# CS 434 Meeting 10 — 3/7/06

## Anouncements

1. I will be away this afternoon and miss office hours.

## Generating Code for expressions used as conditions (cont.)

1. 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"
}
```

2. 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.

## Generating Code for REAL Control Structures

1. Now that we know how to handle conditionals, we can consider the control structures in which they are used.

   - Consider the code that the if statement routine sketched last class would produce for:

```
if x <> 0 then
   if y <> 0 then  stmt 1
             else  stmt 2
else
   stmt 3
```

   It will look like:

```
            CMP #0,x
            BEQ label3

            CMP #0,y
            BEQ label5
              <code for stmt 1>
            JMP label6
label5        <code for stmt 2>
label6      JMP label4
label3        <code for stmt 3>
label4
```

   – Note: the branch after stmt 1 branches to another branch.

2. Just as it was helpful to pass a branch target as a parameter to genCondExpr, we can generate better code for control structures if we branch targets as inherited semantic information to our code generation routines for statements. In particular, we must pass each statement a label for the statement that follows it, sometimes called its *continuation*.

   - The code to handle the if statement then becomes:

```
gen_if(node * ifstmt, codelabel *nextlabel)
{  codelabel lab1;
   oprnd_desc *result;

   genlabel(&lab1);

   gencondexpr( ifstmt->internal.child[0]
                FALSE, &lab1 );
   gen_stmt(ifstmt->internal.child[1],
            nextlabel);
   output 'JMP nextlabel'
   place_label(&lab1);
```

```
    gen_stmt(ifstmt->internal.child[2],
             nextlabel);
}
```

which would produce the following code for the example above (assuming the routine that called the if statement passed label4 as nextlabel):

```
            CMP #0,x
            BEQ label3

            CMP #0,y
            BEQ label5
              <code for stmt 1>
            JMP label4
label5        <code for stmt 2>
            JMP label4
label3        <code for stmt 3>
label4
```

- Note that the JMP after the code for stmt2 no longer branches to a JMP.

3. Notice that it is not just the if statement routine that expects "nextlabel" as a parameter. Even the generic "genstmt" expects this parameter! This is because genstmt may end up finding that it was asked to process an if statement (or a while loop) which will actually use "nextlabel" as the target of a branch.

   Then again, it may find out that it was called for an assignment statement and not use "nextlabel" at all.

4. To make this work, you will have to place a label after each statement in a statement list (except the last) so that you have something to pass to "genstmt". That is, your code to generate a statement list will have a loop that runs through the list of statements generating code for them. As it does this, it will have to generate a label to place between each pair so that it can pass that label as the nextlabel for the statement that precedes it.

5. To keep the code you generate more readable, you might want to store a flag in each codelabel indicating whether it was used. Then, when asked to place the label, only place it in the output if it was actually used.

   Beware that the label placed at the start of a while loop body will not be used until after it has been "placed".

## Parsing: The Problem of Finding a Derivation

1. Given a grammar G and a string $\sigma$ that one believes is a member of L(G) there are two basic ways that one can attempt to find a derivation of $\sigma$ from S.

   **Top-down parsing** Begin with the start symbol of G and repeatedly substitute the left hand side of a production for a non-terminal until the start symbol has been rewritten to match $\sigma$.

   **Bottom-up parsing** Begin with $\sigma$ and repeatedly "simplify" the string by replacing a sub-string that matches the right hand side of a production by the non-terminal on its left hand side until $\sigma$ has been simplified to S.

2. In our discussion we will make the assumption that we want to produce derivations "on-the-fly". That is, that we do not want to hold the entire input string in memory but wish to generate a derivation while making a single pass through the input string.

## Top-down Parsing

1. At any step in the process of a top-down parse, one has a sentential form that one wishes to re-write so that it more closely matches the target string ($\sigma$). At each such step one must make two choices:

   (a) which of the non-terminals in the current sentential form to replace (left-most to support left-to-right processing of input).

   (b) which production to apply to the selected non-terminal.

2. To eliminate the evil influence of intuition, let's consider finding a derivation given a not very meaningful grammar:.

$$< S > \rightarrow \text{ a} < R > | \text{ b} < S > \text{b} < R >$$
$$< R > \rightarrow \text{ b} < R > | \text{ a}$$

3. Consider the process of using a top-down parser to find a derivation for 'bbaababa' relative to the grammar given above.

| Matched Terminals | Tail of Sentential Form | Pending Input |
|---|---|---|
| | $< S >$ | bbaababa |
| b | $< S >$ b $< R >$ | baababa |
| bb | $< S >$ b $< R >$ b $< R >$ | aababa |
| bba | $< R >$ b $< R >$ b $< R >$ | ababa |
| bbaab | $< R >$ b $< R >$ | aba |
| bbaabab | $< R >$ | a |
| bbaababa | $\epsilon$ | $\epsilon$ |

4. Note that:

   (a) If we concatenate "Matched" and "Tail of Sentential Form" we always obtain a complete sentential form of the grammar,

   (b) the "Tail of Sentential Form" column behaves like a stack.

   A top down parser can be implemented by explicitly maintaining this stack as a data structure.

5. To make our parse "deterministic", we want to decide how to expand the first terminal on the stack based only on what we have matched so far and on some finite prefix of the remaining input. If this is possible using a prefix of lenght k, we say that the grammar is LL(k).

6. In many cases, this is not possible for any k.

   - Consider the productions:

     $< \text{stmt} > \rightarrow$ if $< \text{expr} >$ then $< \text{stmt} >$ end
     | if $< \text{expr} >$ then $< \text{stmt} >$ else $< \text{stmt} >$ end

   - Suppose that we have generated a sentential form in which the left-most non-terminal is $< \text{stmt} >$ and the next input characters to be read is "if". Which production should we choose?

7. For most languages, however, we can find a grammar in which one can determine which production to use next by just looking at the first unmatched character. Such a grammar is called an LL(1) grammar.

8. The following grammar:

   $< \text{stmt} > \rightarrow$ if $< \text{expr} >$ then $< \text{stmt} >$ $< \text{iftail} >$
   $< \text{iftail} > \rightarrow$ else $< \text{stmt} >$ end
   | end

   is obviously LL(1) because:

   (a) The right hand side of each production begins with a terminal, and

   (b) if two productions have the same left hand side, then their right hand sides begin with different terminal symbols.

   A grammar with these two properties is said to be an S-grammar. Any S-grammar is LL(1).

9. In the case of top-down parsing, this assumption of a determinisitic parse produced using a single scan through the input leads to the production of left-most derivations.

   - If at some point we have derived the sentential form $xA\beta$ (where $x$ is a string of terminals, $A$ is a non-terminal and $\beta$ is composed of terminals and non-terminals) while trying to parse $\sigma$, we would want to read in at least the prefix $x$ of $\sigma$ before proceeding further.
   - If we expand $A$ at this point, any prefix of terminals included in the left-hand side we substitute for $A$ will need to be checked against the next input characters following $x$.
   - If we instead expand some non-terminal in $\beta$ we will either need to read past all the terminals that will eventually be matched

by $A$ (saving them so that we can check that they match later) or have to remember to check the correctness of the substitution made for $A$ later.

10. One of the attractions of top down parsing is that there is a simple scheme for implementing a top down parser in any language that supports recursion. The following procedure skeletons show how such a "recursive descent" parser for the S-grammar:

    $< \mathrm{S} > \ \rightarrow \ \mathrm{a} < \mathrm{R} > \mid \mathrm{b} < \mathrm{S} > \mathrm{b} < \mathrm{R} >$
    $< \mathrm{R} > \ \rightarrow \ \mathrm{b} < \mathrm{R} > \mid \mathrm{a}$

    would look (it assumes that "ch" holds the next input character to be processed):

```
procedure R;
    if ch = 'b' then
         getnextchar;
         R;
    else if ch = 'a' then
         getnextchar;
    else
         error
    end
end R;

procedure S;
    if ch = 'a' then
         getnextchar;
         R;
    else if ch = 'b' then
         getnextchar;
         S;
         if ch = 'b' then
             getnextchar;
         else
             error;
         end;
         R;
    end
end R
```

11. One of the nice things about recursive descent parsing is that you can "massage" the code instead of the grammar.