Top-Down Parsing

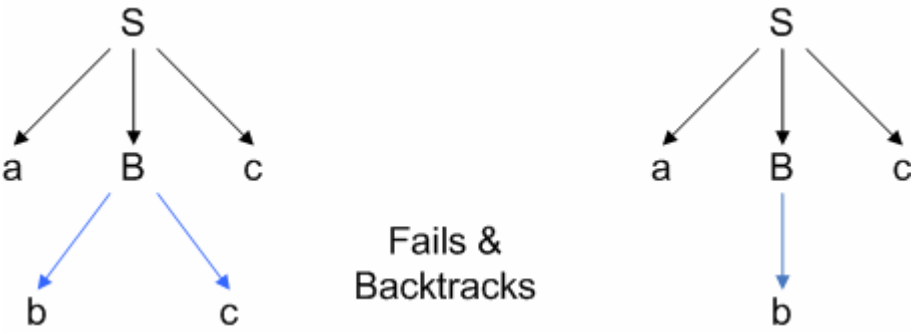
- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

S → aBc
B → bc | b

input: abc



Predictive Parser

A grammar	➔	➔	a grammar suitable for predictive
	eliminate	left	parsing (a LL(1) grammar)
	left recursion	factor	no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

Predictive Parser (example)

stmt → `if` ..... |
`while` ..... |
`begin` ..... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Example:
A → aBb (This is only the production rule for A)

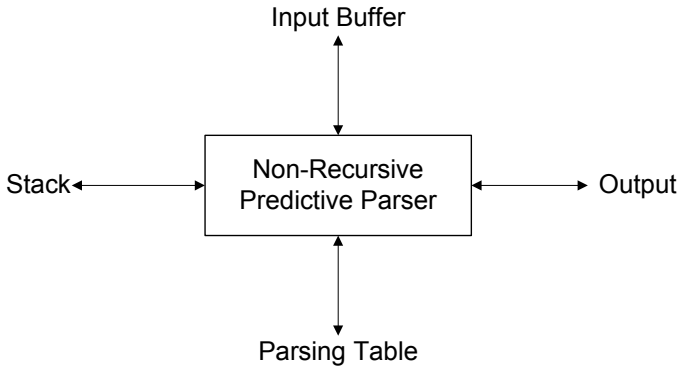
```
proc A {
    - match the current token with a, and move to the next token;
    - call 'B';
    - match the current token with b, and move to the next token;
}
```

Another Ex:
A → aBb | bAB

```
proc A {
  case of the current token {
      'a':  - match the current token with a, and move to the next token;
            - call 'B';
            - match the current token with b, and move to the next token;
      'b':  - match the current token with b, and move to the next token;
            - call 'A';
            - call 'B';
  }
}
```

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

- If X and a are $ ➔ parser halts (successful completion)

- If X and a are the same terminal symbol (different from $)
  ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

- If X is a non-terminal
  ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 ... Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 ... Y_k$ to represent a step of the derivation.

- none of the above ➔ error

LL(1) Parser – Example1

$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$

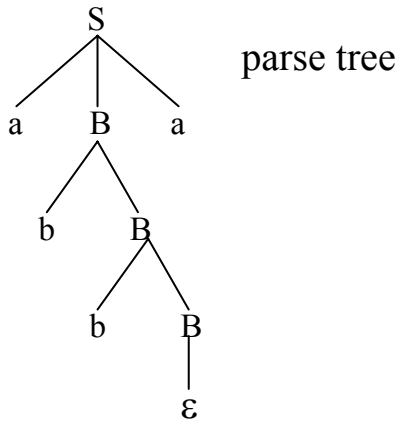
LL(1) Parsing Table

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

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

stack	**input**	**output**
$E	id+id$	E → TE'
$E'T	id+id$	T → FT'
$E'T'F	id+id$	F → id
$ E'T'id		id+id$
$ E'T'	+id$	T' → ε
$ E'	+id$	E' → +TE'
$ E'T+	+id$	
$ E'T	id$	T → FT'
$ E'T'F	id$	F → id
$ E'T'id	id$	
$ E'T'	$	T' → ε
$ E'	$	E' → ε
$	$	accept

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .

- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if $S \Rightarrow \alpha A a \beta$
  - $ is in FOLLOW(A) if $S \Rightarrow \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol
  ➔ FIRST(X)={X}
- If X is a non-terminal symbol and X → ε is a production rule
  ➔ ε is in FIRST(X).
- If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 .. Y_n$ is a production rule
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

- If X is ε
  ➔ FIRST(X)={ε}
- If X is $Y_1Y_2..Y_n$
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

FIRST Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FIRST(F) = { (,id}
FIRST(T′) = {*, ε}
FIRST(T) = { (,id}
FIRST(E′) = {+, ε}
FIRST(E) = { (,id}

FIRST(TE′) = { (,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = { (,id}
FIRST(*FT′) = {*}
FIRST(ε) = {ε}
FIRST((E)) = { (}
FIRST(id) = {id}

Compute FOLLOW (for non-terminals)

- If S is the start symbol
  ➔ $ is in FOLLOW(S)

- if $A \rightarrow \alpha B\beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B\beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

FOLLOW Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FOLLOW(E) = { $,) }
FOLLOW(E′) = { $,) }

FOLLOW(T) = { +,), $ }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = {+, *,), $ }

Constructing LL(1) Parsing Table – Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and $ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,$]
- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table – Example

$E \rightarrow TE'$	FIRST(TE')={(,id}	➔ $E \rightarrow TE'$ into M[E,(] and M[E,id]
$E' \rightarrow +TE'$	FIRST(+TE')={+}	➔ $E' \rightarrow +TE'$ into M[E',+]
$E' \rightarrow \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(E')={$,)}	➔ none ➔ $E' \rightarrow \varepsilon$ into M[E',$] and M[E',)]
$T \rightarrow FT'$	FIRST(FT')={(,id}	➔ $T \rightarrow FT'$ into M[T,(] and M[T,id]
$T' \rightarrow *FT'$	FIRST(*FT')={*}	➔ $T' \rightarrow *FT'$ into M[T',*]
$T' \rightarrow \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(T')={$,),+}	➔ none ➔ $T' \rightarrow \varepsilon$ into M[T',$], M[T',)], M[T',+]
$F \rightarrow (E)$	FIRST((E))={(}	➔ $F \rightarrow (E)$ into M[F,(]
$F \rightarrow id$	FIRST(id)={id}	➔ $F \rightarrow id$ into M[F,id]

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

  L➔ input scanned from left to right
  L➔left most derivation
  1➔ one input symbol used as a look-head symbol to determine parser action

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

$S \rightarrow i\ C\ t\ S\ E \mid a$
$E \rightarrow e\ S \mid \varepsilon$
$C \rightarrow b$

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

FOLLOW(S) = { $,e }
FOLLOW(E) = { $,e }
FOLLOW(C) = { t }

Here there will be two production rules for M[E,e]
Problem ➔ ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST(Aα) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST(Aα) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  - Both α and β cannot derive strings starting with same terminals.
  - At most one of α and β can derive to ε.
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).