Status

• < 2 weeks to go on project 3

• Read Sec. 7.1 through 7.4 by next Monday, May 18
How would we extend Cool to implement...

• Multiple inheritance?
  • Keep a reference to the base class and a list of pointers to base class. When accessing fields in the class, first look in own fields, then look through the list.
  • With SELF_TYPE and static dispatch this could be a nightmare.
Overloading?

- In Cool, we don’t allow function overloading. C++ & Java do.
- How would we allow the equivalent of
  - myfunction(int c) and myfunction(char *p, int d) ?
  - Keep a parameter list and check for duplicate parameter lists
    - myFunc(a : int) : Bool {false};
    - myFunc(b : bool, c : string) : Bool {false}; // OK
    - myFunc(d : int) // Error
Constants

• We have:
  • Red : Int ← 0;

• What would we have to do for:
  • Red : Const Int ← 0;
Enumerated Types?

• Color : Enum ← {Red, Green, Blue}
  • is the same as saying:
    • Red : Const Int ← 0;
    • Green : Const Int ← 1;
    • Blue : Const Int ← 2;

• to use:
  • c : Color;
  • i : int
  • i = Red; // OK
  • c = Green // OK
  • c = 52 // Not OK
Status

• We have covered the front-end phases:
  • Lexical Analysis
  • Syntax Analysis (Parsing)
  • Semantic Analysis

• Now are the back end phases
  • Code Generation (we’ll do this first)
  • Optimization
Before discussing code generation, we need to understand what we are trying to generate.

There are a number of standard techniques for structuring executable code that are widely used.
Outline

• Management of run-time resources
• Correspondence between static (compile-time) and dynamic (run-time) structures
• Storage organization
Run-Time Resources

- Execution of a program is initially under the control of an operating system.

- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)
Memory Layout

Memory

Code

Other Space

Low Address

High Address
Our pictures of machine organization have:

- low address at the top
- high address at the bottom
- lines delimiting areas for different kinds of data

These pictures are simplifications

- E.g., not all memory need be contiguous

In some textbooks, lower addresses are at the bottom.
What is “other space”? 

- Holds all data for the program
- other space = data space

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area
Code Generation Goals

• Two goals:
  • Correctness
  • Speed

• Most complications in code generation come from trying to be fast as well as correct.
Assumptions about Execution

• Execution is sequential: control moves from one point in a program to another in a well-defined order.

• When a procedure is called, control eventually returns to the point immediately after the call.

• Do these assumptions always hold?
Activations

• An invocation of procedure P is an activation of P.
• The lifetime of an activation of P is:
  • All the steps to execute P
  • Including all the steps in procedures that P calls
Lifetimes of Variables

- The *lifetime* of a variable $x$ is the portion of execution in which $x$ is defined.
- Note that:
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept
Activation Trees

- Assumption #2 (execution returns to the point immediately after the call when a procedure returns) requires that when P calls Q, then Q returns before P does.

- Lifetimes of procedure activations are properly nested

- Activation lifetimes can be depicted as a tree.
Class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int { { g(); f(); } };
}

Example
Example 2

```java
Class Main {
    g() : Int { 1 };
    f() : Int { g() };
    main() : Int { { g(); f(); } };
}
```

What is the activation tree for this example?
Example

Class Main {
    g() : Int { 1 }
    f() : Int { g() }
    main() : Int { { g(); f(); } }
}

Main

Stack

Main
g

Example

Class Main {
    g() : Int { 1 }
    f() : Int { g() }
    main() : Int { { g(); f(); } }
}

Main

Stack

Main
g

Monday, May 11, 2009
Example

Class Main {
    g() : Int { 1 };
    f() : Int { g() };
    main() : Int { { g(); f(); } };
}

Main

Stack

f

Main

g

Monday, May 11, 2009
Example

Class Main {
  g() : Int { 1 };
  f() : Int { g() };
  main() : Int { { g(); f(); } };
}

Diagram:
- Main
  - g
  - f
- Stack
  - Main
    - f
    - g
Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures.
Memory Layout

Memory

Low Address

Code

High Address

Stack
Activation Records

• On many machines, the stack starts at high addresses and grows toward lower addresses.

• The information needed to manage one procedure activation is called an activation record (AR) or frame.

• If procedure F calls G, then G’s activation record contains a mix of info about F and G.
What is in G’s AR when F calls G?

- F is “suspended” until G completes, at which point F resumes. G’s AR contains information needed to resume execution of F.
- G’s AR may also contain:
  - Actual parameters to G (supplied by F)
  - G’s return value (needed by F)
  - Space for G’s local variables
  - Note: globals are handled separately
Contents of a Typical AR for G

- Space for G’s return value
- Actual parameters
- Pointer to the previous activation record
  - The control link points to AR of caller of G
  - Or frame pointer
- Machine status prior to calling G
  - Contents of registers & program counter
  - Local variables
- Other temporary values
Example 2, Revisited

Class Main {
    g() : Int { 1 };  
    f(x:Int) : Int { if x=0 then g() else f(x-1)(***)fi };  
    main() : Int { { f(3); (*) } };  
}

AR for f:

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>control link</td>
</tr>
<tr>
<td>argument</td>
</tr>
<tr>
<td>result</td>
</tr>
</tbody>
</table>
Stack After Two Calls to f

- `main`
- `f`
- `f`

- `ret addr(**)`
- `control link`
- `2`
- `result`
- `ret addr(*)`
- `control link`
- `3`
- `result`
• In our case, `main` has no argument or local variables and its result is never used; its AR is uninteresting.

• (*) and (**) are return addresses of the invocations of `f`

• The return address is where the execution resumes after a procedure call finishes.

• This is only one of many possible AR designs

• Would also work for C, Pascal, FORTRAN, etc.
The Main Point

• The compiler must determine, at compile-time, the layout of activation records and generate code that correctly assesses locations in the activation record.

• Thus, the AR and code generator must be designed together.
Discussion

• The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame.

• There is nothing magic about this organization
  • Can rearrange order of frame elements
  • Can divide caller/callee responsibilities differently.

• An organization is better if it improves execution speed or simplifies code generation
Memory Layout with Static Data

- Memory
  - Code
  - Static Data
  - Stack
- Low Address
- High Address
Heap Storage

• A value that outlives the procedure that creates it cannot be kept in the AR.

memory x() { new y }

• The Y value must survive deallocation of x’s AR.

• Languages with dynamically allocated data use a heap to store dynamic data.
Notes

• The code area contains object code.
  • For most languages, fixed size and read only
• The static area contains data (not code) with fixed addresses (e.g., global data)
  • Fixed size, may be readable or writable
• The stack contains an AR for each currently active procedure
  • Each AR usually fixed size, contains locals.
• Heap contains all other data
  • In C, heap is managed by `malloc` and `free`.
  • What about Java and Cool?
Notes

• Both the heap and the stack grow

• Must take care that they don’t grow into each other.

• Solution: start heap and stack at opposite ends of memory and let them grow toward each other.
Memory Layout with Static Data

Memory

Code

Static Data

Heap

Stack

Low Address

High Address
Data Layout

• Low-level details of machine architecture are important in laying out data for correct code and maximum performance

• Chief among these concerns is alignment
Alignment

• Many machines in use are (still) 32 bit (this is changing quickly)
  • 8 bits in a byte
  • 4 bytes in a word (dword, actually)
  • 64 bits or 8 bytes per word (qword) possible but often not used on 64 bit chips (e.g., 64-bit x86)
• Machines are either byte- or word-addressable
• Data is word aligned if it begins at a word boundary
• Or performance penalties for poor alignment
Alignment

- Example: A string
  “Hello”
- Takes 5 characters (without a terminating \0)
- To word-align the next datum, add 3 “padding chars” to the string
- Padding is not part of the string, just unused memory.