CS 434 Meeting 15 — 4/6/06

Anouncements

1. Phase 2.2 (code generation for control structures) should be finished by the end of the week).
2. Phase 3 (Building a little parser with Yacc).

Constructing the LR(0) Machine for a Grammar

1. To make the ideas we talked about last time precise (and eventually prove that we can use them to construct a correct parser) we need to refresh your knowledge of finite automata a bit.
2. First, recall the structure of a deterministic finite state machine.
   (a) A finite set of states, π.
   (b) An input alphabet, Σ.
   (c) A transition function $\delta : \pi \text{ x } \Sigma \rightarrow \pi$.
   (d) A subset F of π called the set of final states.
   (e) An element π_0 of π called the initial state.
3. While you are at it, recall (or at least note) that we can explain the behavior of a deterministic finite state machine by defining a function that extends δ to strings over the input alphabet. In particular, we can define $\Delta : \pi \text{ x } \Sigma^* \rightarrow \pi$ recursively as
   - $\Delta(\pi, \epsilon) = \pi$
   - $\Delta(\pi, \gamma x) = \delta(\Delta(\pi, \gamma), x)$

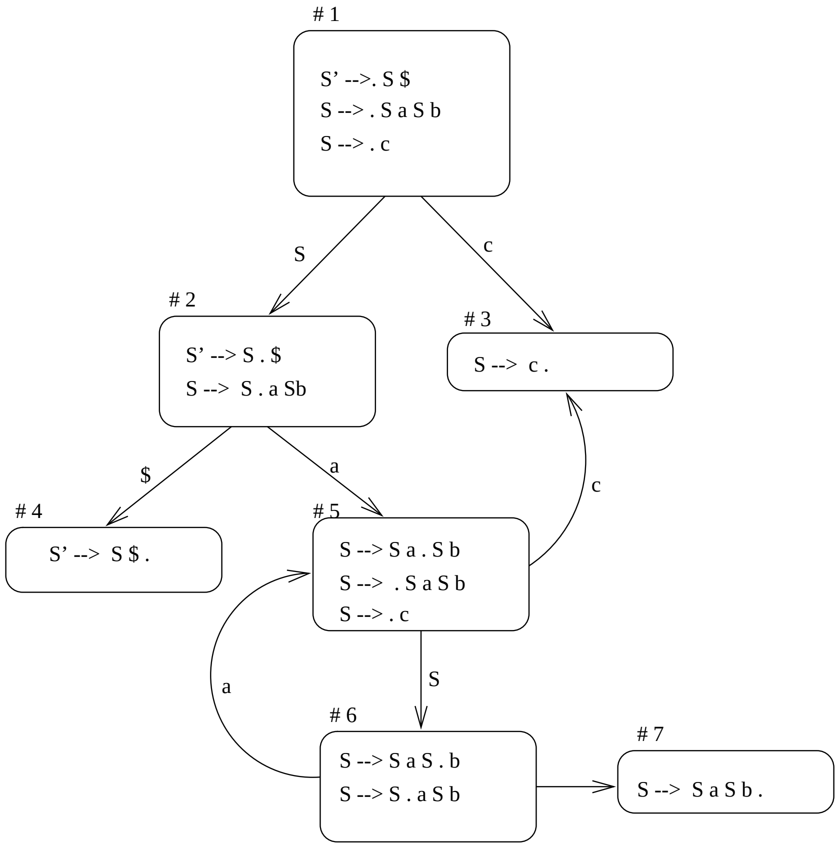
   and then state that the language accepted by the machine is
   $$\{\gamma \in \Sigma^* \mid \Delta(\pi_0, \gamma) \in F\}$$
4. Now, we can give a general definition of the LR(0) machine for an arbitrary grammar G.

   Of course, we need a few more definitions:

   goto Given a set of LR(0) items for a grammar G, we define
   $$goto(\pi, x) = \{[N \rightarrow \beta_1 x . \beta_2] \mid [N \rightarrow \beta_1 . x \beta_2] \in \pi\}$$

   closure Given a set π of LR(0) items for a grammar G with productions P, we define closure(π) to be the smallest set of LR(0) items such that:
   (a) closure(π) $\supseteq \pi$
   (b) if $[N_1 \rightarrow \beta_1 . N_2 \beta_2] \in$ closure(π) and $N_2 \rightarrow \beta_3 \in P$ then$[N_2 \rightarrow . \beta_3] \in$ closure(π)
5. The closure of a set of LR(0) items can be computed using a simple (but important) little algorithm
   - An algorithm to compute closure(π)
     (a) set π' equal to π.
     (b) while there is some $[N \rightarrow \beta_1 . M \beta_2] \in \pi'$ such that $M \rightarrow \beta_3 \in P$ and $[M \rightarrow . \beta_3] \notin \pi'$ add $[M \rightarrow . \beta_3]$ to π'.
6. With these definitions and the assumption that the start symbol S of G is replaced by a new start symbol S' and that the rule $S' \rightarrow S\$$ is added to the set of productions (The \$ just stands for end-of-input). The definition of the LR(0) machine is:
   - Let the set π of states of the machine be the set of all sets of LR(0) items for G.
   - Let the set of final states be all states except the state corresponding to the empty set of LR(0) item.
   - Let the initial state be the state corresponding to the set
     $$\text{closure}(\{[S' \rightarrow . S\$]\})$$
   - Let the transition function $\delta : \pi \text{ x } (V_n \cup V_t) \rightarrow \pi$ be defined by:
     $$\delta(\pi, x) = closure(goto(\pi, x))$$
7. A somewhat interesting example.

$< \text{S'} > \rightarrow\ < \text{S} > \$$
$< \text{S} >\ \rightarrow\ < \text{S} > \text{a} < \text{S} > \text{b} \mid \text{c}$



Using the LR(0) machine

1. The key fact about the LR(0) machine for a grammar is:

   Theorem $[N{\rightarrow}\beta_1.\beta_2] \in \Delta(\pi_0, \gamma)\ iff\ [N{\rightarrow}\beta_1.\beta_2]$ is valid for γ.

2. If the LR(0) set associated with some state includes an item of the form $[N{\rightarrow}\beta.]$ That implies that β might be the handle.

   - In general, whether it is the handle depends on the input that follows

   We will call such an item a *reduce item.*

3. If the only item in the LR(0) set associated with some state is an item of the form $[N{\rightarrow}\beta.]$ then, β must be the handle.

4. If in the LR(0) machine for a grammar G, any state that contains a reduce item contains only that item, then the action that should be performed by a parser is determined as follows. Run the LR(0) machine on the string stored in the parser's stack. Then,

   - If the LR(0) machine's state consists of a single reduce item, reduce by the production used in the item.
   - If the current state contains non-reduce items, shift the next input symbol onto the stack.
   - If the current state is the empty state, declare an error.

   In this case, we say that G is an LR(0) grammar.

5. Note that we don't have to start our FSM over at the bottom of the stack after each reduction to continue parsing. We can avoid this by recording the state of the FSM as it scans each symbol on the stack with the symbol on the stack. Then, after reducing a suffix of the stack, we just start the LR(0) machine in the state associated with the top symbol left in the stack (rather than starting over from scratch).

6. The "trick" is simply to keep another stack (parallel to the stack that store the head of the sentential form we are working on) that stores the state the LR(0) machine would enter for each character in the "regular" stack.

7. In fact, we can completely replace the stack of symbols by a stack of the FSM states that would have been associated with them.

8. To make this clearer, let's walk through the parse of an input string using the LR(0) machine shown in Figure 1. I have numbered each state in the diagram of the machine and used these number to identify states in the stack snapshots.

Generating Code for Procedures

Stack Contents (“real” and “virtual”)	Remaining Input
1	c a c b a c a c b b
ϵ	
1 3	a c b a c a c b b
c	
1 2	a c b a c a c b b
S	
1 2 5	c b a c a c b b
S a	
1 2 5 3	b a c a c b b
S a c	
1 2 5 6	b a c a c b b
S a S	
1 2 5 6 7	a c a c b b
S a S b	
1 2	a c a c b b
S	
1 2 5	c a c b b
S a	
1 2 5 3	a c b b
S a c	
1 2 5 6	a c b b
S a S	
1 2 5 6 5	c b b
S a S a	
1 2 5 6 5 3	b b
S a S a c	
1 2 5 6 5 6	b b
S a S a S	
1 2 5 6 5 6 7	b
S a S a S b	
1 2 5 6	b
S a S	
1 2 5 6 7	ϵ
S a S b	
1 2	ϵ
S	

Figure 1: LR(0) parser stack trace

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:

   - 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
   ```

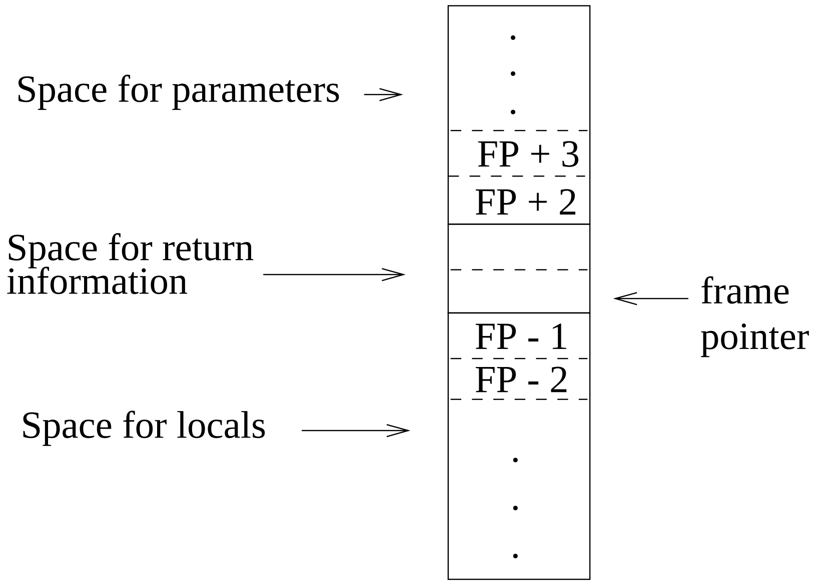
   This takes care of:

   – 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. 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.

5. 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.

6. 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 obvious when dealing with expressions of the form

     ```
     target.method( ... )
     ```

     where it is obvious that the interpretation of "method" may depend on the type of the object currently associated with the variable "target" (which can be any subclass of the type of "target").
   - It is also true for simple examples of the form

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

     since the object executing the method that contains such a simple invocation may be a member of a subclass of the class in which that method's definition was written. In that case, you again have to use the object's method table in case "meth" has been overridden.

   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.

7. 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.

8. 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 amazing 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!

9. 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

       ```
       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.
     - For calls of the form

       `expr.meth( ... )`

       the object produced by "expr" becomes the active object.
     - 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).

10. 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).

11. 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.

12. 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.

13. 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.

14. 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