CS 434 Meeting 14 — 4/3/06

Anouncements

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 γ 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:

   $$< \text{E} > \rightarrow < \text{E} > + < \text{T} > \mid < \text{T} >$$
   $$< \text{T} > \rightarrow \text{a} \mid (< \text{E} >)$$

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

   $$\begin{aligned} < \text{E} > &\rightarrow < \textit{E} > + < \textit{T} > \\ &\rightarrow < \text{E} > + \textit{a} \\ &\rightarrow < \textit{E} > + < \textit{T} > + \text{a} \\ &\rightarrow < \text{E} > + \textit{(} < \textit{E} > \textit{)} + \text{a} \\ &\rightarrow < \text{E} > + (< \textit{E} > + < \textit{T} >) + \text{a} \\ &\rightarrow < \text{E} > + (< \text{E} > + \textit{a}) + \text{a} \\ &\ldots \end{aligned}$$

   Any prefix of any sentential from 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:

   $$\epsilon$$
   $$< \text{E} >$$
   $$< \text{E} > +$$
   $$< \text{E} > + < \text{T} >$$

   (b) From the second step we would identify:

   $$\epsilon$$
   $$< \text{E} >$$

$< \text{E} > +$
$< \text{E} > + \text{a}$

(c) Note that in this step, only the last item (which includes a part of the handle) is "new". This is true in general. So, for example, from the derivation step:

$$\rightarrow \ < \text{E} > + (< E > + < T > \ ) + \text{a}$$

we would only need to identify the following "new" viable prefixes:

$< \text{E} > + ($
$< \text{E} > + (< \text{E} >$
$< \text{E} > + (< \text{E} > +$
$< \text{E} > + (< \text{E} > + < \text{T} >$

6. We can turn these ideas into the following formal definition.

Viable prefix Given a grammar G, we say that $\gamma \in (V_n \cup V_t)^*$ is a *viable prefix* of G if there exists a rightmost derivation

$$S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N \omega \underset{\text{rm}}{\Longrightarrow} \alpha \beta_1 \beta_2 \omega$$

such that $\gamma = \alpha\beta_1$.

7. One way to understand the intuition behind the definition of a viable prefix is that something is a viable prefix of a sentential form it it extends up to but not past the handle..

   As long as the prefix of a sentential form of a shift-reduce parser is a viable prefix for the associated grammar, things are OK (i.e. we have not yet read past the handle and there is at least some possible remaining input that could form a valid sentential form and some hope of finding a rightmost parse of this sentential form).

8. It isn't clear that identifying viable prefixes is in any way simpler than the problem of parsing itself. Basically, given the definition above, one might not expect that the set (i.e. language) of viable prefixes associated with a context-free grammar is simpler than the language associated with the grammar. Luckily, it turns out that the set of viable prefixes associated with a context free grammar forms a regular language.

   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:

   $< \text{S} > \ \rightarrow \ \text{a} < \text{B} > \mid \text{b} < \text{A} > \mid \text{b c}$
   $< \text{A} > \ \rightarrow \ \text{b}$
   $< \text{B} > \ \rightarrow \ \text{b} \mid \text{c}$

   - 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 $< \text{A} >$ in the production

     $< \text{S} > \ \rightarrow \ \text{b} < \text{A} >$

     and therefore also possibly at the beginning of the production

     $< \text{A} > \ \rightarrow \ \text{b}$

     or in between the 'b' and the 'c' in the production

     $< \text{S} > \ \rightarrow \ \text{b c}$

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:

$< S > \rightarrow$ b . $< A >$
$< S > \rightarrow$ b . c
$< A > \rightarrow$. b

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

    $$S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N \omega \underset{\text{rm}}{\Longrightarrow} \alpha \beta_1 \beta_2 \omega$$

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

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:

    $< S > \rightarrow$ a $< B >$ | b $< A >$ | b c
    $< A > \rightarrow$ b
    $< B > \rightarrow$ b | c

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

      [$< S > \rightarrow$. a $< B >$]
      [$< S > \rightarrow$. b $< A >$]
      [$< S > \rightarrow$. b c]

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

      [$< S > \rightarrow$ a . $< B >$]
      [$< B > \rightarrow$. b]
      [$< B > \rightarrow$. c]

    - 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, π.
   (b) An input alphabet, Σ.
   (c) A transition function $\delta : \pi \text{ x } \Sigma \rightarrow \pi$.
   (d) A subset F of π called the set of final states.
   (e) An element π_0 of π 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 δ to strings over the input alphabet. In particular, we can define $\Delta : \pi \text{ x } \Sigma^* \rightarrow \pi$ 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

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

   closure Given a set π of LR(0) items for a grammar G with productions P, we define closure(π) to be the smallest set of LR(0) items such that:

   (a) closure(π) $\supseteq \pi$

   (b) if $[N_1 \rightarrow \beta_1 . N_2 \beta_2] \in$ closure(π) and $N_2 \rightarrow \beta_3 \in P$ then $[N_2 \rightarrow . \beta_3] \in$ closure(π)

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(π)

     (a) set π' equal to π.

     (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 π'.

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 π 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 $\delta : \pi$ x $(V_n \cup V_t) \rightarrow \pi$ be defined by:

     $$\delta(\pi, x) = closure(goto(\pi, x))$$

4. A somewhat interesting example.

   < S‘ > → < S > \$
   < S > → < S > a < S > b | c