CS 434 Meeting 11 — 3/9/06

Anouncements

1. Distribute assignment 2.

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 (σ). 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 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 length k, we say that the grammar is LL(k).

3. In many cases, this is not possible for any k.
   - Consider the productions:
     $$\begin{array}{rl} <\text{stmt}> \rightarrow & \text{if} <\text{expr}> \text{then} <\text{stmt}> \text{end} \\ | & \text{if} <\text{expr}> \text{then} <\text{stmt}> \text{else} <\text{stmt}> \text{end} \end{array}$$
   - Suppose that we have generated a sentential form in which the left-most non-terminal is < stmt > and the next input characters to be read is "if". Which production should we choose?

4. 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.

5. The following grammar:
   $$\begin{array}{rl} <\text{stmt}> \rightarrow & \text{if} <\text{expr}> \text{then} <\text{stmt}> <\text{iftail}> \\ <\text{iftail}> \rightarrow & \text{else} <\text{stmt}> \text{end} \\ & | \ \text{end} \end{array}$$
   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).

6. 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:
   $$\begin{array}{rcl} <\text{S}> & \rightarrow & \text{a} <\text{R}> | \text{ b} <\text{S}> \text{b} <\text{R}> \\ <\text{R}> & \rightarrow & \text{b} <\text{R}> | \text{ a} \end{array}$$
   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
```

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

Bottom-up Parsing

1. Recall that in a bottom-up parser, we repeatedly simplify sentential forms (starting with the input) by replacing the right hand side of a production by the left hand side.

   - Again consider the process of parsing the string 'bbaababa' relative to the grammar:

$$\begin{aligned} <S> &\rightarrow a <R> \mid b <S> b <R> \\ <R> &\rightarrow b <R> \mid a \end{aligned}$$

Head of sentential form	Remaining input
	bbaababa
b	baababa
bb	aababa
bba	ababa
bbaa	baba
bbaR	baba
bbS	baba
bbSb	aba
bbSba	ba
bbSbR	ba
bS	ba
bSb	a
bSba	
bSbR	
S	

   - Note that the parser depends on a stack to remember details of decisions it has already made (somewhat as in a top-down parser).
     - In a bottom up parser, the stack holds a prefix of a sentential form that the parser hopes the input will complete.
     - In a top down parser, the stack holds a suffix of a sentential form which the remaining input must match.
   - At each step a three part decision must be made:
     (a) Whether it is time to quit (i.e. either the parse is complete or an error has been encountered).
     (b) If we are to continue, should we shift more input symbols onto the stack or are we ready to replace the top of the stack with the left-hand side of some production.
     (c) If we are to reduce, which production should be used.
   - As a result, bottom-up parsers are also known as shift-reduce parsers.
   - Note that one can extract a right-most derivation by reversing the steps taken during a bottom-up parse.
   - To be "determinisitic", a bottom-up parser must make these decisions based on examining at most k symbols of the remaining

input.

2. One cannot always parse bottom-up by simply reducing as soon as one can match the right hands side of a production.
   - Consider

     If we begin parsing ‘bbaababa’ by reducing the first ‘a’ to obtain ’bbRababa’ we will get stuck because the string we obtain is not a sentential form of the language.
   - Consider the effects of ϵ productions.

3. Assuming that the string being considered is a sentential form, the correct string to reduce is the *handle* as defined by:

   simple phrase Given a grammar G and a string $w = \alpha\gamma\beta$ such that

   (a) $w, \alpha, \gamma, \beta \in (V_n \cup V_t)^*$,

   (b) for some $U \in V_n$, $U \rightarrow \gamma \in P$ and $\alpha U \beta$ is a sentential form of G

   we say that γ is a *simple phrase* of the sentential form w.

   handle The leftmost simple phrase of a sentential form is called the *handle*.

4. The problem one must solve to construct a bottom-up parser is to find rules for determining when one is looking at the handle.
   - These rules can only involve the current contents of the stack and a bounded number of input characters.

5. The class of grammars for which one can parse bottom-up deterministically (i.e. in without backup) looking ahead at most k characters at each step is called the class of LR(k) grammars.

Massaging Grammars

1. We have seen that for a given language some grammars may be harder to parse with than others (In fact, some may be impossible to parse with “deterministically”). Accordingly, one sometimes must apply techniques for rewriting a grammar into a more appropriate form.

2. We have already used on of these techniques known as left-factoring. Remember the problem I noted with using the standard rules for an if statement in a top down parser.

$$\begin{array}{rl} <\text{stmt}> \rightarrow & \text{if} <\text{expr}> \text{then} <\text{stmt}> \text{end} \\ | & \text{if} <\text{expr}> \text{then} <\text{stmt}> \text{else} <\text{stmt}> \text{end} \end{array}$$

   The problem can be remedied by “factoring” out the common prefix of these two productions giving the rules:

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

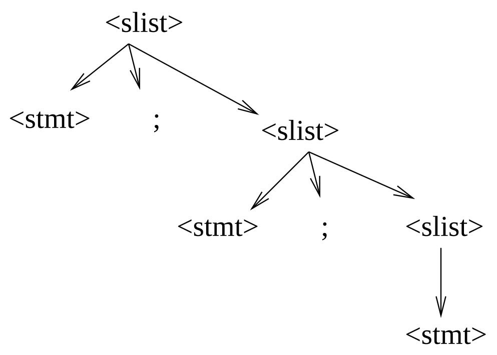
   Note, however, that the phrase structure one gets by parsing may no longer be quite what you had in mind.

3. Another common problem one may encounter in a grammar is a self-recursive rule.
   - A left recursive rule (i.e. of the form A $\rightarrow$ A α) is fatal in a top-down parser because if the parser chooses to apply such a rule once it will do so indefinitely .
   - A right recursive rule (i.e. of the form A $\rightarrow$ α A) is undesirable (i.e. I better not see any in your parsers) in a bottom up parser because it may lead to overflow in the parser’s stack.

4. Grammar alterations designed to avoid undesirable forms of recursion can seriously impact phrase structure.
   - Consider a simple grammar for statement lists:

$$\begin{array}{rl} <\text{slist}> \rightarrow & <\text{stmt}> \\ | & <\text{stmt}> ; <\text{slist}> \end{array}$$

   - The parse tree produced for a list of statement using this grammar would look like:



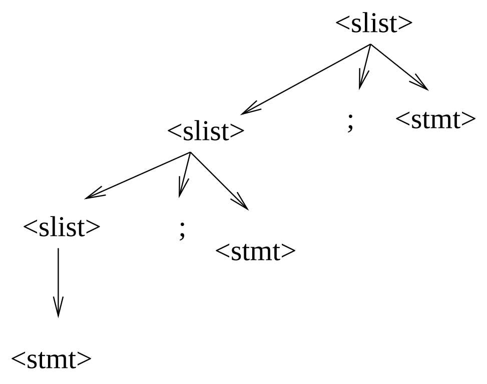
With the exception of the dangling semi-colons, this should look a lot like the syntax tree my parser builds for statement lists (and other list nodes).

- Unfortunately, A grammar of this form forces a bottom-up parser to shift all the statements onto the stack before reducing any of them to "slists". This wastes memory.
- It isn't difficult to fix this by switching to the grammar:

$$\begin{array}{rl} < \text{slist} > \rightarrow & < \text{stmt} > \\ | & < \text{slist} > ; < \text{stmt} > \end{array}$$

  This grammar:

  - describes the same language, and
  - can be parsed efficiently bottom-up.
- The trees produced by the alternate grammar would look more like:



If you think about how to process the statements in this tree in the order in which they would be executed, you will quickly realize that a simple loop can't do it.