CS 434 Meeting 7 — 2/23/06

Anouncements

1. Phase 1.2 has now officially begun. Don't forget to change the SUB-PHASE variable in your makefile. It will be due on Thursday, March 2nd.

Parse Trees and Ambiguity

1. For our purposes, it is not enough that a grammar specify the set of strings that belong to a language, we also want a grammar to impose grammatical structure on strings since this structure influences the meaning associated with a string.

2. Representing structural information as a tree provides a better way to eliminate irrelevant choices about the order of non-terminal expansion.

   Parse Tree A *parse tree* for a sentential form S of a grammar G is a tree in which each node is labeled with an element of $(V_t \cup V_n)$ in such a way that:

   (a) The root is labeled with the start symbol,

   (b) the nodes of the frontier of the tree (i.e. the leaves of the tree) are labeled with the symbols that form S, and

   (c) each interior node is labeled with some non-terminal, N, such that $N \rightarrow \alpha \in P$ and α is the string of labels found on the node N's children.

3. Given a parse tree, we can extract a derivation by walking the tree. Depending on whether we visit the children of each node left-to-right or right-to-left, we will obtain a derivation with one of two special forms:

   Direct leftmost derivation Given a grammar, $G = (V_t, V_n, S, P)$, and two strings x and y in $(V_t \cup V_n)^*$ such that $x = \alpha A \beta$ and $y = \alpha\gamma\beta$ where $\alpha \in V_t^*$, $\gamma, \beta \in (V_t \cup V_n)^*$ and $(A, \gamma) \in P$ we say that y can be directly derived leftmost from x. In this case we write $x \underset{\text{lm}}{\Longrightarrow} y$.

   Direct rightmost derivation Given a grammar, $G = (V_t, V_n, S, P)$, and two strings x and y in $(V_t \cup V_n)^*$ such that $x = \alpha A \beta$ and $y = \alpha\gamma\beta$ where $\beta \in V_t^*$, $\alpha, \gamma \in (V_t \cup V_n)^*$ and $(A, \gamma) \in P$ we say that y can be directly derived rightmost from x. In this case we write $x \underset{\text{rm}}{\Longrightarrow} y$.

   Left-most Derivation Given a grammar, $G = (V_t, V_n, S, P)$, and two strings x and y in $(V_t \cup V_n)^*$ we say that x *derives* y *leftmost* if there exists a sequence of string $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_m$ all in $(V_t \cup V_n)^*$ such that

   (a) for all $i < m$, $\alpha_i \underset{\text{lm}}{\Longrightarrow} \alpha_{i+1}$,

   (b) $x = \alpha_0$, and

   (c) $y = \alpha_m$.

   In this case we write $x \overset{*}{\underset{\text{lm}}{\Longrightarrow}} y$.

   Rightmost Derivation Given a grammar, $G = (V_t, V_n, S, P)$, and two strings x and y in $(V_t \cup V_n)^*$ we say that x *derives* y *rightmost* if there exists a sequence of string $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_m$ all in $(V_t \cup V_n)^*$ such that

   (a) for all $i < m$, $\alpha_i \underset{\text{rm}}{\Longrightarrow} \alpha_{i+1}$,

   (b) $x = \alpha_0$, and

   (c) $y = \alpha_m$.

   In this case we write $x \overset{*}{\underset{\text{rm}}{\Longrightarrow}} y$.

   Intuitively, a derivation is rightmost (leftmost) if at each step it is the rightmost (leftmost) non-terminal that is replaced by the right-hand side of some production.

   The first derivation shown earlier happens to be a leftmost derivation of x a x z y y in G.

Attribute Grammars as a framework for semantic processing

1. Semantic processing routines often return values that summarize the properties of the sub-phrases to which they were applied.

   - For example, your function to process expressions should return the type descriptor of the expression.

2. It can be useful to think of the values returned by such semantic processing routines as labels that get affixed to the nodes of the parse tree or abstract syntax tree as it is processed.

   - These "labels" are often referred to as "attributes". Those interested in building formal systems for specifying the details of semantic processing often base their schemes on the notion of such attribute values.
     - A grammar for the souce language or an abstract grammar for abstract syntax trees is annotated with rules that associate attribute values with the terminals and non-terminals of the grammar.
     - Synthesized attributes are those whose values are determined by a node's subtrees (a phrase's substructure). The type of an expression might be an example.
     - Inherited attributes are those whose values are determined by the context of a phrase (the current scope's nesting level, the entire symbol table (in a purely applicative compiler)).

Lowering Expression Sub-trees

1. We can solidify our understanding of the run-time layout of memory (and get a head start on code-generation) by considering how to transform the trees produced by the parser to represent references to variables into trees that explicitly describe the address calculations and memory references involved.

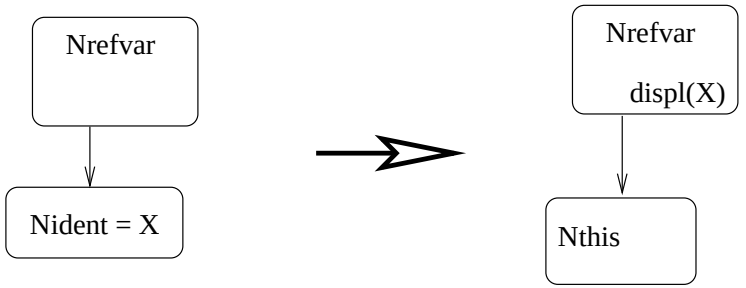
2. A reference to an identifier as a variable will be represented in the syntax tree by an Nrefvar node with an Nident node for the variable's name as its child.

3. To actually access a variable in memory at run-time the hardware must add the displacement to the variable to the frame pointer for the method or pointer to the object in which the variable is stored.

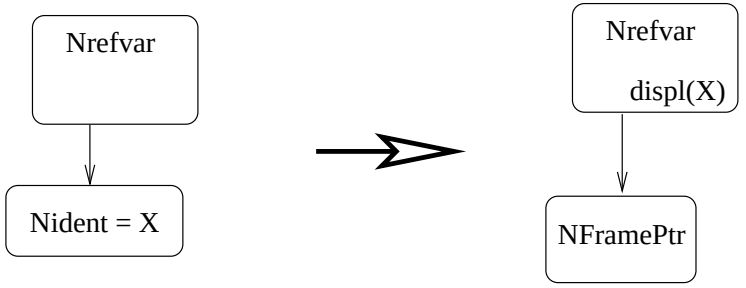
   - We can think of an Nrefvar node as representing the value stored in the memory location described by its child.
   - We already have node types to represent addition (Nplus) numeric constants (Nconst), and the pointer to the current object (Nthis), so if we add a node type to represent a reference to the active method's frame (NFramePtr), we can explicitly describe the steps required to access a variable.
   - There is also a field in the Nrefvar node that can be used to hold the displacement to a variable relative to a given memory location.

4. This leads to the following simple transformation for simple variable references:

   - For instance variables local to the class of the current method, we just specify the offset to the variable relative to the "this" pointer.

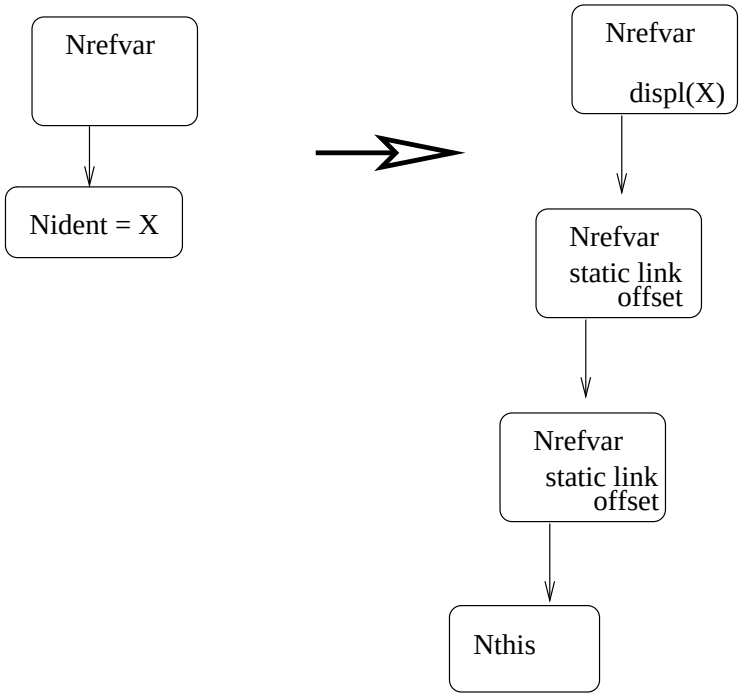


   - For variables declared as locals in a method and for method parameters, we use the offset to the variable relative to the method frame pointer.



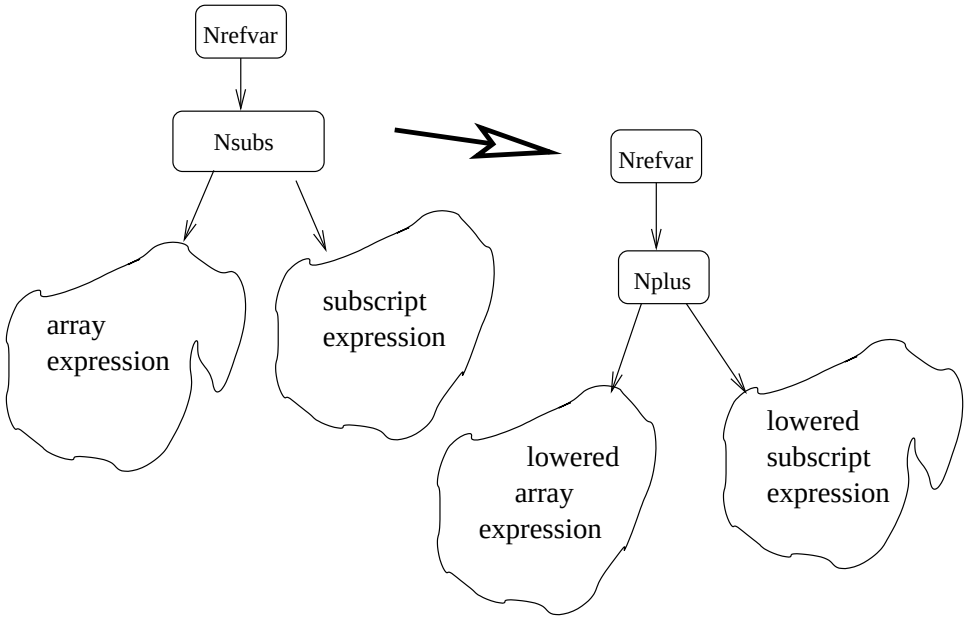
   - Given the organization of the static links, all we need to do for references to non-local variables is follow the chain of static links

up the correct number of levels. For example, the following tree could be used to reference a variable declared two levels above the current class.



5. Similar transformations can be appled to more complex variables. Nsubs nodes can be transformed into trees that describe the subscript calculations required. In such a transformed tree:

   - the root will again be an Nplus node
   - the left subtree will just be a transformed version of the array sub-variable tree (as with the record sub-variable in the Nselect case).
   - the right subtree will in general be an Ntimes node multiplying the subscript expression by the element size. In Woolite, since all array elements occupy just one word, we can leave out the Ntimes node.



6. There will be several advantages to making these transformations. Because they replace special purpose tree nodes (Nident and Nsubs) with nodes types that would already be present in the tree (Nplus, Ntimes), they reduce the number of cases to be handled by later phases (code generation, optimization).

7. There is a "hook" included in the format of the Nrefvar node to support later optimizations:

   - A terrifying consequence of the transformations I have suggested is that once they are complete, there would be little information left in the tree about which variables are being referenced by the expressions in a program.

     To avoid this, the Nrefvar node includes a field that you may someday use to hold a pointer for a descriptor of the variable being referenced.

Code Generation for Expressions

1. First, I would like to give a quick overview of a simple, "ad hoc" approach to code generation in hopes of:

   - showing you that code generation for expressions can be approached as a process of applying "code templates" to the syntax tree,
   - making you recognize some of the issues involved in selecting the type of "attribute" one will associated with expression sub-trees, and
   - making you understand the desirability of a clear distinction between "high level" and "low level" code generation.

2. To do this, we will limit our attention to expressions including simple arithmetic operators:

   - We will see later that operators typically used in decision making (relationals and logicals) warrant a very different approach.

3. When I speak of applying "code templates" I am basically talking about the fact that the correct code for an expression can be generated using very little information about the context in which the expression occurs.

   - The correct code for 'expr1 + expr2' will look something like:

     code to compute the value of expr1
     code to compute the value of expr2
     move expr2's-value,location-for-result
     add expr1's-value,location-for-result

   All one has to do is make sure that each code generation routine tells its caller where the results can be found.

4. Each code generation routine will use the "attribute value" it returns to tell its caller where the result of the code it produces can be found.

   We will call these attributes *operand descriptors*.

5. To illustrate the use of operand descriptors, consider the following skeleton of a very naive routine to generate code for the addition operator. It assumes that "operand_desc" is the type used to represent the attribute values returned during code generation for expressions.

```
operand_desc *expr_gen(node * expr)
{ operand_desc *left_loc, *right_loc, *new_loc;

  switch (expr->internal.type) {
     .
     .
  case Nplus:
    left_loc = expr_gen(expr->internal.child[0]);
    right_loc = expr_gen(expr->internal.child[1]);

    new_loc = get_temporary();
    output ‘‘move left_loc,new_loc’’;
    output ‘‘add right_loc,new_loc’’;

    opFree(left_loc);
    opFree(right_loc);

    return new_loc;
    }
    .
    .
```

6. When writing such a routine, we would rather not have to worry about

   - How to get a temporary.
   - What type of "move" instruction we actually need to generate if left_loc is a constant, variable in memory, etc.

7. The issues of allocation registers and other temporaries and actual instruction selection decisions that depend on the operand types should

be handled by "low level" code generation routines (with names like get_temporary and output_instructions).

8. Routines like the "expr_gen" function sketched above form the "high level" component of the code generator in that they are concerned with mapping the operations of the source language into the operations the hardware can perform, but not in the precise details of each instruction generated.

9. Maintaining a clear distinction between high level and low level code generation tasks will both keep your code well organized and result in a code generator that could be revised to output code for a different target machine with minimal effort.

Operand Descriptors

1. In addition to providing a mechanism for communication between high-level code generation routines, the operand descriptor type is an essential component of the interface between the high and low level code-generation modules.

2. The type used for operand descriptors should be flexible enough to handle several possibilities:

   registers Since access to values in registers is generally faster than access to memory, we would like the code we generate to keep intermediate results in registers whenever possible.

   memory locations Since we will eventually run out of registers, our compiler may have to store some intermediate results in memory. Even if this were not the case, other factors would make it necessary/desirable to have operand descriptors for memory locations.

   - when we process an expression like `x+y` we would like to produce something like:

     ```
     move   x,D1
     add    y,D1
     ```

     rather than

     ```
     move   x,D1
     move   y,D2
     add    D2,D1
     ```

   - To make this possible, the routine that "generates code" for the sub-expression "y" has to be able to simply return a descriptor for y where it resides in memory without producing any code.

   constants When processing an expression like `x+1` we would like to produce the code:

   ```
   move   x,D1
   add    #1,D1
   ```

   rather than

   ```
   move   x,D1
   move   #1,D2
   add    D2,D1
   ```