Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing Part - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\},\ \ \delta(q_1, 0, 0) = \{(q_1, 00)\},$
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\},\ \ \delta(q_2, 1, 0) = \{(q_2, \epsilon)\},$
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

PDA - Examples (contd.)

- $L = \{ww^R \mid w \in \{a, b\}^+\}$
  $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, where δ is defined as follows
  $\delta(q_0, a, Z) = \{(q_0, aZ)\},\ \delta(q_0, b, Z) = \{(q_0, bZ)\},$
  $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \epsilon)\},\ \delta(q_0, a, b) = \{(q_0, ab)\},$
  $\delta(q_0, b, a) = \{(q_0, ba)\}, \delta(q_0, b, b) = \{(q_0, bb), (q_1, \epsilon)\},$
  $\delta(q_1, a, a) = \{(q_1, \epsilon)\},\ \ \delta(q_1, b, b) = \{(q_1, \epsilon)\},$
  $\delta(q_1, \epsilon, Z) = \{(q_2, \epsilon)\}$
- $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash (q_0, ba, baZ) \vdash (q_1, a, aZ) \vdash (q_1, \epsilon, Z) \vdash (q_2, \epsilon, \epsilon)$
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_0, a, aaZ) \vdash (q_1, \epsilon, aZ) \vdash$ *error*
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_1, a, Z) \vdash$ *error*

Nondeterministic and Deterministic PDA

- Just as in the case of NFA and DFA, PDA also have two versions: NPDA and DPDA
- However, NPDA are strictly more powerful than the DPDA
- For example, the language, $L = \{ww^R \mid w \in \{a, b\}^+\}$ can be recognized only by an NPDA and not by any DPDA
- In the same breath, the language, $L = \{wcw^R \mid w \in \{a, b\}^+\}$, can be recognized by a DPDA
- In practice we need DPDA, since they have exactly one possible move at any instant
- Our parsers are all DPDA

Parsing

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar
- If there are no restrictions on the language and the form of grammar used, parsers for context-free languages require $O(n^3)$ time (n being the length of the string parsed)
  - Cocke-Younger-Kasami's algorithm
  - Earley's algorithm
- Subsets of context-free languages typically require $O(n)$ time
  - Predictive parsing using $LL(1)$ grammars (top-down parsing method)
  - Shift-Reduce parsing using $LR(1)$ grammars (bottom-up parsing method)

Top-Down Parsing using LL Grammars

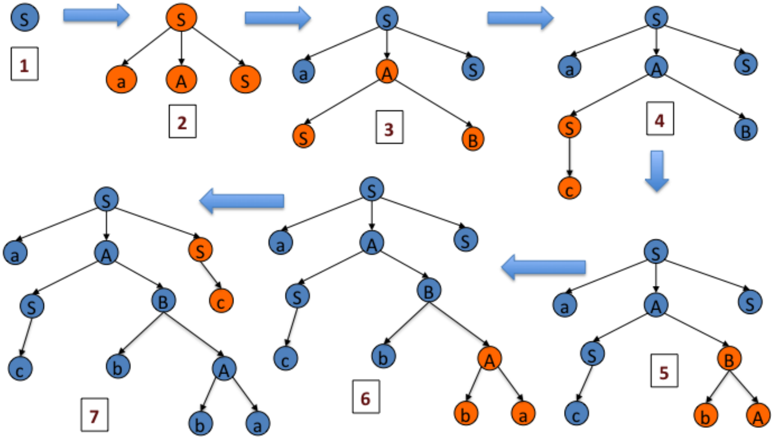
- Top-down parsing using predictive parsing, traces the left-most derivation of the string while constructing the parse tree
- Starts from the start symbol of the grammar, and “predicts” the next production used in the derivation
- Such “prediction” is aided by parsing tables (constructed off-line)
- The next production to be used in the derivation is determined using the next input symbol to lookup the parsing table (look-ahead symbol)
- Placing restrictions on the grammar ensures that no slot in the parsing table contains more than one production
- At the time of parsing table constrcution, if two productions become eligible to be placed in the same slot of the parsing table, the grammar is declared unfit for predictive parsing

Top-Down LL-Parsing Example

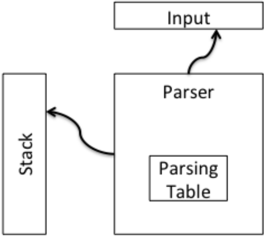
S → aAS | c
A → ba | SB
B → bA | S

Leftmost derivation of the string *acbbac*

S => aAS => aSBS => acBS => acbAS => acbbaS => acbbac
1 2 3 4 5 6 7



LL(1) Parsing Algorithm




```
Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
   let X be the top stack symbol;
   let a be the next input symbol /*may be $*/;
   if X is a terminal symbol or $ then
       if X == a then {
            pop X from Stack;
            remove a from input;
       } else ERROR();
   else  /* X is a non-terminal symbol */
        if M[X,a] == X → Y1Y2... Yk then {
              pop X from Stack;
              push Yk, Yk-1, ..., Y1 onto Stack;
                     (Y1 on top)
        }
} until Stack has emptied;
```

LL(1) Parsing Algorithm Example

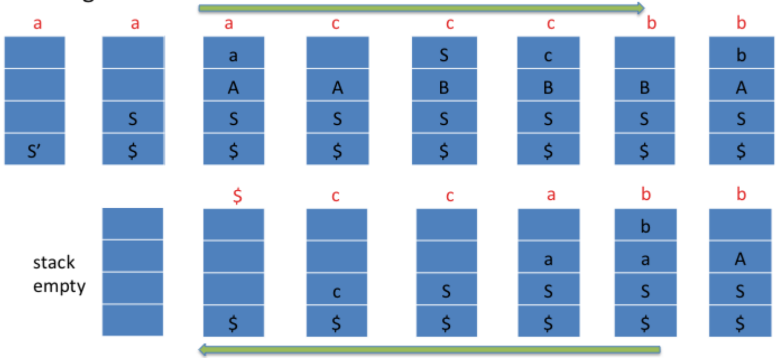
Grammar

S' → S$
S → aAS | c
A → ba | SB
B → bA | S

string: *acbbac*

LL(1) Parsing Table

	a	b	c	$
S'	S' → S$		S' → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S	



Strong LL(k) Grammars

Let the given grammar be G

- Input is extended with k symbols, $\$^k$, k is the lookahead of the grammar
- Introduce a new nonterminal S', and a production, $S' \rightarrow S\$^k$, where S is the start symbol of the given grammar
- Consider leftmost derivations only and assume that the grammar has no *useless symbols*
- A production $A \rightarrow \alpha$ in G is called a *strong LL*(k) production, if in G
  $S' \Rightarrow^* wA\gamma \Rightarrow w\alpha\gamma \Rightarrow^* wzy$
  $S' \Rightarrow^* w'A\delta \Rightarrow w'\beta\delta \Rightarrow^* w'zx$
  $|z| = k,\ z \in \Sigma^*, w \text{ and } w' \in \Sigma^*$, then $\alpha = \beta$
- A grammar (nonterminal) is strong LL(k) if all its productions are strong LL(k)

Strong LL(k) Grammars (contd.)

- Strong LL(k) grammars do not allow different productions of the same nonterminal to be used even in two different derivations, if the first k symbols of the strings produced by $\alpha\gamma$ and $\beta\delta$ are the same
- Example: $S \rightarrow Abc|aAcb,\ \ A \rightarrow \epsilon|b|c$
  S is a strong LL(1) nonterminal
  - $S' \Rightarrow S\$ \Rightarrow Abc\$ \Rightarrow bc\$,\ bbc\$$, and $cbc\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = b,\ b$, or c, respectively
  - $S' \Rightarrow S\$ \Rightarrow aAcb\$ \Rightarrow acb\$,\ abcb\$$, and $accb\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = a$, in all three cases
  - In this case, $w = w' = \epsilon$, $\alpha = Abc$, $\beta = aAcb$, but z is different in the two derivations, in all the derived strings
  - Hence the nonterminal S is strong LL(1)

Strong LL(k) Grammars (contd.)

A is not strong LL(1)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}c\$$, $w = \epsilon$, $z = b$, $\alpha = \epsilon$ ($A \rightarrow \epsilon$)
  $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}bc\$$, $w' = \epsilon$, $z = b$, $\beta = b$ ($A \rightarrow b$)
- Even though the lookaheads are the same ($z = b$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(1)

A is not strong LL(2)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{bc}\$$, $w = \epsilon$, $z = bc$, $\alpha = \epsilon$ ($A \rightarrow \epsilon$)
  $S' \Rightarrow^* aAcb\$ \Rightarrow a\underline{bc}b\$$, $w' = a$, $z = bc$, $\beta = b$ ($A \rightarrow b$)
- Even though the lookaheads are the same ($z = bc$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(2)

A is strong LL(3) because all the six strings (*bc\$, bbc, cbc, cb\$, bcb, ccb*) can be distinguished using 3-symbol lookahead (details are for home work)

Testable Conditions for LL(1)

- We call strong LL(1) as LL(1) from now on and we will not consider lookaheads longer than 1
- The classical condition for LL(1) property uses *FIRST* and *FOLLOW* sets
- If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
  $FIRST(\alpha) = \{a \mid a \in T,\ and\ \alpha \Rightarrow^* ax,\ x \in T^*\}$
  $FIRST(\epsilon) = \{\epsilon\}$
- If A is any nonterminal, then
  $FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*,$
  $a \in T \cup \{\$\}\}$
- $FIRST(\alpha)$ is determined by α alone, but $FOLLOW(A)$ is determined by the "context" of A, i.e., the derivations in which A occurs

FIRST and *FOLLOW* Computation Example

- Consider the following grammar
  $S' \to S\$, \; S \to aAS \mid c, \; A \to ba \mid SB, \; B \to bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$ because
  $S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$
- $FIRST(A) = \{a, b, c\}$ because
  $A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$ because
  $S' \Rightarrow \underline{S\$}$,
  $S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
  $S' \Rightarrow^* a\underline{S}BS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
  $S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$
- $FOLLOW(A) = \{a, c\}$ because
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Computation of *FIRST*: Terminals and Nonterminals

```
{
 for each (a ∈ T) FIRST(a) = {a}; FIRST(ϵ) = {ϵ};
 for each (A ∈ N) FIRST(A) = ∅;
 while (FIRST sets are still changing) {
     for each production p {
        Let p be the production A → X1X2...Xn;
        FIRST(A) = FIRST(A) ∪ (FIRST(X1) - {ϵ});
        i = 1;
        while (ϵ ∈ FIRST(Xi) && i ≤ n − 1) {
            FIRST(A) = FIRST(A) ∪ (FIRST(Xi+1 − {ϵ}); i++;
        }
        if (i == n) && (ϵ ∈ FIRST(Xn))
           FIRST(A) = FIRST(A) ∪{ϵ}
 }
}
```