03/24/2010

Recitation #9

Administrative

- Assignment 4 is due next Monday (03/28 @ 11:59 p.m.)

Bottom-Up Parser

- **Bottom-up Parser**, i.e. we start with the terminals in the input string and subsequently compute recognized parse trees by going from already recognized rhs of productions to the non-terminal on the lefthand side
    - In a way it's a flipped parse tree: beginning at the leaves (terminals) and working up towards the root
    - Most prevalent type is based on concept of LR(k) parsers; "L" stands for left-to-right scanning, "R" constructs rightmost derivation in reverse, "k" is the number of input symbols of lookahead → usually deal with LR(1)
- **Reductions**
    - Reduction is the reverse of a production
    - Then bottom-up parsing can be thought of as "reducing" the input string to the start variable
    - Example expression "id * id":
      $\mathrm{id} * \mathrm{id},\ F * \mathrm{id},\ T * \mathrm{id},\ T * F,\ T,\ E$
    - It builds a right derivation in reverse:
      $E \to T \to T * F \to T * \mathrm{id} \to F * \mathrm{id} \to \mathrm{id} * \mathrm{id}$
    - How do we decide which reduction to apply? → "**Handle Pruning**"
        - A handle is a substring that matches the body of a production AND whose reduction represents one step along the rightmost derivation reverse
        - The substring to the right of the handle only contains non-terminals!
        - Not all leftmost substrings that match the body of some production are handles; only when they are part of the rightmost derivation
        - By keep replacing current handles and executing reductions, we "prune" our way to the start symbol

Example Expression Grammar:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathrm{id}$$

- **Shift-Reduce Parsing**
    - $ marks the bottom of the stack and the right side of the input
    - Top of the stack is to the right of $; **handles always appear on top of stack**
    - Four possible actions:
        - *Shift*. Shift the next input symbol onto the top of the stack
        - *Reduce*. Reduce a on the top of the stack to its non-terminal
        - *Accept*. Successful completion of parsing if $S on stack
        - *Error*. Discover syntax error and call error recovery.

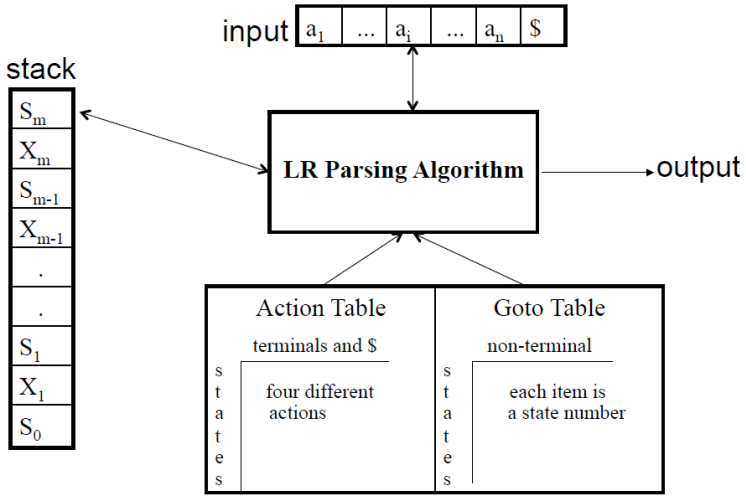
- Example: String $\mathrm{id}_1 + \mathrm{id}_2 * \mathrm{id}_3$

	Stack	Input	Handle	Action
1	$\$$	$\mathrm{id}_1 + \mathrm{id}_2 * \mathrm{id}_3 \$$	None	Shift
2	$\$\mathrm{id}_1$	$+\mathrm{id}_2 * \mathrm{id}_3 \$$	id_1	Reduce by $F \to \mathrm{id}$
3	$\$F$	$+\mathrm{id}_2 * \mathrm{id}_3 \$$	F	Reduce by $T \to F$
4	$\$T$	$+\mathrm{id}_2 * \mathrm{id}_3 \$$	T	Reduce by $E \to T$
5	$\$E$	$+\mathrm{id}_2 * \mathrm{id}_3 \$$	None	Shift
6	$\$E+$	$\mathrm{id}_2 * \mathrm{id}_3 \$$	None	Shift
7	$\$E+\mathrm{id}_2$	$*\mathrm{id}_3 \$$	id_2	Reduce by $F \to \mathrm{id}$
8	$\$E+F$	$*\mathrm{id}_3 \$$	F	Reduce by $T \to F$
9	$\$E+T$	$*\mathrm{id}_3 \$$	None	Shift
10	$\$E+T*$	$\mathrm{id}_3 \$$	None	Shift
11	$\$E+T*\mathrm{id}_3$	$\$$	id_3	Reduce by $F \to \mathrm{id}$
12	$\$E+T*F$	$\$$	$T*F$	Reduce by $T \to T*F$
13	$\$E+T$	$\$$	$E+T$	Reduce by $E \to E+T$
14	$\$E$	$\$$	None	**DONE**

  - Some notes on this table:
    - In line 9, T is on the right side of the stack, but is not a handle, even though there is a rule $E \to T$. It's not a handle because if the reduction would be executed, $E+E$ wouldn't get us anywhere. Also, $E+T$ is not a handle because replacing it with E brings us into trouble later
    - Look at lines 9 and 13; the stack looks the same, but in one you shift, in the other you reduce → Make that decision based on looking at the next input symbol

  - The key here is finding the handle
    - We know that handle is on stack and that there are finite number of handles → build DFA
    - LR(1) parsers build a DFA that runs over the stack and finds them, based on one symbol lookahead on input

- **LR parsers** are table-driven (much like LL(1) parsing table)
  - Can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written
  - No backtracking required
  - Detects a syntactic error as soon as it is possible to do so on a left-to-right scan of input
  - All LL(1) grammars are proper subset of LR(1) grammars



SLR Parser

- **Terminology**
  - $LR(0)$ *item*: of grammar G is a production of G with a dot at some position of the body, e.g. $A \to \varepsilon$ yields $A \to \bullet$ and $A \to aBb$ yields:
    $A \to \bullet aBb \quad A \to a \bullet Bb \quad A \to aB \bullet b \quad A \to aBb \bullet$
    (intuitively, $A \to a \bullet Bb$ indicates that we have just seen on the input string a and hope next to see a string derivable from Bb; $A \to aBb \bullet$ indicates we have seen aBb and it may be time to reduce)
  - *Canonical* $LR(0)$ *collection*: Sets of LR(0) items that provide basis for building DFA and make parsing decision → LR(0) automaton has one state for each item set in canonical LR(0) collection.
  - *Augmented Grammar*: If G is grammar with start symbol S, then the augmented grammar G' with a new start symbol S' and production $S' \to S$

- **Closure of Item Sets**
  - I is set of items
  - CLOSURE(I) can be computed as follows:
    - Add every item in I to CLOSURE(I)
    - If $A \to \alpha \bullet B\beta$ in CLOSURE(I) and $B \to \gamma$ is a production, then add $B \to \bullet\gamma$ to CLOSURE(I).
    - Apply (2) until no more new items can be added to CLOSURE(I)
  - Intuition: $A \to \alpha \bullet B\beta$ indicates that we might next see a substring derivable from $B\beta$. Prefix will derive from B, that's why we add $B \to \bullet\gamma$.
  - In particular we are interested in CLOSURE($\{S' \to \bullet S\}$)
  - Closure lists all handles that might trigger reduction operation

- Kernel Items: initial item $S' \rightarrow \bullet S$ and all items whose dots are not at the left end
- Nonkernel Items: All items with their dots at the left end, except for $S' \rightarrow \bullet S$. → Nonkernel items don't need to be saved explicitly as they can be reconstructed by closure operation on kernel items

- **GOTO function**
  - GOTO(I, X), where I is set of items and X is grammar symbol
  - GOTO(I, X) is the closure of the set of all items $A \rightarrow \alpha X \bullet \beta$ such that $A \rightarrow \alpha \bullet X \beta$ is in I
  - Defines transitions in the LR(0) automaton; states of automaton are sets of items and GOTO(I, X) specifies transition from state I under input X

- **Canonical LR(0) Collection Construction**
  - Construct set from augmented grammar G'
  - This constructs states of LR(0) automaton
  - How do these states help us parse?
    - If currently in state j and we read input a
      - Shift, if there is a transition in the automaton from state j on input a
      - Items in state j tell us how to reduce otherwise
  - Start state is CLOSURE($\{S' \rightarrow \bullet S\}$)
  - All states are accepting states in DFA
  - Algorithm:

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

- **Building SLR-Parsing table**
  - First, build the FIRST and FOLLOW sets for the grammar
  - Then follow algorithm in book pages 325 – 326

- **Worksheet to build SLR parser for simple grammar**