CS 434 Meeting 20 — 4/25/06

SLR(1) parsing

1. Suppose that we find that after reading some prefix ω_1 of an input $\omega_1 x \omega_2$ we end up in a state that contains a reduce item $[N \rightarrow \beta.]$ which conflicts with some other item.
   - If we decide to reduce using the production in this item, we are basically assuming that
   $$S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N x \omega_2 \underset{\text{rm}}{\Longrightarrow} \alpha \beta x \omega_2 \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \omega_1 x \omega_2$$
   - That is, we are assuming that there is some sentential form in which an x can follow an N.
2. We can give the set of symbols that might appear after a non-terminal a name:
   $$\text{Follow}(N) = \{x \in V_t \mid A \overset{*}{\Longrightarrow} \alpha N x \beta\} \cup \{\epsilon \text{ if } S \overset{*}{\Longrightarrow} \alpha N\}$$
3. Given the notion of the "follow" set, we can illustrate the use of look-ahead in LR parsing, by considering the simplest form of look-ahead LR parsing — SLR(1) parsing (that's S for simple).
4. In general, we will say that a set of LR(0) items contains an SLR(1) conflict if either:
   (a) It contains two reduce items $[N \rightarrow \beta_1.]$ and $[M \rightarrow \beta_2.]$ such that the intersection of Follow(N) and Follow(M) is non-empty, or
   (b) It contains a reduce item $[N \rightarrow \beta_1.]$ and a shift item $[M \rightarrow \beta_2 . x \beta_2]$ such that $x \in \text{Follow}(N)$.
5. If the LR(0) machine for a grammar G contains no states with SLR(1) conflicts we say that G is an SLR(1) grammar.
6. An SLR(1) parser for an SLR(1) grammar G behaves as follows in state π when the next input symbol is "x":
   - reduce using production $N \rightarrow \beta$ if $[\ N \rightarrow \beta\ .\ ]\ \in \pi$, and $x \in \text{Follow}(N)$.
   - shift in next input if π contains one or more items of the form $[\ N \rightarrow \alpha . x \beta\ ]$.
   - error otherwise.

Computing Follow(N)

1. If a grammar contains the rule $N \rightarrow \alpha_1 M R \alpha_2$, then Follow(M) must contain the first symbols that might appear in strings derived from R. In addition, if R derives the empty string, then the first symbols that might be derived from α_2 will be in Follow(M).
2. The computation of the Follow set will depend on several other sub-computations.

   nullable Given a grammar G, we say that a non-terminal N is *nullable* if $N \overset{*}{\Longrightarrow} \epsilon$.

   first set Given a grammar G and $\alpha \in (V_n \cup V_t)^*$ we define First(α) to be the set of terminals that might appear as the first symbol in a string derived from α. First(α) will include ϵ if $\alpha \overset{*}{\Longrightarrow} \epsilon$. Thus,
   $$\text{First}(\alpha) = \{a \in V_t \mid \alpha \overset{*}{\Longrightarrow} a\beta, \text{for some} \beta \in (V_n \cup V_t)^*\} \cup \{\epsilon \text{ if } \alpha \overset{*}{\Longrightarrow} \epsilon\}$$
3. The set of nullable non-terminals can be computed by the following algorithm:
   (a) Set "nullable" equal to the set of non-terminals appearing on the left side of productions of the form $N \rightarrow \epsilon$.
   (b) Until doing so adds no new non-terminals to "nullable", examine each production in the grammar adding to "nullable" all left-hand-sides of productions whose right-hand-side consist entirely of symbols in "nullable".
4. Given that we have computed the set of nullable non-terminal, we can compute First for each terminal and non-terminal using a "run until nothing changes" algorithm:
   - We use a table called 'FirstSet' with one entry for each terminal and non-terminal. Throughout the algorithm, for any $x \in (V_t \cup V_n)$
   $$\text{First}(x) \supseteq \text{FirstSet}[x]$$

- FirstSet will be initialized as follows.
  (a) Set FirstSet[x] to {ϵ} for all nullable non-terminals and to {} for all other non-terminals.
  (b) Set FirstSet[x] equal to {x} for all terminals.
  (c) For each production of the form $N \rightarrow t\beta$ add t to FirstSet[N].
- For each production of the form $N \rightarrow \beta$, write β as $\beta = \alpha\beta'$ where α is a string of nullable non-terminals, and β' is either the ϵ or a string of terminals and non-terminals beginning with a non-nullable symbol we will call M.
- The approximations for the First sets stored in the FirstSet table are then improved until they become exact by repeating the following process until no further changes occur.
  – For each production $N \rightarrow \alpha\beta'$
    (a) For each $x \in \alpha$, add (FirstSet[x] $- \epsilon$) to FirstSet[N].
    (b) If $\beta' = \epsilon$ then add ϵ to FirstSet[N] otherwise add (FirstSet[M] $- \epsilon$) to FirstSet[N].

5. Consider how to determine nullable, First and Follow for the grammar:

S → A B C
A → a | CB
B → C | A d | ϵ
C → f | ϵ

- All the non-terminals are nullable:
  – B and C are directly nullable.
  – The production $A \rightarrow BC$, then implies A is nullable.
  – Then, the production $S \rightarrow ABC$ implies S is nullable.
- The productions break down as:

	α	β
S →	A B C	
A →		a
A →	CB	
B →	C	
B →	A	d
B →		ϵ
C →		f
C →		ϵ

- The FirstSet values start out as shown in the table below and can be refined by iterating over the production table.

S	A	B	C	a	d	f
{ ϵ }	{ ϵ, a }	{ ϵ }	{ ϵ, f }	{ a }	{ d }	{ f }
{ }	{ }	{ }	{ }	{ a }	{ d }	{ f }
{ }	{ }	{ }	{ }	{ a }	{ d }	{ f }
{ }	{ }	{ }	{ }	{ a }	{ d }	{ f }

6. The computation of Follow(N) depends of the First sets defined earlier. If

$$M \rightarrow \alpha N \beta$$

then Follow(N) must contains First(β) . If β is nullable, then Follow(N) must also contain Follow(M). These observations are enough to give us an approximation algorithm for computing Follow. See the books on reserve for details.

LR(1) Parsing

1. Simply using Follow sets to interpret look ahead symbols may give less information than is really available.
   - Consider the grammar:
     E → (L , E)
     E → S
     L → L , E
     L → E
     S → ident
     S → (S)

The state reached on input '(S' contains an SLR(1) conflict but 1 symbol look ahead is enough to allow us to parse.

To see why, build the LR(0) machine. The conflict is between the items $[S \rightarrow (S.)]$ and $[E \rightarrow S.]$ and ")" is clearly in the Follow set of E. However, if one reduces using the production in the reduce item, one would quickly ends up in a state where you further reduce the E to an L. Then, you end up in a state where the only possible action is to shift in a comma. So, if the next input is not a comma, reducing by $E \rightarrow S$ is a dead end.

2. LR(1) parsing is a generalization of LR(0) parsing that keeps track of both what points in what productions we might be up to and what might follow the rhs's of the productions we are working on.

3. We start with plenty of new (but familiar) definitions.

   LR(1) item Given a grammar G, we say that $[N \rightarrow \beta_1.\beta_2, a]$ is an *LR(1) item* or *LR(1) configuration* for G if $N \rightarrow \beta_1\beta_2$ is a production in G and $a \in (V_t \cup \epsilon)$. The symbol 'a' is called the *lookahead*.

   Configuration Set We will refer to a set of LR(1) items as an LR(1) *configuration set*.

   Valid item Given a grammar G, we say that an LR(1) item $[N \rightarrow \beta_1.\beta_2, a]$ 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 $\alpha\beta_1 = \gamma$ and $a \in \text{First}(\omega)$.

Building an LR(1) Machine

1. First, we need to extend the definitions which we used to define the transition function for an LR(0) machine to account for the lookaheads we have added to LR(1) items.

   goto Given a set of LR(1) items for a grammar G, we define

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

   closure Given a set π of LR(1) items for a grammar G with productions P, we define closure(π) to be the smallest set of LR(1) items such that:

   (a) closure(π) $\supseteq \pi$

   (b) if $[N_1 \rightarrow \beta_1.N_2\beta_2, a] \in$ closure(π) and $N_2 \rightarrow \beta_3 \in P$ then, for each $b \in \text{First}(\beta_2 a)$, $[N_2 \rightarrow .\ \beta_3, b] \in$ closure(π)

2. With these definitions, it should be obvious, that the next step is to define the LR(1) finite automaton for a grammar G consisting of:

   - A set of states with one state for every subset of LR(1) items.
   - An alphabet consisting of the terminals and non-terminals of G.
   - A set of final states consisting of the set of all states except the state corresponding to the empty set of LR(1) items.
   - A transition function defined by:

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

   - The state closure($[S' \rightarrow .S\$, \epsilon]$) as its initial state.

3. The notions of a kernel item and a reduce item transfer naturally from LR(0) items to LR(1) items.

4. Note that the language accepted by the LR(1) FSM is the same as that accepted by the LR(0) machine (i.e. the set of viable prefixes). The extra states in the machine, however, include information that can be used to make a better parser.

5. Consider what happens when we build the LR(1) machine for the non-SLR(1) grammar considered earlier.

   E → (L , E)
   E → S
   L → L , E
   L → E
   S → ident
   S → (S)

6. A set of LR(1) items contains a conflict if it contains a reduce items of the form $[N \rightarrow \beta_1 ., x]$ and either another reduce item of the form $[M \rightarrow \beta_2 ., x]$ or a shift item of the form $[M \rightarrow \alpha . x \beta_2, y]$.

7. We say that a grammar is LR(1) if the reachable states in its LR(1) machine are conflict free.

8. Given an LR(1) grammar, its LR(1) parser, shifts in state π with input x if state π contains a shift item of the form $[N \rightarrow \alpha . x \beta, a]$, reduces using production $N \rightarrow \beta$ if state π contains a reduce item of the form $[N \rightarrow \beta ., x]$ and reports error otherwise.