Week 13
Generating Code for Classes

Paul Chew
CS 212 – Spring 2004

Announcements

- Part 4
  - The due date has shifted to Monday, May 10
  - Arrays of objects are legal
  - A class's super class must be defined *before* the class
  - *Do not* use the System.exit method
  - You may need to download the latest SaM Simulator
    - Instruction PUSHIMMPA is new
  - Example program has been corrected
- Part 4 assignment page has been updated to reflect these changes and clarifications
- No Sections today (but will meet next week)
- If you are interested in finding a new partner, send me email

Example Bali4 Code

```
// Uses a Queue and a Stack
int main ( )
    {int n; Stack s; Queue q;} {
    n = 0; s = Stack(); q = Queue();
    while n < 5 do {
          s.put(n); q.put(n);
          n = n + 1;
          }
    n = 0;
    while n < 5 do {
          print s.get(); print q.get();
          n = n + 1;
          }
    return 0;
    }
```

```
class Node
    {public int data; public Node link;}
    {public Node (data, link) { }
          { this.data = data;
            this.link = link;
          }
    }{ }
```


Example Bali4 Code Continued

```
class Queue
  { private Node head, last; } { }
  { public void put (int i)
     { Node n; }
     { n = null;
       n = Node(i, n);
       if head == null then head = n;
       else last.link = n;
       last = n;
       return; }
    public int get ()
     { Node n; }
     { n = head;
       head = head.link;
       return n.data; }
}
```

```
class Stack
  { private Node top; } { }
  { public void put (int i)
     { }
     { top = Node(i, top);
       return;
     }
    public int get ()
     { Node n; }
     { n = top;
       top = top.link;
       return n.data;
     }
}
```


What Info is Needed to Generate Code?

- For a local variable
  - Offset from FBR
- For a field
  - Offset of field from start of object
- For a method
  - Offset of method from start of dispatch vector
- For a constructor
  - The size (# of fields) of the object
  - Location of the dispatch vector for the class
- All of this information is available in the AST
  - You collect this info and store in a Symbol Table
- You will need more than one pass over the AST because you cannot generate code until you know
  - Size of an object
    - Need this to create code for constructor call
  - Method's offset in dispatch vector
    - Need this to create code for method call

Outline for Bali4 Compiler

- Build the AST
- Walk the AST to find
  - Size (# fields) for each class
    - A class can inherit fields
  - Dispatch vector for each class
    - A class can inherit a dispatch vector
- Create code for each class's dispatch vector
- Walk the AST again, generating code for functions, constructors, and methods
- This is just one possible way to compile Bali4
  - You don't have to use this outline
- For our example
  - I won't take time now to build the AST
  - Node
    - Size = 2; DV = empty
  - Queue
    - Size = 2; DV = put, get
  - Stack
    - Size = 1; DV = put, get

Dispatch Vectors

- The simplest method is to build the dispatch vectors as part of the program code

```
"DV$Queue":
    JUMP "M$Queue$put$i"
    JUMP "M$Queue$get$"
"DV$Stack":
    JUMP "M$Stack$put$i"
    JUMP "M$Stack$get$"
```

- Idea is that the object itself stores the location of its dispatch vector
  - Example: a Queue object stores the address "DV$Queue"
- To jump to the start of a method
  - Push the method's offset
  - Push the address of object's DV
  - Add (offset to addressOfDV)
  - JUMPIND (jump indirect)

Saving the PC

- Previous example shows how to jump to a method's code, but we also need to save the PC (i.e., we want to use JSR)
  - In effect, we want to use JSRIND (jump to subroutine indirectly)
  - This instruction doesn't exist, but it's easy to fake
    - Place this code somewhere:

```
jsrind: SWAP
        JUMPIND
```

- The calling code for a method looks like this:

```
<Code to place space for return value on Stack>
<Code to place arguments on Stack>
<Code to save/update FBR>
<Code to place address of method on top of stack>
JSR jsrind
<Code to restore FBR>
<Code to clear arguments from Stack>
```


Initial Code

```
program:
    ADDSP 1       // rv for main
    LINK          // Create frame
    JSR "F$main$"
    POPFBR        // Restore FBR
    STOP
jsrind:
    SWAP
    JUMPIND
"DV$Queue":
    JUMP "M$Queue$put$i"
    JUMP "M$Queue$get$"
"DV$Stack":
    JUMP "M$Stack$put$i"
    JUMP "M$Stack$get$"
```

- Recall that when a class *inherits* from another class
  - It uses the same dispatch vector (with any new stuff on the end)
  - This is necessary so that an instance of the class works correctly when using methods of its super class

Code for main

```
// Uses a Queue and a Stack
int main ( )
    {int n; Stack s; Queue q;} {
    n = 0; s = Stack(); q = Queue();
    while n < 5 do {
          s.put(n); q.put(n);
          n = n + 1;
          }
    n = 0;
    while n < 5 do {
          print s.get(); print q.get();
          n = n + 1;
          }
    return 0;
    }
```

```
"F$main$":
PUSHIMM 0       // n, offset = 2
PUSHIMM 0       // s, offset = 3
PUSHIMM 0       // q, offset = 4
PUSHIMM 0
STOREOFF 2      // n=0
PUSHIMM 2       // Size of Stack
MALLOC          // Create a Stack
PUSHIMM 1
ADD             // Stack, not size
DUP
PUSHIMMPA "DV$Stack"
STOREIND        // Store DV address
STOREOFF 3      // Store stack in s
<Similar code for q = Queue()>
```


Code for "q.put(n)"

- The calling code for a method looks like this:

```
<Code to place space for return value on Stack>
<Code to place arguments on Stack>
<Code to save/update FBR>
<Code to place address of method on top of stack>
JSR jsrind
<Code to restore FBR>
<Code to clear arguments from Stack>
```

- Recall: q is treated as an additional (hidden) argument

```
                        // No return value
PUSHOFF 4               // q
PUSHOFF 2               // n
LINK                    // Store/update FBR
PUSHIMM 0               // Offset for put method
PUSHOFF -2              // Address of object (q)
PUSHIND                 // Addr of dispatch vector
ADD                     // Addr of correct method
JSR jsrind              // Method call
POPFBR                  // Restore FBR
ADDSP -2                // Clear arguments
```


Code for "print q.get()"

- The calling code for a method looks like this:

```
<Code to place space for return value on Stack>
<Code to place arguments on Stack>
<Code to save/update FBR>
<Code to place address of method on top of stack>
JSR jsrind
<Code to restore FBR>
<Code to clear arguments from Stack>
```

- Recall: q is treated as an additional (hidden) argument

```
PUSHOFF 0               // Space for RV
PUSHOFF 4               // q
LINK                    // Store/update FBR
PUSHIMM 1               // Offset for get method
PUSHOFF -1              // Addr of object
PUSHIND                 // Addr of dispatch vector
ADD                     // Addr of correct method
JSR jsrind              // Method call
POPFBR                  // Restore FBR
ADDSP -1                // Clear arguments
WRITE                   // Print the result
```

Code for a Constructor (Node)

- For this example, there are no local variables
- There are 3 arguments: the (hidden) object and the two explicit arguments

```
class Node
    {public int data; public Node link;}
    {public Node (data, link) { }
          { this.data = data;
            this.link = link;
          }
    }{ }
```

- Recall: The calling code allocates the space and passes the new object (along with any other arguments)

```
"C$Node$int$Node":
PUSHOFF -3      // Push addr of this
PUSHIMM 1       // Offset for this.data
ADD             // Addr of this.data
PUSHOFF -2      // Push data (the arg)
STOREIND        // Store into this.data
PUSHOFF -3      // Push addr of this
PUSHIMM 2       // Offset for this.link
ADD             // Address of this.link
PUSHOFF -1      // Push link (the arg)
STOREIND        // Store into this.link
JUMPIND         // Return
```


Code for "n = Node(i, n)"

- The calling code for a constructor looks like this

<Push/create object (need size); use as ret value>
<Push arguments>
<Push/update FBR>
<Push/update PC (i.e., jump to constructor)>
<Pop/restore FBR>
<Clear arguments (ret value is left on stack)>

```
PUSHIMM 3       // Size of Node
MALLOC          // Constr for Node
PUSHIMM 1
ADD             // Node, not size
PUSHOFF -1      // Push argument i
PUSHOFF 2       // Push argument n
LINK            // Save/update FBR
JSR "C$Node$int$Node"
POPFBR          // Restore FBR
ADDSP -2        // Clear args (not rv)
STOREOFF 2      // Store into n
```


Code for a Method (Queue.get)

- Method get within class Queue

```
public int get ()
    { Node n; }
    { n = head;
      head = head.link;
      return n.data; }
```

```
"M$Queue$get$":
PUSHIMM 0       // n, offset = 2
PUSHOFF -1      // Push addr of this
PUSHIMM 1       // Offset of head
ADD             // Addr of head
PUSHIND         // Value of head
STOREOFF 2      // n = head
PUSHOFF -1      // Push addr of this
PUSHIMM 1       // Offset of head
ADD             // Addr of head
PUSHOFF -1      // Push addr of this
PUSHIMM 1       // Offset of head
ADD             // Addr of head
PUSHIND         // Head's val (Node's addr)
PUSHIMM 2       // Offset of link
ADD             // Addr of head.link
PUSHIND         // Value of head.link
STOREIND        // head = head.link
PUSHOFF 2       // n's val (Node's addr)
PUSHIMM 1       // Offset of data within Node
ADD             // Address of n.data
PUSHIND         // Value of n.data
STOREOFF -2     // Store into rv
ADDSP -1        // Clear local variables
JUMPIND         // Return
```