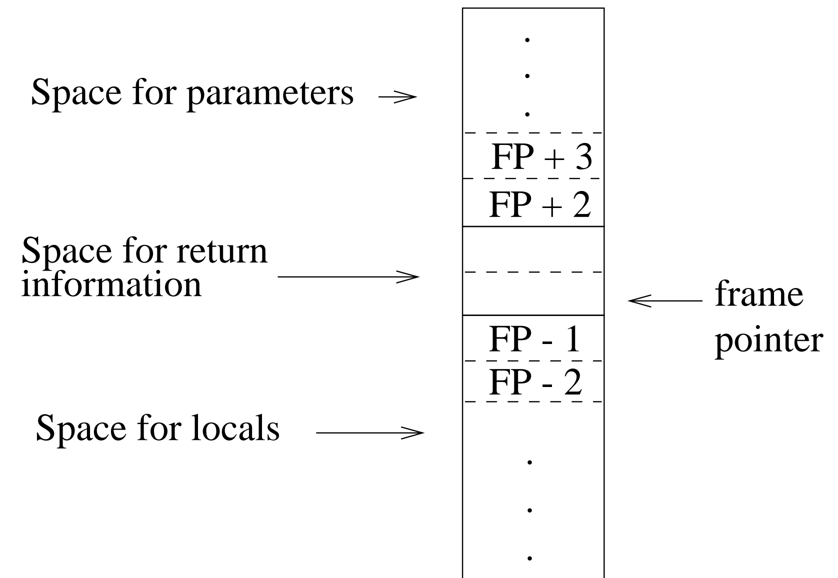
CS 434 Meeting 16 — 4/11/06

Anouncements

1. Phase 3? (Building a little parser with Yacc) should be finished by the end of the week.

Generating Code for Methods

1. There are several issues to consider to prepare to generate code for methods:
   (a) What runtime structures are involved in the invocation of a method.
   (b) How to handle parameters.
   (c) How to handle return values.
   (d) What the basic method header and trailer code looks like.
      - Using the chain of static link pointers
      - saving return information
      - Saving and restoring registers.
   (e) When to generate code for each method while traversing the syntax tree.
2. Recall the basic 34000 call and return sequences:
   - We will follow the usual 68000 convention and assume that A7 is the stack pointer and than A6 is used to point to (the middle of) the frame for the active method.
   - Code at the call site:
```
< Code to pass parameter values >
JSR    proc-name
```
   - Code surrounding the procedure body:
```
proc-name   LINK    A6,#-<local variable space size>
                 < Code for procedure body >
            UNLK
            RTD     #-param-count
```
   This takes care of:
   - allocating the frame
   - saving return PC and FP
   - placing parameters on the stack and removing them
3. This depends on (and results in) the following frame structure:



4. The handling of method parameters in your compiler will reflect simple, but outdated technology. The most glaring simplification/inefficiency will be the use of memory rather than registers to pass parameters and other information between caller and callee.
   - Recognizing the importance of effficient calls, modern compilers (and hardware calling conventions) tend to use registers for parameters when possible. To support methods with many parameters, however, such schemes have to retain the possiblity of storing some parameters in memory. We will keep things simple by placing all parameters in memory on the stack.

- Luckily, in Woolite, all parameter values take just one word of memory. Therefore, at the call site, all you have to do is generate code to determine the parameter's value and then push it onto the stack.

5. Handling return values can be approached in one of two ways:

   - Reserve a special register for the return value.
   - Save an extra word on the stack (just before the parameters) and leave the return value there.

   Saving the result on the stack will prove simpler. This means:

   - output an instruction to subtract 1 from the stack pointer before pushing parameters.
   - get a temporary and output an instruction to pop the top value off the stack into this temporary after control return following a JSR

6. In addition, we will assume that A5 holds a pointer to the object associated with the currently executing method.

   - Recall that the memory allocated for an object will contain space for:
     - All the object's instance variables
     - A pointer to a table containing the addresses of the code for all the method's associated with the current object's class (and a pointer to the similar table for any superclass).
     - A pointer to the instance of the object of the class that statically surrounds the definition of the current object's class that was active when the current object was created.

7. A very fundamental difference between the standard calling sequence and the code you must generate is that you cannot produce JSR instructions that statically specify the name of the target method's first instruction.

   - This is a result of support for inheritance. Consider a program containing code of the form;

```
class program {
    class A {
        void m1( ... ) {
            print( "hi" );
        }

        void m2( ... ) {
            m1( ... );
        }
    }

    class B extends A {
        void m1( ... ) {
            print( "bye" );
        }
    }

    void main() {
        ( new A() ).m2();
        ( new B() ).m2();
    }
}
```

   - The first invocation in main, should print "hi", while the second should print "bye".
   - Both invocations result in the execution of the invocation "m1(...)"; included in the definition of the class A. Clearly, the code we generate for this invocation cannot just JSR to the code for m1 in A.

     Note: If you are just dying to do something clever to improve the quality of the code you produce you should observe that it is safe to skip the method table if the class

containing the method being invoked has no subclasses or if none of the subclasses override the method.

8. As a result, in place of a single JSR instruction, you will have to generate:

   - Code to load the pointer to the correct object's method table.
   - Code to load the pointer to the desired method into a register.
   - a JSR to the desired method.

   or, you can build a table of branches to the methods instead of just a table of their address and then generate:

   - Code to load the pointer to the correct object's method branch table, and
   - a JSR to the JMP to the desired method.

   Since the 34000 doens't do any sophisticated pipelining or branch prediction, the second method will be best.

9. When to generate code (or the beauty of using the assembler):

   - During phase 1, initialize code labels for the "methodtab" fields of each class' decldesc. This will allow you to generate code to reference the table (and thereby generate code to construct objects of the class) before you actually generate any code for the class itself (i.e. its methods).
   - When asked to generate code for a class:
     - Generate code for each of the methods defined in the class. As you start to generate code for each method, you will initialize a code label and place it on the first line of the method's code
     - After all the methods are generated, place the label associated with the "methodtab" field of the class and then output the method table (a sequence of JMP instructions to each of the methods).
     - Finally, generate code for subclasses (or do this in the opposite order if you want all the method tables to come out together).

   The nice thing about using the assembler is how flexibly you can change the order of things. If you prefer to have all the method tables before all the code, you can easily arrange that too!

10. Another difference between method calls in Woolite and the standard 68000/34000 call sequence is the need to deal with the object pointer (A5).

    - For calls of the form

      ```
      meth( ... )
      ```

      in which "meth" is defined in the same class as the method that contains the invocation, this actually isn't an issue since the active object isn't changed so A5 should remain the same.
    - For all other calls, the active object is changed when the new method is invoked.
      - For calls of the form

        ```
        expr.meth( ... )
        ```

        the object produced by "expr" becomes the active object.
      - For calls of the form

        ```
        meth( ... )
        ```

        in which meth is defined in a surrounding class definition (i.e. scope rules rule), you have to follow the chain of static links stored in objects to get "up" to the object corresponding to the class in which the method was defined.
      - In both cases, once the method is completed, you have to restore the original value of A5.
    - To deal with these issues, you need to add code before the actual JSR to save the value of A5 somewhere and replace it with the correct value.
      - It needs to be saved somewhere in the caller's frame. You could allocate a memory temporary or take advantage of a word that is known to be free (the word that we will reserve for return values might work).

11. About that chain of static links....

- I goofed. When I told you to lower expression subtrees I should have told you to perform a similar rewriting of method invocations. During your first phase, if you encounter an invocation of the form

  `meth( ... )`

  you should modify the syntax tree by turning this into a subtree for an expression of the form

  `expr.meth( ... )`

  where "expr" is a chain of refvar nodes that loads the needed values from the chain of static links.

  I said "goofed" in this case because if you try to do this during code generation instead of during the binding phase you will discover that you don't have enough information. The level information in a method's decldesc cannot be used to decide if a method is non-local (since it might have been inherited). You have to instead use the level information in binding stack entries (which are no longer accessible during code generation).

  To understand the need to do this better, consider the example in Figure 1:

  - The method m1 is clearly defined at level 2 (i.e., class program is level 0, class base is level 1, and m1 is defined within base).
  - The invocation of m1 within m2 references m1 as inherited by class outer.
  - Class outer is defined at level 2, so its methods (include m1) are considered to be at level 3.
  - As a result, when inner calls m1, the code generated should only move one level up the chain of static links for inner to get to the level of outer rather than moving through two links to get to the level of base (since two steps up from inner is wrapper rather than base!).

```
class program {
    void main() {
      ...
    }
    class base {
        int y;
        void m1() {  ... y ...     }
    }
    class wrapper {
        class outer extends base {
            class inner {
                void m2() {   m1();  }
            }
        }
    }
 }
```

Figure 1: What nesting level is associated with m1 in outer?

Saving Registers during Calls

1. When generating calls within expression and code for non-void methods, you must include code to ensure that register values are preserved during all calls.

   (a) Registers are typically partitioned into:

   Callee saved registers whose values the called method must save and restore (if it uses them).

   Caller saved registers whose value the calling method must save and restore (if it depends on them after the call).

   It is possible (and not uncommon) to put all registers in one group or the other (callee-saved is the favorite).

   (b) Callee saving has the advantage of keeping the total size of your code small (each proc only contains one set of instructions to save registers).

   Note: You have to generate the register saving instructions before you know what registers need to be saved. The fact that you are generating assembly code lets you leave this problem to the assembler by using a symbolic name for the mask that determines what registers need to be saved.

   (c) Caller saving has the advantage that you only save the registers in use at a particular call rather than all registers ever used in the procedure (although you may end up saving registers that aren't altered by the called procedure).

2. A register allocator can be designed to take advantage of these two classes of registers:

   - Use caller saved registers to hold values whose usefulness begins and ends in a section of code including no calls (entire body of any "leaf" procedure.
   - Use callee saved registers for values produced before and needed after a call.

3. Where are register values saved?

   (a) In the caller's frame — for caller saved registers.

   (b) In the called method's frame — for callee saved.

   In both cases, the stack pointer will be incremented as part of the register saving (rather than treating them as locals counted in determining localsize).