CS 434 Meeting 14 — 4/3/06

Announcements
1. Phase 2.2 (code generation for control structures) should be finished by the end of the week.
2. About those code labels...
3. Phase 3 (Building a little parser with Yacc) will be assigned on Thursday.

LR Parsing

1. We have discussed how a shift-reduce parser works, now it is time to learn how to build one.

By way of review:

- As each input symbol is read, a shift-reduce parser either:
  - pushes the symbol onto a stack which represents a prefix of the sentential form the parser believes it is parsing, or
  - Pops the handle off of the top of the stack replacing all the symbols popped by the non-terminal on the left hand side of the rule used to perform the reduction.

2. In order to know when to shift and when to reduce, a bottom up parser must be able to determine when it has the handle of a sentential form sitting on top of its stack.

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

we say that $\gamma$ is a simple phrase of the sentential form $w$.

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

3. One possible approach to this task is to try to make sure that the contents of the stack are always some prefix of a sentential form that may include but does not extend past the handle. We will call such a prefix a viable prefix.

4. Given that a shift-reduce parser should eventually find a rightmost derivation for any valid input, we can restrict our attention to handles of sentential forms that are encountered in rightmost derivations.

5. To get a concrete sense of what such prefixes would look like, consider the following grammar:

$$
< E > \rightarrow < E > + < T > | < T > \\
< T > \rightarrow a | ( < E > )
$$

and sample rightmost derivation in which we have displayed the handle of each sentential form in italics:

$$
< E > \rightarrow < E > + < T > \\
\quad \rightarrow < E > + a \\
\quad \rightarrow < E > + < T > + a \\
\quad \rightarrow < E > + ( < E > + a ) + a \\
\quad \rightarrow < E > + ( < E > + < T > + a ) + a \\
\ldots
$$

Any prefix of any sentential form in such a derivation that does not extend past the handle should be considered a viable prefix.

(a) From the first step we would identify the following strings as viable prefixes:

$$
\epsilon < E > \\
< E > + \\
< E > + < T >
$$

(b) From the second step we would identify:

$$
\epsilon < E >
$$
We will demonstrate this by explaining how to build a finite state machine that recognizes the set of viable prefixes of a context free grammar.

9. Consider the problem of parsing strings using the following grammar:

\[
\begin{align*}
&S \rightarrow a \langle B \rangle | b \langle A \rangle | b \ c \\
&A \rightarrow b \\
&B \rightarrow b \ | \ c 
\end{align*}
\]

- In general, we can’t say whether a ‘b’ or ‘c’ that appears in the input is a handle or not.
- After reading a b, we know that if the following character is a ‘b’ it is the handle, but that if it is a ‘c’ the pair ‘bc’ forms the handle. We even know which production to use when we reduce.
- One way to explain how we know what to do after reading a ‘b’ is that after reading a ‘b’ we know that we are either “in between” the ‘b’ and the \( \langle A \rangle \) in the production \( \langle S \rangle \rightarrow b \langle A \rangle \) and therefore also possibly at the beginning of the production \( \langle A \rangle \rightarrow b \) or in between the ‘b’ and the ‘c’ in the production \( \langle S \rangle \rightarrow b \ c \).

10. Our approach to building LR(0) parsers will be based on a notation for describing “what point in a rule we are up to”. To be precise, we need the following definitions:

**LR(0) item** Given a grammar \( G \), we say that \( [N \rightarrow \beta_1, \beta_2] \) is an LR(0) item or LR(0) configuration for \( G \) if \( N \rightarrow \beta_1 \beta_2 \) is a production in \( G \).

**Configuration Set** We will refer to a set of LR(0) items as a configuration set.
For example, the configuration set:

\[
\begin{align*}
\langle S \rangle & \rightarrow \textbf{b} \cdot \langle A \rangle \\
\langle S \rangle & \rightarrow \textbf{b} \cdot \textbf{c} \\
\langle A \rangle & \rightarrow \textbf{b} \\
\end{align*}
\]

describes where we might be in various productions after reading a ‘b’ while parsing relative to the grammar discussed above.

11. Our intuition concerning how an LR(0) item describes “where we are” is made precise by the definition:

**Valid item** Given a grammar \( G \), we say that an LR(0) item, \([N \rightarrow \beta_1, \beta_2]\), is valid for \( \gamma \in (V_n \cup V_t)^* \) if there is a rightmost derivation

\[
S \xrightarrow{\epsilon} \alpha N \omega \xrightarrow{\text{rm}} \alpha \beta_1 \beta_2 \omega
\]

such that \( \alpha \beta_1 = \gamma \).

12. It should be clear that there is some connection between the definitions of valid items and viable prefixes. The connections are:

- If any LR(0) item is valid for a string \( \gamma \) then \( \gamma \) must be a viable prefix.
- If some string \( \gamma \) is a viable prefix, then there must be some LR(0) item that is valid for \( \gamma \).

13. Since a string is a viable prefix if and only if the set of LR(0) items for the string is non-empty, building a machine that keeps track of the set of valid LR(0) items as it reads input will enable us to identify viable prefixes.

- Once such a machine starts telling us there are no valid items we will know that we are no longer looking at a viable prefix we will know that we either have reached the end of the handle or hit an error.

14. Imagine what such a machine would look like for our trivial grammar:

\[
\begin{align*}
\langle S \rangle & \rightarrow \textbf{a} \langle B \rangle \mid \textbf{b} \langle A \rangle \mid \textbf{b} \textbf{c} \\
\langle A \rangle & \rightarrow \textbf{b} \\
\langle B \rangle & \rightarrow \textbf{b} \mid \textbf{c} \\
\end{align*}
\]

- The initial state would have to correspond to all LR(0) items valid for the null string:

\[
\begin{align*}
\langle S \rangle & \rightarrow \textbf{b} \cdot \langle A \rangle \\
\langle S \rangle & \rightarrow \textbf{b} \cdot \textbf{c} \\
\langle A \rangle & \rightarrow \textbf{b} \\
\langle B \rangle & \rightarrow \textbf{b} \\
\end{align*}
\]

- From this state, there should be a transition on input \textbf{a} to the state corresponding to the configuration set:

\[
\begin{align*}
\langle S \rangle & \rightarrow \textbf{a} \cdot \langle B \rangle \\
\langle B \rangle & \rightarrow \textbf{b} \\
\langle B \rangle & \rightarrow \textbf{c} \\
\end{align*}
\]

- and so on ...

**A Quick Review of Finite Automata**

1. To make all this precise (and eventually prove that it works) we may 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, \( \pi \).
   b. An input alphabet, \( \Sigma \).
   c. A transition function \( \delta : \pi \times \Sigma \rightarrow \pi \).
   d. A subset \( F \) of \( \pi \) called the set of final states.
   e. An element \( \pi_0 \) of \( \pi \) 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 \( \delta \) to strings over the input alphabet. In particular, we can define \( \Delta : \pi \times \Sigma^* \rightarrow \pi \) 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

\[ \{ \gamma \in \Sigma^* \mid \Delta(\pi_0, \gamma) \in F \} \]

**Constructing the LR(0) Machine for a Grammar**

1. 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
  \[ \text{goto}(\pi, x) = \{ [N \rightarrow \beta_1 x \beta_2] \mid [N \rightarrow \beta_1 \beta_2] \in \pi \} \]

- **closure** Given a set \( \pi \) of LR(0) items for a grammar \( G \) with productions \( P \), we define closure(\( \pi \)) to be the smallest set of LR(0) items such that:
  
  (a) closure(\( \pi \)) \( \supseteq \) \( \pi \)
  
  (b) if \( [N_1 \rightarrow \beta_1 N_2 \beta_2] \in \text{closure}(\pi) \) and \( N_2 \rightarrow \beta_3 \in P \) then \( [N_2 \rightarrow . \beta_3] \in \text{closure}(\pi) \)

2. The closure of a set of LR(0) items can be computed using a simple (but important) little algorithm

- An algorithm to compute closure(\( \pi \))
  
  (a) set \( \pi' \) equal to \( \pi \).
  
  (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 \( \pi' \).

3. 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 \( \pi \) 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 closure(\( \{ [S' \rightarrow . S \$] \} \))
- Let the transition function \( \delta : \pi \times (V_n \cup V_t) \rightarrow \pi \) be defined by:
  \[ \delta(\pi, x) = \text{closure}(\text{goto}(\pi, x)) \]

4. A somewhat interesting example.

\[ < S' > \rightarrow < S > \$ \]
\[ < S > \rightarrow < S > a < S > b | c \]