CS 434 Meeting 9 — 2/2/06

Anouncements

1. Phase 1.2 is still not due today.
2. Assignment on formal grammars is due today
3. Phase 2.1starts today. Suggested completion deadline is in 12 days.

Generating Code for Conditionals

1. We now want to look at how to generate code for constructs that involve the use of branch or jump instructions. Clearly control structures (such as if and while statements fall in this category). We will see that it also makes sense to include the operators that occur in most conditional expressions (relationals and logical operators).
2. First, consider the "obvious" code template for an if statement of the form:

   ```
   if conditional then stmt 1 else stmt 2
   ```

   Namely:

   ```
           code to compute value of conditional
           CMP #0, cond-result
           BEQ ELSEPART2
           code for stmt 1
           BRA JOINUP2
   ELSEPART2
           code for stmt 2
   JOINUP2
   ```

3. We will quickly find that this "obvious" code is not such a good idea, but to prepare to better understand a more sophisticated alternative, let's first make sure we have the mechanisms needed to generate code based on this simple template.
4. First, just as we had to introduce operand descriptors to handle temporaries, we must introduce some sort of structure to handle the code labels we will branch to.

   - Since you will be generating assembly code, you can just use strings to hold label names.
     - We must assume a routine 'genlabel' that can generate unique labels and a routine 'placelabel' which will associate a label with the next instruction we generate.
     - The routines that output branch instructions will accept labels as operands.

5. Given such a "codelabel" type, we could easily write a function to generate code for if statements:

   - To emphasize the issue of object lifetime (or perhaps just to be contrary), I do not depend on malloc and free to manage code labels. Instead, I prove that C is a wonderful (or horrible) language by allocating them in function frames and passing pointers to them all over the place!
   - We will also ignore the possibility of an empty else part for now.

   ```
   gen_if(node * ifstmt)
   {  codelabel elselabel, joinlabel;
       opdesc *condValue;

      genlabel(&elselabel);
      genlabel(&joinlabel);

      condValue = genexpr(ifstmt->internal.child[0],  FALSE);

      output( "CMP", "#0", condValue );
      outputBranch( "BEQ", &elselabel );
      gen_stmt(ifstmt->internal.child[1]);
      outputBranch( "JMP", &joinlabel );
      place_label(&elselabel);
   ```

```
    gen_stmt(ifstmt->internal.child[2]);
    place_label(&joinlabel);
}
```

6. Now, consider the "obvious" code for a simple condition like "x > 0":

```
     CMP  X,#0
     BGT  SETTRUE
     CLR  DO
     BRA  JOINUP1
SETTRUE MOVE #1,DO
JOINUP1 ...
```

7. This code looks reasonable until you combine it with the code that would be generated by the if statement routine we proposed if the condition were included in an if statement of the form

```
if ( x $>$ 0 ) {
     stmt1
} else {
     stmt2
 }
```

yielding:

```
     CMP  X,#0
     BGT  SETTRUE
     CLR  DO
     BRA  JOINUP1
SETTRUE MOVE #1,DO
JOINUP1 CMP  #0, DO
     BEQ  ELSEPART2
code for stmt1
BRA  JOINUP2
ELSEPART2
code for stmt 2
JOINUP2
     ...
```

8. you would really like the code for a statement of the form

   if x > 0 then stmt 1 else stmt 2

   to look like:

```
     CMP X,#0
     BLE ELSEPART
     code for stmt 1
     BRA JOINUP
ELSEPART
     code for stmt 2
JOINUP ...
```

9. To avoid silly code like this, we will implement two distinct code generation procedures for expressions.

   One generates code that leaves the value of the expression in a location described by the operand descriptor returned (genexpr). This is the routine we discussed last time. This routine will be called to process expressions used as parameters, in assignment statements, etc.

   The other expression code generator will generate code to alter the flow of control based on the value of the expression (gen-cond-expr). This routine will be used to generate code for expressions used as conditions in loops and if statements.

10. When an expression is used as a boolean, we generally want the code for the expression to either "fall through" to the then part or loop body or "branch around" the then part or loop body. To generate nice code we need to tell the code generating routine when we want to branch around (i.e. when the boolean is true or false) and where we want to go to. So, gen-cond-expr takes 2 parameters in addition to an expression subtree:

sense which is a boolean. If it is true then we want to "branch around" if the boolean is true.

target which is the code-label to which the code should "branch around".

11. By examining a (not particularly smart) version of gen-if-statement, we can see how gen-cond-expr can be used (we will continue to assume that there is an else part):

```
gen_if(node * ifstmt)
{  codelabel elselabel, joinlabel;

   genlabel(&elselabel);
   genlabel(&joinlabel);

   gen-cond-expr(ifstmt->internal.child[0],
                 FALSE,
                 &elselabel);
   gen_stmt(ifstmt->internal.child[1]);
   output( ``JMP'',  &joinlabel );
   place_label(&elselabel);
   gen_stmt(ifstmt->internal.child[2]);
   place_label(&joinlabel);
}
```

12. gen-cond-expr must be able to generate code for any expression. The most interesting cases as far as our effort to ensure that the code we generate for if statements is efficient are those involving relational and logical operators. Accordingly, as a first sketch of the code for the gen-cond-expr function we will limit our attention to separating out the three cases it must address:

```
gencondexpr(node *expr, int sense, codelabel *target)
{
   if ( expr's operator is an arithmetic one ) {
      gencondarithmetic( expr, sense, target );
   } else if ( expr's operator is a relational )
      genrelational(expr,sense,target)
   else
      genlogical(expr,sense,target)
}
```

13. The interesting work is going to be done in genrelational and genlogical.

```
genrelational(node *expr, int sense, codelabel *target)
{  oprndesc *leftop, *rightop;

   leftop = genexpr(expr->internal.child[0])
   rightop = genexpr(expr->internal.child[1])

   output "CMP leftop,rightop"

switch (expr->internal.type) {
   case Neq:
       if ( sense )
          output "BEQ target"
       else
          output "BNE target"
       break;
   case Nne:
       ...
```

14. Combining this with the routine for generating if statements sketched above we can see how this produces the type of code we would want for simple if statements that use a single relational as a condition.

15. In addition to handling relational operators better than genExpr, this scheme leads to a simple implementation of "short-circuit" logical operators. Consider just a bit of genlogical:

```
genlogical(node *expr, int sense, codelabel *target)
{
   if (operator is  'and') {
      genlabel( &fallthru );
```

```
        if ( sense ) {
            gencondexpr( left-sub-expr, FALSE , &fallthru);
        } else
            gencondexpr( left-sub-expr, FALSE , target );
        gencondexpr( right-sub-expr, sense, target );
        placelabel( &fallthru);

    } else if ( operator is 'or' ) {
        . . .
    } else if ( operator is not ) {
        . . .
```

16. gen-cond-expr must be able to generate code for any expression, including simple arithmetic operations. The easiest way to do this is to count on the genexpr function we discussed in the last class to calculate the value of the expression and then compare it to 0.

```
gencondarithmetic(node *expr, int sense, codelabel *target)
{  oprndesc *valdesc;

      valdesc = genexpr(expr)
      output "CMP #0,valdesc"
      if ( sense )
         output "BNE target"
      else
         output "BEQ target"
}
```

17. Genexpr can use a similar trick to generate code for logical and relational operators. That is, genexpr will call gen-cond-expr for such operators.