RECURSIVE DESCENT PARSER

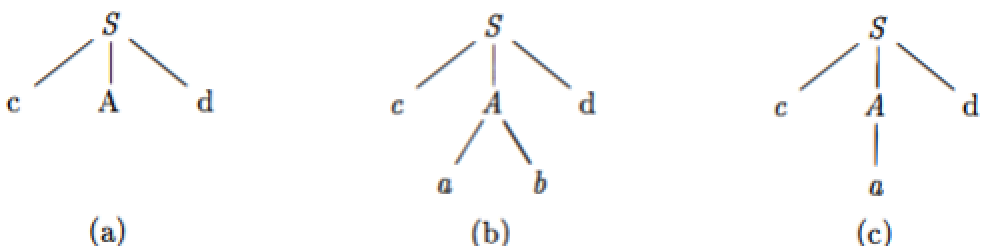
- These parsers use a procedure for each nonterminal. The procedure looks at its input and decides which production to apply for its nonterminal.
- Terminals in the body of the production are matched to the input at the appropriate time, while nonterminals in the body result in calls to their procedure.
- General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs.

Example : Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

To construct a parse tree top-down for the input string w = cad



Step 1:

- Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.

An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S

Step 2:

- The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’.

Step 3:

- The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’. But the third leaf of tree is b which does not match with the input symbol d.

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

FIRST( ) & FOLLOW( ):

- First(α) is defined as set of terminals that begins strings derived from α.
- If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ε is also in First(α)
- In predictive parsing when we have A-> α|β, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input.
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form.

Algorithm to compute FIRST(X)

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a nonterminal and X → $Y_1Y_2...Y_k$ is a production for some k≥1, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and **ε** is in all of FIRST(Y_1),..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. (If **ε** is in FIRST($Y_j$) for all j = 1,2, . . . , k, then add **ε** to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_l does not derive **ε**, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add F1RST($Y_2$), and So on.
3. If X→ **ε** is a production, then add **ε** to FIRST(X).

Algorithm to compute FOLLOW(A)

To compute **FOLLOW(A)** for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.
2. If there is a production A→αBβ, then everything in FIRST(β) except **ε** is in FOLLOW(B).
3. If there is a production A→αB, or a production A→αBβ, where FIRST(β) contains **ε**, then everything in FOLLOW (A) is in FOLLOW (B).

Example : To compute FIRST and FOLLOW

Consider again the non-left-recursive grammar

E →TE'

E' →+ T E' | ε

T →F T'

T' →* F T' | ε

F →(E) | id

- FIRST(F) = FIRST(T) = FIRST(E) = {(, id}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, id and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ε, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).
- FIRST(E') = {+, ε }. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is ε. Whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal.
- FIRST(T') = {* , t}. The reasoning is analogous to that for FIRST(E').
- FOLLOW(E) = FOLLOW(E') = {), $}. Since E is the start symbol, FOLLOW(E) must contain $. The production body (E) explains why the right parenthesis is in FOLLOW(E) . For E' , note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).
- FOLLOW(T) = FOLLOW(T') = {+,), $}. Notice that T appears in bodies only followed by E' . Thus, everything except ε that is in FIRST(E') must be in FOLLOW(T) ; that explains the symbol +.

PREDICTIVE PARSER – LL(1) PARSER

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

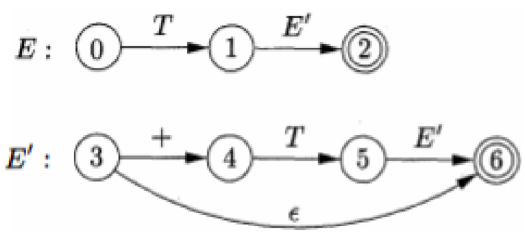
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.
- This parser looks up the production in parsing table.
- The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

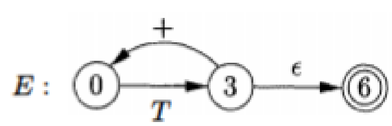
Transition Diagrams for Predictive Parsers:

- Transition diagrams are useful for visualizing predictive parsers.
- For example, the transition diagrams for nonterminals E and E' of expression grammar

E → TE'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

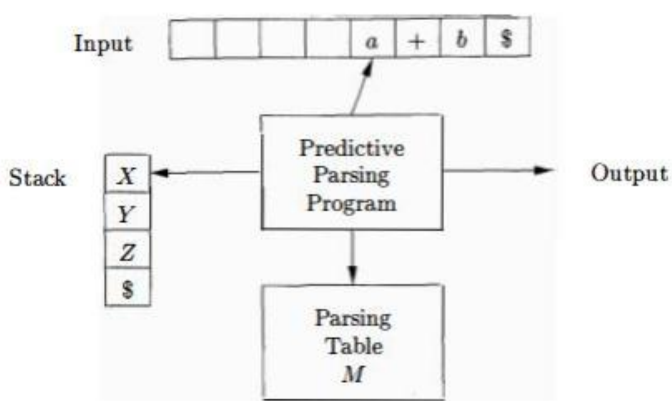
- To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar.
- Transition diagrams for predictive parsers differ from those for lexical analyzers.
- Parsers have one diagram for each nonterminal.
- The labels of edges can be tokens or nonterminals.
- A transition on a token (terminal) means that we take that transition if that token is the next input symbol.





Predictive Parsing Program:

The predictive parser has an input, a stack, a parsing Table and an output.



Input: Contains the string to be parsed, followed by right end marker $.
Stack: Contains a sequence of grammar symbols, preceded by $, the bottom of stack marker. Initially the stack contains the start symbol of the grammar preceded by $.
Parsing Table: It is a two dimensional array M[A,a], where A is a non-terminal and a is a terminal or $.
Output: Gives the output whether the string is valid or not.