	paquet@cse.concordia.ca
TAs:	Vashisht Marhwal	vmarhwal97@gmail.com
	Hamed Jafarpour	hamed.jafarpour@concordia.ca

Content

- Commands for running moon code
- Important notes for Assignment 5
- Code generation for function definitions and function call stack

Commands for running moon code

- require c compiler. on linux system following commands can be runnable
- gcc moon.c -o moon
- ./moon sample.m lib.m

Example

source code →assembly code

```
program {
    int x;
    int y;
    int z;
    x = 2;
    y = 34;
    z = x + y * x;
    put (z);
};
```

```
entry   % ======program entry======
align   % following instruction align
addi    R1, R0, topaddr % initialize the stack pointer
addi    R2, R0, topaddr % initialize the frame pointer
subi    R1, R1, 12  % set the stack pointer to the top position of the stack
addi    R14, R0, 2  %
sw  -12(R2), R14    %
addi    R8, R0, 34  %
sw  -8(R2), R8  %
lw  R6, -12(R2) %
lw  R9, -8(R2)  %
lw  R11, -12(R2)    %
mul R9, R9, R11 %
add R6, R6, R9  %
sw  -4(R2), R6  %
lw  R10, -4(R2) %
putc    R10 %
hlt % ======end of program======
```

```
ERIC_LAI  ~/Downloads/moon  ./moon ../OnlyProgram.m
Loading ../OnlyProgram.m.
F
221 cycles.
```

2+2*34=70 ⟶ ascii code F

```
program {
    int x;
    x = 65;
    if (x == 1) then {
        x = 65;
    } else {
        x = 66;
    };
    put (x);
};
```

```
        entry   % ======program entry======
        align   % following instruction align
        addi    R1, R0, topaddr % initialize the stack pointer
        addi    R2, R0, topaddr % initialize the frame pointer
        subi    R1, R1, 4   % set the stack pointer to the top position of the stack
        addi    R14, R0, 65 %
        sw  -4(R2), R14 %
        lw  R8, -4(R2)  %
        ceqi    R8, R8, 1   %
        bz  R8, else_1  % if statement
        addi    R6, R0, 65  %
        sw  -4(R2), R6  %
        j   endif_1 % jump out of the else block
    else_1  addi    R9, R0, 66  %
        sw  -4(R2), R9  %
    endif_1 nop % end of the if statement
        lw  R11, -4(R2) %
        putc    R11 %
        hlt % ======end of program======
```

```
ERIC_LAI  ~/Downloads/moon  ./moon ../IfStatement.m
Loading ../IfStatement.m.
B
162 cycles.
```

How to do Assignment 5 ?

Assignment 5 is fairly involved

- You will likely not have time to implement every feature of the language.
- Familiarize yourself with the Moon processing environment
- Implement simple statements for compiler code generation
- Read/Write critical for testing
- Simple arithmetic requires few memory considerations
- Pick a static memory scheme
- Use Tags or stack-based approach
- Prioritize the implementation of language features
    - By difficulty
    - By utility
    - By grade weight

Important notes for Tag based approach

- Tags in moon code are necessary for jumping between functions and conditional structures
- They are straightforward to use, but make sure generated tags are always unique
- Prefixes can help with this
- Be careful of tricky edge cases:
- Function overloading
- Function overriding and inheritance
- Similar free functions and member functions
- If using tag for memory, uniqueness is much harder

Example
- class_function_functionName_param1Type_param2Type
- if_22, then_22, else_227

Code generation: suggested sequence

Suggested sequence:

- variable declarations (integers first)
- expressions (one operator at a time)
- assignment statement
- read and write statements
- conditional statement
- loop statement

Tricky parts:

- function calls
- expressions involving arrays and classes (offset calculation)
- floating point numbers (non-native in Moon)
- function call stack
- expressions involving access to object members (offset calculations)
- calls to member functions (access to object's data members)

Code generation: Function definitions

Solution:

- Branching to the function's code
- Passing/storing the parameter values
- Storing/passing the return value

Code generation: Function definitions

Example:

```
int fn (int a,int b)
{
  statement1;
  return(expr);
};
```

```
fnres   res 4
fnp1    res 4
fnp2    res 4
fn [4]  {code for statement1}
        {code for expr yields tn}
        lw r1,tn(r0)
        sw fnres(r0),r1
        jr r15
```

Code generation: Function definitions

- previous example uses static memory allocation for a function, which is assuming that there can be at most one instance of a function being executed at any time.
- To allow more than one instance of a function to execute at the same time, a dynamic memory allocation scheme is necessary, i.e. a function call stack

multiple function call instances

- 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 a separate state of each function instances 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 a function call stack.
- Another problem with multiple function instances is that r15 is used to store the return address that is going to be branched upon after a call. If there is more than one consecutive call (i.e. main calls f1, then f1 calls f2), then the return address needs to be stored in the function call's stack frame.

function call stack and stack frames

- The location of the stack frame on top of the stack (topofstack) is managed by adding/subtracting stack frame sizes as an accumulated offset from the base address of the stack (fstack).

- Before a function is called, topofstack is incremented by the stack frame size of the function currently being executed.
- Then, when the functions' code uses its local variables, it refers to them using offsets relative to topofstack.

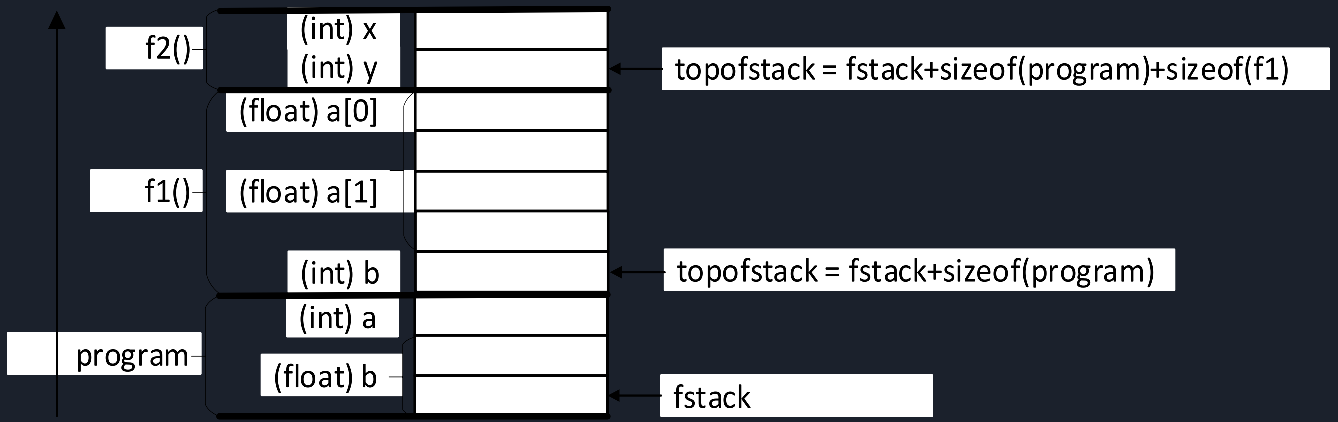
- After the function returns, the calling function "removes" the called function's stack frame, i.e. topofstack is decremented by its function call stack frame size.

function call stack and stack frames

```
program{
  int a;
  float b;
  … f1(…) … }

int f1(…){
  float a[2];
  int b;
  … f2(…) …}

int f2(…){
  int x,y; }
```



function call stack: compute variables/block sizes and offsets

- 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

function call stack: compute variables/block sizes and offsets

Example

```
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   |
|     ====================================================
=====================================================
```

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

Function calls using stack: full example

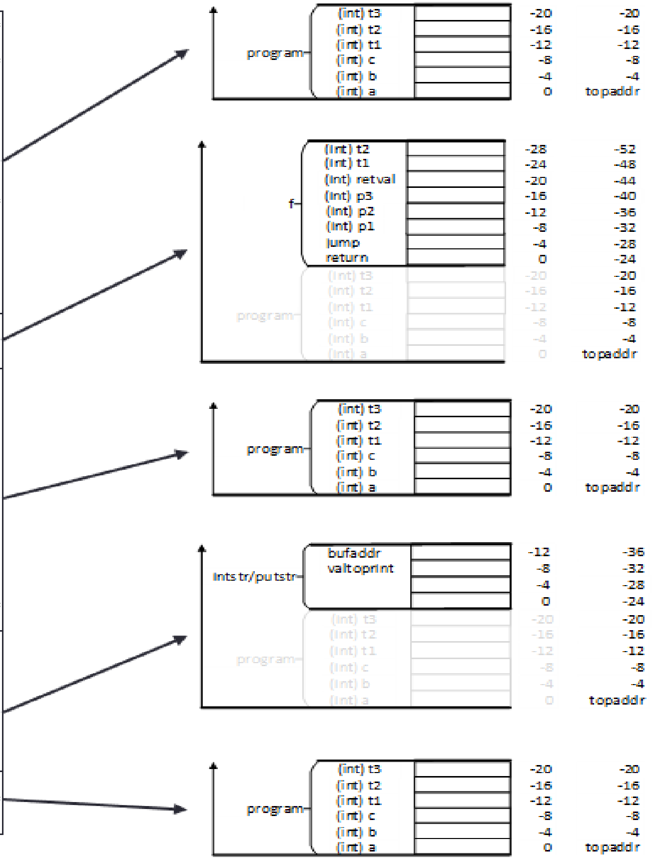
```
int f(int p1, int p2, int p3){
  int retval;
  retval=p1+p2*p3;
  put (retval);
  return(retval);
}
program{
  int a;
  int b;
  int c;
  a=1;
  b=2;
  c=3;
  a=f(a,b,c)*b+c;
  put(a);
} // output: 717
```

```
=====================================================
| table: global            scope size: 32           |
=====================================================
| func        | f          | int                      |
|    =====================================================
|    | table: f                  scope size: 28         |
|    =====================================================
|    | var        | p1         | int        | 4      | -8   |
|    | var        | p2         | int        | 4      | -12  |
|    | var        | p3         | int        | 4      | -16  |
|    | var        | retval     | int        | 4      | -20  |
|    | tempvar    | t1         | int        | 4      | -24  |
|    | tempvar    | t2         | int        | 4      | -28  |
|    =====================================================
| func        | program    | void                     |
|    =====================================================
|    | table: program            scope size: 24         |
|    =====================================================
|    | var        | a          | int        | 4      | -0   |
|    | var        | b          | int        | 4      | -4   |
|    | var        | c          | int        | 4      | -8   |
|    | tempvar    | t1         | int        | 4      | -12  |
|    | tempvar    | t2         | int        | 4      | -16  |
|    | tempvar    | t3         | int        | 4      | -20  |
|    =====================================================
=====================================================
```

Function calls using stack: full example

```
program{
   int a;
   int b;
   int c;
   a=1;
   b=2;
   c=3;
   a=f(a,b,c)*b+c;
   put(a);
}
```

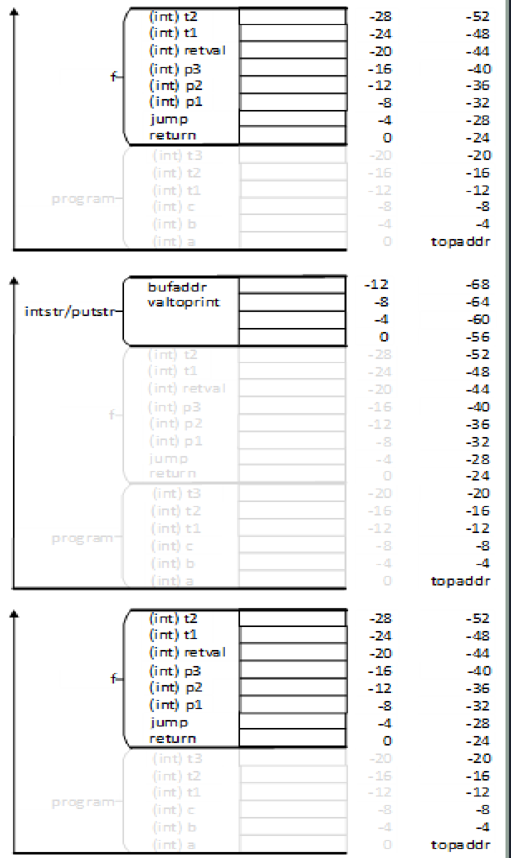
```
      % begin program function definition =================
      entry                   % start program
      % a=1;b=2;c=3 =======================================
      addi r14,r0,topaddr
      addi r1,r0,1            % a=1
      sw -0(r14),r1           % a=1
      addi r1,r0,2            % b=2
      sw -4(r14),r1           % b=2
      addi r1,r0,3            % c=3
      sw -8(r14),r1           % c=3
      % function call to f =================================
      lw r1,0(r14)            % pass a into p1
      sw -32(r14),r1          % pass a into p1
      lw r1,-4(r14)           % pass b into p2
      sw -36(r14),r1          % pass b into p2
      lw r1,-8(r14)           % pass c into p3
      sw -40(r14),r1          % pass c into p3
      addi r14,r14,-24        % increment stack frame
      jl r15,f                % jump to f
      subi r14,r14,-24        % decrement stack frame
      % a=f(a,b,c)*b+c ====================================
      lw r1,-24(r14)          % get return value from f
      sw -12(r14),r1          % put it to t1
      lw r1,-12(r14)          % t2=t1*b
      lw r2,-4(r14)           % t2=t1*b
      mul r3,r1,r2            % t2=t1*b
      sw -16(r14),r3          % t2=t1*b
      lw r1,-16(r14)          % t3=t2+c
      lw r2,-8(r14)           % t3=t2+c
      add r3,r1,r2            % t3=t2+c
      sw -20(r14),r3          % t3=t2+c
      lw r1,-20(r14)          % a=t3
      sw 0(r14),r3            % a=t3
      % put(a) ============================================
      lw r1,-0(r14)           % get a from stack grame
      addi r14,r14,-24        % increment stack frame
      sw -8(r14),r1           % put a onto stack
      addi r1,r0, buf         % put adr of buf onto stack
      sw -12(r14),r1          % put adr of buf onto stack
      jl r15, intstr          % jump to intstr
      sw -8(r14),r13          % put string onto stack
      jl r15, putstr          % jump to putstr
      subi r14,r14,-24        % decrement stack frame
      hlt                     % stop program
      % end program function definition ===================
buf   res 20
```



Function calls using stack: full example

```
   % begin function f definition ========================================
f  sw -4(r14),r15             % put r15 on stack frame
   % retval=p1+p2*p3
   lw r1,-12(r14)             % t1=p2*p3
   lw r2,-16(r14)             % t1=p2*p3
   mul r3,r1,r2               % t1=p2*p3
   sw -24(r14),r3             % t1=p2*p3
   lw r1,-8(r14)              % t2=p1+t1
   lw r2,-24(r14)             % t2=p1+t1
   add r3,r1,r2               % t2=p1+t1
   sw -28(r14),r3             % t2=p1+t1
   lw r1,-28(r14)             % retval=t2
   sw -20(r14),r1             % retval=t2
   % put(retval) =========================================================
   lw r1,-20(r14)             % get retval from stack frame
   addi r14,r14,-32           % increment stack frame
   sw -8(r14),r3              % put value to print on stack frame
   addi r3,r0, buf            % put address of buffer on stack frame
   sw -12(r14),r3             % put address of buffer on stack frame
   jl r15, intstr             % jump to intstr subroutine
   sw -8(r14),r13             % put result on stack frame for putstr
   jl r15, putstr             % call putstr subroutine
   subi r14,r14,-32           % decrement stack frame
   % return(retval) ======================================================
   lw r1,-20(r14)             % get value of retval from stack
   sw 0(r14),r1               % put return value on stack
   lw r15,-4(r14)             % retrieve r15 from stack
   jr r15                     % jump back to calling fct
   % end function f definition ==========================================
```

```
int f(int p1, int p2, int p3){
  int retval;
  retval=p1+p2*p3;
  put (retval);
  return(retval);
}
```



Thanks!