Final code generation

- Write a set of auxiliary procedures for generating labels and temporaries.
- Find types of variables. (Hopefully you found this in earlier part).
- Find all temporaries that you might need (GenCode for each expression can find all temporaries).
- Make a map of how stack and heap is going to look like. That is, how are specific variables going to get mapped in the memory.
- Associate an offset for all variables and parameters.
- Establish a protocol for calling a function and returning from a function.
- For every expression type, find a correspondence between the expression and its corresponding implementation in assembly. (Check GenCode related handout)
- For every statement type, find a corresponding implementation (Check GenCode related handout)
Class

- Find size of class
- Determine all fields: include its fields and field of class classes.
- Assign an offset for members of fields (including those that you receive from superclass). You will need type for every class:
- Arrange for code generation for every method:

```cpp
ClassType::CodeGen()
{  // find the fields of this class
    Fields *fl = Fields();
    Methods *mt = Methods();  // methods of this class
    int offset = 0;  // initial offset

    // assume that fit is an iterator
    for (fit = fl->begin(); fit!= fl->end(); fit++) {
        // assign offset
        (*fit).Offset = offset;
        class_size = class_size + (*fit).Size();
        // set offset for next field
        offset = offset + (*it).Size();
    }

    // assume that iterator mit will go over all methods of class
    for (mit = mt->begin(); mit!= mt->end(); mit++) {
        // generate code for method
        (*mit)->GenCode();
    }
}
```
Allocation of space for variables

- Every variable (including instance and local) has an offset associated with it.
- Two kinds of variables:
  1. Primitive types: Allocate space for primitive types on stack
     - Allocate 1 word for each primitive type
  2. Reference types:
     - Allocate space for reference on stack (1 word)
     - Allocate space for object on heap (equal to size of class)
- Example:
  ```java
  void func() {
    // create an object
    ClassC x = new ClassC();
    ... 
    // store a value in x.c
    x.c = 4;
    ... 
  }
  ```
  - func's activation record will allocate space for x. The space for x is, thus, allocated when func is executed.
  - Generate code that will invoke sbrk to allocate space on heap for x. Use the size of ClassC to determine amount of space (can be determined by compiler).
  - Store the address returned by sbrk on stack at an offset for the reference (say, xoffset).
Generation of code for variable access

- What kind of code should be generated to refer to a specific name?
- Remember: every variable gets mapped to a memory location: either in stack or heap.
- Local variable: Data is on stack:
  - Find offset for each variable
  - Can find variable by $fp + offset.
- Reference type:
  - Find address for data on heap
  - Find offset within heap
  - Example:
    1. Read the value of x into register r1 to get address on data associated with x on heap.
    2. Add offset of c to content of r1 to get address of x.c into register r2.
    3. Store value 4 into the address specified in register r2.

So generated code for the assignment will look something like the following:

```assembly
# load reference for x in register 16
lw $16, $fp(xoffset)
# load value 4 in register 17
lw $17, 0x04
# store $17 in address specified by 16+coffset.
sw $17, $16(coffset)
```
Method declaration

- For R p(T1 a1, T2 a2, ..) , do the following.
- Make the implicit parameter explicit: R p(Class this, T1 a1, T2 a2, ...
- Assign an offset for each parameter. The first parameter will be nearest to the framepointer.
- Layout space for local variables: lvarsize = size of local + temporary variables
- Find registers that you want to save: rsize := size of registers.
- Determine stack frame size: stack frame size := lvarsize + rsize;
- Generate an assembly routine of the form:
  .global _p_C_asm
  .ent _p_C_asm
  _p_C_asm:
- Use stack frame size to generate code that will allocate space on stack:
  Generate code of the form sp := sp - stack frame size;
- Generate code for saving registers at specific offsets with AR
- Generate code for the body
- Generate code for loading registers from the current activation record.
- Generate code for jumping to routine that called this routine.
**Method activation:** \( x.p(f_1, f_2, \ldots, f_n) \)

- Generate code for pushing value of \( f_i \) on stack. Note: you should generate code for evaluating and pushing the parameters on stack in such a way that \( f_n \) is pushed first, and \( f_1 \) last.
- Generate code for pushing value of \( x \) on stack.
- Generate code for jumping to the assembly routine generated for \( p \)
- Save any registers that you want to save

```c
// Assumption: m stores information about a method
MethodInvocationExpression *m;

// get parameters of method
ParamList p = m->GetParams();

// object on which method is invoked
Object *o = m->Object();

// Traverse parameter list in reverse order
// assume that this is how it is stored
for (pit = p.begin(); pit != p.end(); pit++) {
    (*pit).GenCode(); // generate code for each parameter
}
for (pit = p.begin(); pit != p.end(); pit++) {
    // push the symbtab that stores value for each parameter on stack
    Print('push (*pit).symtab on stack');
}
// push the object on stack
Print('push o->symtab on stack');
// construct assembly name for p
String asmFunc = AssemblyName(o->ClassName(), m->MethodName());
Print('jump to asmFunc');
```
Implementation of dynamic functions

- Define a table, say `vtbl`, for a class. The table contains pointers to virtual functions of the class. Each object of a class contains pointer to the virtual function table of the class.

```cpp
class A {
    int a;
    void f(int);
    void g(int);
    void h(char);
};
class B extends A {
    int b;
    void g(int);
};
class C extends B {
    int c;
    void h(int);
}
```

- Class C layout:

![Diagram of class C layout]

- Invoke a method `f`:
  - push arguments on stack
  - find `f`’s index in virtual function table
  - jump to the label in table.
Example: input program

class P {
    int x;
    int y;
    boolean z;
    void M(int a) {
        int b, c;
        int d;

        x = y+1;
        d = a;
    }
}
class Q {
    void N() {
        P ip;
        int jp;

        ip = new P;
        ip.M(4);
    }
}
A compact parse tree
Code Generation for program

- Code generation for example = code generation for class P and code generation for class Q.
- The following computations need to be done:
  - Class: Offsets for all instances variables + size of class. Offset will be used for accessing a specific instance variable in heap.
    For instance, in order to access instance variable x of an object X (through expression X.x) of P, we need offset of x.
  - Parameters: offsets for all parameters + total space taken by parameters.
    Offset will be used for accessing specific parameter on stack.
  - Block (local and temp) variables: Offsets for all local variables + total size.
    Total size will be used for building AR, and offset for accessing local variables within activation record.

- How will stack look for procedures?

  ![Layout for P.M](image)

  ![Layout for Q.N](image)
General Approach

- Remember: $sp points to first free location in stack. $fp: points to first local variable.
- For each method, generate a unique assembly routine.
- All local variables (primitives and references) and parameters of a program or procedure will go on stack.
  - Local variable: -offset($fp) (offset of local variable)
  - Parameter: offset($fp) (offset of parameter)
- Use a load - compute - store model.
- Global data: For every global variable, say GL, generate code like the following:
  ```
  .data
  GL: .word 0
  ```
  Initialize GL to 0, and now GL can be accessed symbolically.
- Literal: Literals like strings, float, and double values can be stored in following manner:
  ```
  .data
  astr: .asciiz "this is a string"
  temp1: .double 4.56, 0.4e-25
  temp2: .float 2.34, 0.5e-20
  ```
Find the temporaries that you need for P.M: 1.

Compute various sizes for P.M:
- Variables: 4+4+4+4: 16 bytes (var size)
- Registers: 4 bytes ($31) + 4 bytes ($fp)
- Framesize = 16 + 4 + 4 = 24

Code generation of P.M = prologue + body + epilogue.

We first look at prologue.

Generate header for P.M:
```
.text
.ent _P_M_asm
_P_M_asm:
```

Generate code that will construct an activation frame for P.M:
```
subu $sp, 24 # 24 = size of activation frame.
```

Generate code for saving registers: save $31 and $fp only.
```
sw $31, 24-16($sp) # $sp+framesize-var size
sw $fp 24-16-4($sp) # store after $31
```

Generate code for setting new frame pointer:
```
addu $fp, $sp, 24 # 24 = framesize
```

This should take care of prologue for P.M
Code generation for body of P.M

- Code generation here will involve generating body for statements of P.M.
- generate code for evaluating \( x := y + 1; \)
  - First evaluate the right hand side \( y + 1; \)
  - Store this value in a temp.
  - Generated code should first find \( y \) in memory, and change its value. So the compiler must first find where \( y \) comes from. By looking at symbol tables, it finds that \( y \) is an instance variable of P. So \( y \) is a component of object \( \text{this} \) passed to M. Generated code can find \( y \) in two steps
    - Find "this" on stack using \( fp \) (it should be the first argument):
      \[
      \text{lw} \; \$19, \; 4($fp) \; \# \; 4 = \text{offset of object}
      \]
    - Now access \( y \) from this
      \[
      \text{lw} \; \$20, \; 4($19) \; \# \; 4 = \text{offset of } y \text{ on heap}
      \]
      Register 20 contains the value of \( y \).
  - Add 1 to \( \$20 \). Then store it in temp:
    \[
    \text{addi} \; \$20, \; \$20, \; 1 \; \# \; \text{add 1}
    \]
    \[
    \text{sw} \; \$20, \; -12($fp) \; \# \; \text{store in temp (at offset 20)}
    \]
  - Generate code for assigning temp to \( x \). (Note that \( x \) will need to be accessed in the same manner as \( y \) as above):
    \[
    \text{lw} \; \$19, \; -12($fp) \; \# \; \text{load temp; offset for temp=12}
    \]
    \[
    \text{lw} \; \$20, \; 4($fp) \; \# \; 4 = \text{offset of this}
    \]
    \[
    \text{sw} \; \$19, \; 0($20) \; \# \; \text{store } \$19 \text{ into } x \text{ (at offset 0)}
    \]
- generate code for evaluating \( d = a \): Load content of \( a \) into register and store it into \( d \).
  \[
  \text{lw} \; \$19, \; 8($fp) \; \# \; 8 = \text{offset of } a
  \]
  \[
  \text{sw} \; \$19, \; -8($fp) \; \# \; \text{store in } d \text{ (at offset -8)}
  \]
Epilogue for P.M

- Generate code for restoring $31 and $fp:
  
  lw $31, 24-16($sp)  # 24-16 = offset for 31
  lw $fp 24-16-4($sp)  # 24-16-4 = offset for sp

- generate code for cleaning up stack
  
  addu $sp, 24

- generate code for returning:
  
  j $31

- Generated end:
  
  .end   _P_M_asm
Code Generation for Q.N

- Computations by compiler:
  - Variables: 4 bytes (for ip) + 4 bytes (for jp) = 8 bytes
  - Registers: 4 (register 31) + 4 bytes (register $fp)
    Framesize = 8 + 4 + 4 = 16

- Generate header for Q.N:
  .ent _Q_N_asm
  _Q_N_asm:

- Generate code for constructing activation frame for Q:
  subu $sp, 16 # 16 = size of activation frame.

- Generate code for saving registers: save $31 and $fp only.
  sw $31, 16-8($sp)
  sw $fp 16-8-4($sp)

- Generate code for setting new frame pointer:
  addu $fp, $sp, 16 # 16 = framesize
Generate body for Q.N

- Generate code for ip := new P
  1. Allocate space for an object. Size of space defined by size of P.
     This is done by calling sbrk
  2. Store the address returned by sbrk in ip.
     - li $v0, 9 # 9 = sbrk code
     - li $a0, 12 # 12 = sizeof(P)
     - syscall # system call
     - sw $v0, -0($fp) # store returned address($2) on stack
  3. The address of space allocated is stored on stack in location $fp=0. (that is, first local variable).
  4. This is the place where a constructor should be called. It will involve the following:
     - Evaluate arguments of constructor
     - Push arguments on stack
     - Call constructor
Generate body for Q.N - cont’d.

- Generate code for method invocation: ip.M(4)

1. generate code for evaluating parameters and store the evaluated values on stack.
2. Note that there are two parameters to M: i) object on which method is being invoked (ip in this case) and ii) int parameter with value 4.
3. Push first argument (ip):
   ```
   subu $sp, 4 # allocate 4 bytes for first arg
   li $19, -0($fp) # load content of ip (at offset -0)
   sw $19, 4($sp) # store value of arg into stack
   ```
4. Evaluate second argument and push it on stack:
   ```
   subu $sp, 4 # allocate 4 bytes for first arg
   li $19, 4 # load 4 in register $19
   sw $19, 4($sp) # store val of arg into stack
   ```
5. Now jump to assembly routine _P_M_asm:
   ```
   jal _P_M_asm
   ```
6. After returning from procedure, need to take back space which was allocated for arguments. So generate code for deallocating argument space (two arguments)
   ```
   addu $sp, 8 # 8 = size of arguments
   ```
Generate body for Q.N - cont’d.

Epilogue for Q.N:

1. Generate code for restoring $31 and $fp:
   
   ```
   lw $31, 16-8($sp)  # restore $31
   lw $fp 16-8-4($sp)  # restore $fp
   ```

2. Generate code for cleaning up stack:
   
   ```
   addu $sp, 16
   ```

3. Generate code for returning:
   
   ```
   j $31
   ```

4. Generated end:
   
   ```
   .end   _Q_N.asm
   ```