Top-Down Parsing

- A parser is top-down if it discovers a parse tree top to bottom
  - A top-down parse corresponds to a preorder traversal of the parse tree
  - A leftmost derivation is applied at each derivation step
- Top-down parsers come in two forms
  - **Predictive Parsers**
    - Predict the production rule to be applied using lookahead tokens
  - **Backtracking Parsers**
    - Will try different productions, backing up when a parse fails
- Predictive parsers are much faster than backtracking ones
  - Predictive parsers operate in linear time – will be our focus
  - Backtracking parsers operate in exponential time – will not be considered
- Two kinds of top-down parsing techniques will be studied
  - Recursive-descent parsing
  - LL parsing

Top-Down Parsing by Recursive-Descent

- We view a nonterminal *A* as a definition of a procedure *A*
  - Procedure *A* will match the token sequence generated by nonterminal *A*
- The RHS of a production of *A* specifies the code for procedure *A*
  - Terminals are matched against input tokens
  - Nonterminals produce calls to corresponding procedures
- If multiple production rules exist for a nonterminal *A*
  - One of them is predicted based on a **lookahead token**
    - The lookahead token is the next input token that should be matched
  - The predicted rule is the only one that applies
    - Other rules will NOT be tried
    - This is a predictive parsing technique, not a backtracking one
- A syntax error is detected when
  - Next token in the input sequence does NOT match the expected token

Example on Recursive-Descent Parsing

- Consider the following grammar for expressions in EBNF notation

  expr → *term* { **addop** *term* }

  term → *factor* { **mulop** *factor* }

  factor → ‘**(**’ *expr* ‘**)**’ | **id** | **num**

- Since three nonterminals exist, we need three parsing procedures
  - The curly brackets expressing repetition is translated into a while loop
  - The vertical bar expressing alternation is translated into a case statement

```
procedure expr( )
begin
  term( );
  while token = ADDOP do
    match(ADDOP);
    term( );
  end while;
end expr;
```

```
procedure term( )
begin
    factor( );
    while token = MULOP do
        match(MULOP);
        factor( );
    end while;
end term;
```

```
procedure factor( )
begin
  case token of
    '(':    match('('); expr( ); match(')');
    ID:     match(ID);
    NUM:    match(NUM);
    else    syntax_error(token);
  end case;
end factor;
```

Lookahead Token and Match Procedure

- The recursive-descent procedures use a *token* variable
- The *token* variable is the **lookahead token**
  - Keeps track of the next token in the input sequence
  - Is initialized to the first token before parsing begins
  - Is updated after every call to the match procedure
- The *match* procedure matches lookahead *token* with its parameter
  - Is called to match an **expected token** on the RHS of a production
  - Match succeeds if expected token = lookahead token and fails otherwise
  - Match calls scanner function to update the lookahead token

```
procedure match ( ExpectedToken )
begin
   if token = ExpectedToken then token := scan( ) ;
   else syntax_error( token , ExpectedToken ) ;
   end if;
end match ;
```

Syntax Tree Construction for Expressions

- A recursive-descent parser can be used to construct a syntax tree

  SyntaxTree := *expr* ( ) ; Calling parser function for start symbol

- Parsing functions allocate and return pointers to syntax tree nodes
- Construction of a syntax tree for simple expressions is given below
  - **New** *node* allocates a tree node and returns a pointer to it

```
function expr( ) : TreePtr
begin
   left := term( );
   while token = ADDOP do
      op := ADDOP.op ; match(ADDOP);
      right := term( );
      left := new node(op, left, right);
   end while;
   return left;
end expr;
```

```
function term( ) : TreePtr
begin
   left := factor( );
   while token = MULOP do
      op := MULOP.op;  match(MULOP);
      right := factor( );
      left := new node(op, left, right);
   end while;
   return left;
end term;
```

Syntax Tree Construction – cont'd

- For a *factor*, we have the following parsing function
  - *symtable.lookup*(ID.*name*) searches a symbol table for a given *name*
  - *lookup* function returns a pointer to an identifier symbol in *symtable*
  - Identifiers are inserted into symbol table when parsing a declaration
  - The NUM.*ptr* is a pointer to a literal symbol in the literal table
  - Literal constants are inserted into the literal table when scanned

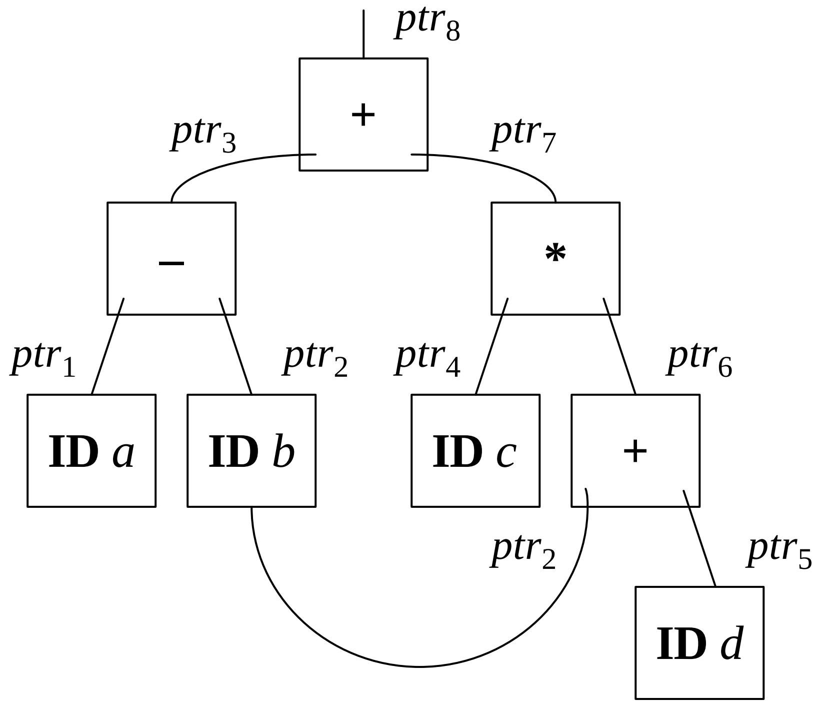
```
function factor( ) : TreePtr
begin
   case token of
      '(':     match('('); ptr := expr( ); match(')');
      ID:      ptr := symtable.lookup(ID.name); match(ID);
      NUM:     ptr := NUM.ptr; match(NUM);
      else     syntax_error(token, "Expecting a number, an identifier, or (" );
   end case;
   return ptr;
end factor;
```

Node Structure for Expression Trees

- A syntax tree node for expressions should have at least:
  - **Node operaror**: **+** , **–** , ***** , **/** , etc. Different for each operator
    - For symbol table entries, the node operator is **ID**
    - For literal table entries, the node operator is **NUM**
    - Other node operators can be added to statements and various types of literals
  - **Left Pointer**: pointer to left child expression tree
    - Can point to a tree node, to a symbol node, or to a literal node
  - **Right Pointer**: pointer to right child expression tree
    - Can point to a tree node, to a symbol node, or to a literal node
- The following fields are also important:
  - **Line, Pos**: keeps track of line and position of each tree node
  - **Type**: associates a type with each tree node
    - Type information is used to check the type of expressions

Tracing the Construction of a Syntax Tree

- Although recursive-descent is a top-down parsing technique …
  - The construction of the syntax tree for expressions is bottom up
  - Tracing verifies the precedence and associativity of operators
- The tree construction of $a - b + c * (b + d)$ is given below
  - $ptr_1 \leftarrow$ *symtable.lookup*(a)
  - $ptr_2 \leftarrow$ *symtable.lookup*(b)
  - $ptr_3 \leftarrow$ **new** *node*('–' , ptr_1 , ptr_2)
  - $ptr_4 \leftarrow$ *symtable.lookup*(c)
  - $ptr_2 \leftarrow$ *symtable.lookup*(b)
  - $ptr_5 \leftarrow$ *symtable.lookup*(d)
  - $ptr_6 \leftarrow$ **new** *node*('+' , ptr_2 , ptr_5)
  - $ptr_7 \leftarrow$ **new** *node*('*' , ptr_4 , ptr_6)
  - $ptr_8 \leftarrow$ **new** *node*('+' , ptr_3 , ptr_7)



Syntax Tree Construction for **if** Statements

- An Extended BNF grammar for **if** statements with optional **else**:

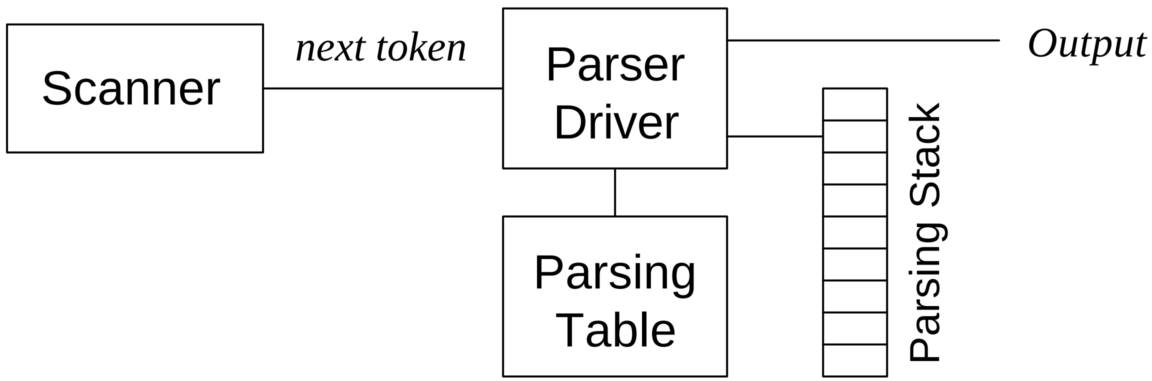
  if-stmt $\rightarrow$ **if** *expr* **then** *stmt* **[else** *stmt* **]**

- A parsing function can eliminate the ambiguity of **else**
  - By matching an **else** token as soon as encountered
- Syntax tree of **if** stmt is constructed bottom-up

```
function ifstmt( ) : TreePtr
begin
   match(IF);  exprptr := expr( ); match(THEN);  thenptr := stmt( );
   if token = ELSE then
      match(ELSE); elseptr := stmt( );
      elseptr := new node(ELSE, thenptr, elseptr);        // ELSE node
      ifptr := new node(IF, exprptr, elseptr);            // IF node points to ELSE node
   else  ifptr := new node(IF, exprptr, thenptr); end if; // No ELSE node
   return ifptr;
end ifstmt;
```

LL Parsing

- Uses an explicit stack rather than recursive calls to perform a parse
- LL(*k*) parsing means that *k* tokens of lookahead are used
  - The first L means that token sequence is read from left to right
  - The second L means a leftmost derivation is applied at each step
- An LL parser consists of
  - **Parser stack** that holds grammar symbols: non-terminals and tokens
  - **Parsing table** that specifies the parser action
  - **Driver function** that interacts with parser stack, parsing table and scanner



LL Parsing Actions

- The LL parsing actions are:
  - **Match**: to match top of parser stack with next input token
  - **Predict**: to predict a production and apply it in a derivation step
  - **Accept**: to accept and terminate the parsing of a sequence of tokens
  - **Error**: to report an error message when matching or prediction fails
- Consider the following grammar: $S \to (S) S \mid \varepsilon$

Parsing of (( ))

Stack grows backward from right to left

Parser Stack	Input	Parser Action
S	(( )) $	Predict $S \to (S) S$
(S) S	(( )) $	Match (
S) S	( )) $	Predict $S \to (S) S$
(S) S) S	( )) $	Match (
S) S) S	)) $	Predict $S \to \varepsilon$
) S) S	)) $	Match)
S) S	) $	Predict $S \to \varepsilon$
) S	) $	Match)
S	$	Predict $S \to \varepsilon$
Empty	$	Accept

Grammar Analysis: Nonterminals that Derive ε

- Grammar analysis is necessary to
  - Determine whether a grammar can be used in LL parsing
  - Construct the LL parsing table that defines the actions of an LL parser
- A common analysis is to determine which nonterminals derive ε
  - Nonterminals that derive ε are called **nullable**
- To determine which nonterminals derive ε …
  - We use an iterative marking algorithm
  - First, nonterminals that derive ε directly in one step are marked
  - Nonterminals that derive ε in two, three, … steps are found and marked
  - Continue until no more nonterminals can be marked as deriving ε
- Consider the following grammar

$A \rightarrow B\ C\ D$

$B \rightarrow \mathbf{b}\ C \mid \varepsilon$

$C \rightarrow \mathbf{c}\ D \mid \varepsilon$

$D \rightarrow \mathbf{d} \mid \varepsilon$

A, *B*, *C*, and *D* are all **nullable**

B, *C*, and *D* derive ε directly

A derives ε indirectly: $A \Rightarrow B\ C\ D \Rightarrow C\ D \Rightarrow D \Rightarrow \varepsilon$

Grammar Analysis: The First Set

- Suppose we have the following grammar:
  - The RHS of the productions of *S* do not begin with terminals
  - Parser has no immediate guidance which production to apply to expand *S*
  - We may follow all possible derivations of *S* as shown below

$$
\begin{aligned}
S &\rightarrow A\,\mathbf{a} \mid B\,\mathbf{b} \\
A &\rightarrow D\,\mathbf{c} \mid C\,A \\
B &\rightarrow \mathbf{d}\,A \mid \mathbf{e} \\
C &\rightarrow \mathbf{f}\,C \mid \mathbf{b} \\
D &\rightarrow \mathbf{h} \mid \mathbf{i}
\end{aligned}
$$

- S
  - $\Rightarrow A\,\mathbf{a}$
    - $\Rightarrow D\,\mathbf{c}\,\mathbf{a}$
      - $\Rightarrow \mathbf{h}\,\mathbf{c}\,\mathbf{a}$
      - $\Rightarrow \mathbf{i}\,\mathbf{c}\,\mathbf{a}$
    - $\Rightarrow C\,A\,\mathbf{a}$
      - $\Rightarrow \mathbf{f}\,C\,A\,\mathbf{a} \Rightarrow \ldots$
      - $\Rightarrow \mathbf{b}\,A\,\mathbf{a} \Rightarrow \ldots$
  - $\Rightarrow B\,\mathbf{b}$
    - $\Rightarrow \mathbf{d}\,A\,\mathbf{b} \Rightarrow \ldots$
    - $\Rightarrow \mathbf{e}\,\mathbf{b}$

- We predict $S \rightarrow A\,\mathbf{a}$ when
  - First token is **h**, **i**, **f**, or **b**. First(*A***a**) = {**h**, **i**, **f**, **b**}
- We predict $S \rightarrow B\,\mathbf{b}$ when
  - First token is **d** or **e**. First(*B***b**) = {**d**, **e**}
- Otherwise, we have an error

Grammar Analysis: Determining the First Set

- Formally, First(α) = $\{\mathbf{a} \in T \mid \alpha \Rightarrow^* \mathrm{a}\beta\}$
  - First(α) is the set of all terminals that can begin a sentential form of α
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in$ First(α)
- To calculate First(α) we apply the following rules
  - $\text{First}(\varepsilon) = \{\varepsilon\}$ $\alpha = \varepsilon$
  - $\text{First}(\mathbf{a}\beta) = \{\mathbf{a}\}$ $\alpha = \mathbf{a}\beta$
  - $\text{First}(A) = \text{First}(\beta_1) \cup \text{First}(\beta_2) \cup \ldots$ $\alpha = A$ and $A \to \beta_1 \mid \beta_2 \mid \ldots$
  - $\text{First}(A\beta) = \text{First}(A)$ $\alpha = A\beta$ and A is NOT nullable
  - $\text{First}(A\beta) = (\text{First}(A) - \{\varepsilon\}) \cup \text{First}(\beta)$ $\alpha = A\beta$ and $A \Rightarrow^+ \varepsilon$
- Consider the following grammar:

$S \to A\ B\ C\ \mathbf{d}$	$\text{First}(A) = \{\mathbf{e}, \mathbf{f}, \varepsilon\}$
$A \to \mathbf{e} \mid \mathbf{f} \mid \varepsilon$	$\text{First}(B) = \{\mathbf{g}, \mathbf{h}, \varepsilon\}$
$B \to \mathbf{g} \mid \mathbf{h} \mid \varepsilon$	$\text{First}(C) = \{\mathbf{p}, \mathbf{q}\}$
$C \to \mathbf{p} \mid \mathbf{q}$	$\text{First}(S) = \text{First}(ABC\mathbf{d}) = (\text{First}(A) - \{\varepsilon\}) \cup (\text{First}(B) - \{\varepsilon\}) \cup \text{First}(\text{Cd})$
	$= \{\mathbf{e}, \mathbf{f}\} \cup \{\mathbf{g}, \mathbf{h}\} \cup \{\mathbf{p}, \mathbf{q}\} = \{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}, \mathbf{p}, \mathbf{q}\}$

Grammar Analysis: The Follow Set

- Suppose we have the following grammar
  - We follow derivations of S as shown below …

$$
\begin{array}{lll}
S & \rightarrow & A\ \mathbf{c}\ B \\
A & \rightarrow & \mathbf{a}\ A \\
A & \rightarrow & \varepsilon \\
B & \rightarrow & \mathbf{b}\ B\ S \\
B & \rightarrow & \varepsilon
\end{array}
$$

$S \Rightarrow A\ \mathbf{c}\ B$

- $\Rightarrow \mathbf{a}\ A\ \mathbf{c}\ B \Rightarrow \ldots$
- $\Rightarrow \mathbf{c}\ B$
  - $\Rightarrow \mathbf{c}\ \mathbf{b}\ B\ S \Rightarrow \mathbf{c}\ \mathbf{b}\ B\ A\ \mathbf{c}\ B$
    - $\Rightarrow \mathbf{c}\ \mathbf{b}\ B\ \mathbf{a}\ A\ \mathbf{c}\ B \Rightarrow \ldots$
    - $\Rightarrow \mathbf{c}\ \mathbf{b}\ B\ \mathbf{c}\ B \Rightarrow \ldots$
  - $\Rightarrow \mathbf{c}$

- We predict $A \rightarrow \mathbf{a}\ A$ when
  - Next token is **a** because First(**a***A*) = {**a**}
- We predict $A \rightarrow \varepsilon$ when
  - Next token is **c** because Follow(*A*) = {**c**}
- Similarly, we predict $B \rightarrow \mathbf{b}\ B\ S$ when
  - Next token is **b** because First(**b** *B S*) = {**b**}
- We predict $B \rightarrow \varepsilon$ when
  - Next token is **a**, **c**, or **$** (**end-of-file token**) because Follow(*B*) = {**a**, **c**, **$**}

Grammar Analysis: Determining the Follow Set

- Formally, Follow(A) = $\{\mathbf{a} \in T \mid S \Rightarrow^+ \alpha\, A\, \mathbf{a}\, \beta\}$
  - Follow(A) is the set of terminals that may follow A in any sentential form
- If $S \Rightarrow^+ \alpha\, A$ then $\$ \in$ Follow(A)
  - If A is not followed by any terminal then it is followed by the end of file
  - The **$** represents the end-of-file token
- We compute Follow(A) using the following rules:
  - If A is the start symbol then $\$ \in$ Follow(A)
  - Inspect RHS of productions for all occurrences of A

    Let a typical production be $B \to \alpha\, A\, \beta$
    - If β is NOT nullable then add First(β) to Follow(A)
      - Any token that can begin a sentential form of β can follow A
    - If β is ε or derives ε then add (First(β) – $\{\varepsilon\}$) $\cup$ Follow(B) to Follow(A)
      - If β vanishes in a given derivation then what follows A is what follows B
      - $\ldots \Rightarrow \delta\, B\, \gamma \Rightarrow \delta\, \alpha\, A\, \beta\, \gamma \Rightarrow \delta\, \alpha\, A\, \gamma \Rightarrow \ldots$ what follows A is what follows B is First(γ)

Examples on the First and Follow Sets

Example 1:

$S \rightarrow A\ \mathbf{c}\ B$
$A \rightarrow \mathbf{a}\ A$
$A \rightarrow \varepsilon$
$B \rightarrow \mathbf{b}\ B\ S$
$B \rightarrow \varepsilon$

Nonterminals that derive ε are A and B

First(S) = First($A\mathbf{c}B$) = (First(A)–{ε}) ∪ First($\mathbf{c}B$) = {**a**, **c**}
First(A) = First($\mathbf{a}A$) ∪ First(ε) = {**a**, ε}
First(B) = First($\mathbf{b}BS$) ∪ First(ε) = {**b**, ε}

Follow(S) = {**$**} ∪ Follow($B$)
Follow(A) = {**c**}
Follow(B) = Follow(S) ∪ First(S) = {**$**, **a**, **c**}
Follow(S) = {**$**, **a**, **c**}

Example 2:

$E \rightarrow T\ Q$
$Q \rightarrow \mathbf{+}\ T\ Q \mid \mathbf{-}\ T\ Q \mid \varepsilon$
$T \rightarrow F\ R$
$R \rightarrow \mathbf{*}\ F\ R \mid \mathbf{/}\ F\ R \mid \varepsilon$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id}$

Nonterminals that derive ε are Q and R

First(E) = First(TQ) = First(T) = First(FR) = First(F) = {**(** , **id**}
First(Q) = {**+** , **–** , ε}
First(R) = {***** , **/** , ε}

Follow(E) = {**$** , **)**}
Follow(Q) = Follow(E) = {**$** , **)**}
Follow(T) = (First(Q)–{ε}) ∪ Follow(E) ∪ Follow(Q) = {**+**, **–**, **$**, **)**}
Follow(R) = Follow(T) = {**+**, **–**, **$**, **)**}
Follow(F) = (First(R)–{ε}) ∪ Follow(T) ∪ Follow(R) = {*****, **/**, **+**, **–**, **$**, **)**}

Grammar Analysis: Determining the Predict Set

- The predict set of a production $A \to \alpha$ is defined as follows:
  - If α is **NOT nullable** then Predict($A \to \alpha$) = First(α)
  - If α is **Nullable** then Predict($A \to \alpha$) = (First(α) – {ε}) $\cup$ Follow(A)
  - This is the set of lookahead tokens that will cause the selection of $A \to \alpha$
- Example on determining the predict set:

$E \to T\ Q$	Predict $E \to T\ Q$ = First(TQ) = First(T) = {**(** , **id**}
$Q \to \mathbf{+}\ T\ Q$	Predict $Q \to \mathbf{+}\ T\ Q$ = First(**+**TQ) = { **+** }
$Q \to \mathbf{–}\ T\ Q$	Predict $Q \to \mathbf{–}\ T\ Q$ = First(**–**TQ) = { **–** }
$Q \to \varepsilon$	Predict $Q \to \varepsilon$ = Follow(Q) = {**$** , **)**}
$T \to F\ R$	Predict $T \to F\ R$ = First(FR) = First(F) = {**(** , **id**}
$R \to \mathbf{*}\ F\ R$	Predict $R \to \mathbf{*}\ F\ R$ = First(*****FR) = { ***** }
$R \to \mathbf{/}\ F\ R$	Predict $R \to \mathbf{/}\ F\ R$ = First(**/**FR) = { **/** }
$R \to \varepsilon$	Predict $R \to \varepsilon$ = Follow(R) = {**+** , **–** , **$** , **)**}
$F \to \mathbf{(}\ E\ \mathbf{)}$	Predict $F \to \mathbf{(}\ E\ \mathbf{)}$ = { **(** }
$F \to \mathbf{id}$	Predict $F \to \mathbf{id}$ = { **id** }

LL(1) Grammars

- Not all context-free grammars are suitable for LL parsing
- CFGs suitable for LL(1) parsing are called **LL(1) Grammars**
- A grammar is LL(1) if for productions with the same LHS A

  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

  Predict($A \rightarrow \alpha_i$) $\cap$ Predict($A \rightarrow \alpha_j$) = $\varnothing$ for all $i \neq j$

  The predict sets of productions with same LHS are pairwise disjoint

- The following grammar is LL(1)

$S \rightarrow A\ \mathbf{c}\ B$	
$A \rightarrow \mathbf{a}\ A$	Predict($A \rightarrow \mathbf{a}\ A$) = {**a**}
$A \rightarrow \varepsilon$	Predict($A \rightarrow \varepsilon$) = Follow($A$) = {**c**}
$B \rightarrow \mathbf{b}\ B\ S$	Predict($B \rightarrow \mathbf{b}\ B\ S$) = {**b**}
$B \rightarrow \varepsilon$	Predict($B \rightarrow \varepsilon$) = Follow($B$) = Follow($S$) $\cup$ First(S) =
	= {**$**} $\cup$ Follow(B) $\cup$ {**a**, **c**} = {**$**, **a**, **c**}

Disjoint

Disjoint

Constructing the LL(1) Parsing Table

- The predict sets can be represented in an LL(1) parse table
  - The rows are indexed by the nonterminals
  - The columns are indexed by the tokens
- If A is a nonterminal and **tok** is the lookahead token then
  - ***Table***[A][**tok**] indicates which production to predict
  - If no production can be used ***Table***[A][**tok**] gives an error value
- ***Table***[A][**tok**] = $A \rightarrow \alpha$ iff **tok** $\in$ predict($A \rightarrow \alpha$)
- Example on constructing the LL(1) parsing table:

1:	$S \rightarrow A\ \mathbf{c}\ B$	Predict(1) = {**a**, **c**}
2:	$A \rightarrow \mathbf{a}\ A$	Predict(2) = {**a**}
3:	$A \rightarrow \varepsilon$	Predict(3) = {**c**}
4:	$B \rightarrow \mathbf{b}\ B\ S$	Predict(4) = {**b**}
5:	$B \rightarrow \varepsilon$	Predict(5) = {**$**, **a**, **c**}

	a	**b**	**c**	**$**
S	1		1	
A	2		3	
B	5	4	5	5

Empty slots indicate error conditions

Constructing the LL(1) Parsing Table – cont'd

❖ Here is a second example on constructing the parsing table

1: $E \rightarrow T\ Q$	Predict(1)	= { **(** , **id** }
2: $Q \rightarrow$ **+** $T\ Q$	Predict(2)	= { **+** }
3: $Q \rightarrow$ **–** $T\ Q$	Predict(3)	= { **–** }
4: $Q \rightarrow \varepsilon$	Predict(4)	= { **$** , **)** }
5: $T \rightarrow F\ R$	Predict(5)	= { **(** , **id** }
6: $R \rightarrow$ ***** $F\ R$	Predict(6)	= { ***** }
7: $R \rightarrow$ **/** $F\ R$	Predict(7)	= { **/** }
8: $R \rightarrow \varepsilon$	Predict(8)	= {**+**, **–**, **$**, **)**}
9: $F \rightarrow$ **(** E **)**	Predict(9)	= { **(** }
10: $F \rightarrow$ **id**	Predict(10)	= { **id** }

	+	**–**	*****	**/**	**(**	**)**	**id**	**$**
E					1		1	
Q	2	3				4		4
T					5		5	
R	8	8	6	7		8		8
F					9		10	

❖ Because the above grammar is LL(1)

- A unique production number is stored in a table entry

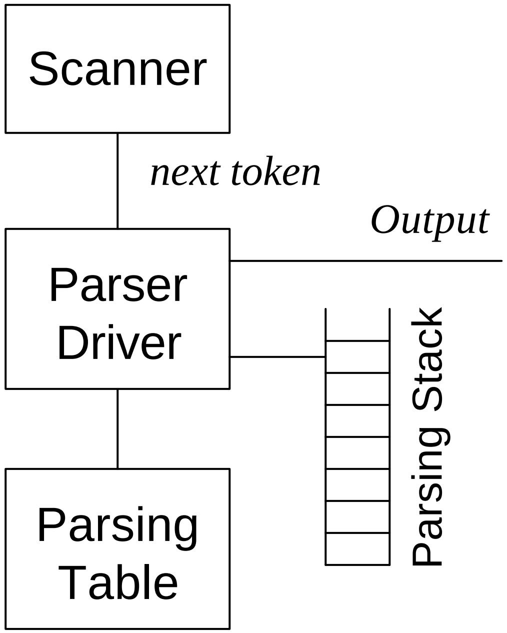
❖ Blank entries correspond to error conditions

- In practice, special error numbers are used to indicate error situations

LL(1) Parser Driver Algorithm

- The LL(1) parser driver algorithm can be described as follows:

Token := *scan*( )
Stack.*push*(*StartSymbol*)
while not *Stack*.*empty*( ) **do**
  X := *Stack*.*pop*( )
  if *terminal*(*X*)
    if *X* = *Token* **then** *Token* := *scan*( )
    else process a syntax error at *Token* **end if**
  else (* X is a nonterminal *)
    Rule := *Table*[*X*][*Token*]
    if $Rule = X \rightarrow Y_1\ Y_2\ \dots\ Y_n$ **then**
      for *i* **from** *n* **downto** 1 **do** *Stack*.*push*(Y_i) **end for**
    else process a syntax error at *Token* **end if**
  end if
end while
if *Token* = $ **then** accept parsing
else report a syntax error at *Token* **end if**



Tracing an LL(1) Parser

Consider the parsing of **id * (id + id) $**

1:	$E \rightarrow T\ Q$	6:	$R \rightarrow$ ***** $F\ R$
2:	$Q \rightarrow$ **+** $T\ Q$	7:	$R \rightarrow$ **/** $F\ R$
3:	$Q \rightarrow$ **–** $T\ Q$	8:	$R \rightarrow \varepsilon$
4:	$Q \rightarrow \varepsilon$	9:	$F \rightarrow$ **(** E **)**
5:	$T \rightarrow F\ R$	10:	$F \rightarrow$ **id**

	+	**–**	*****	**/**	**(**	**)**	**id**	**$**
E					1		1	
Q	2	3				4		4
T					5		5	
R	8	8	6	7		8		8
F					9		10	

Stack grows backwards from right to left

Parser Stack	Input	Parser Action
E	**id*(id+id)$**	Predict $E \rightarrow T\ Q$
$T\ Q$	**id*(id+id)$**	Predict $T \rightarrow F\ R$
$F\ R\ Q$	**id*(id+id)$**	Predict $F \rightarrow$ **id**
id $R\ Q$	**id*(id+id)$**	Match **id**
$R\ Q$	***(id+id)$**	Predict $R \rightarrow$ ***** $F\ R$
***** $F\ R\ Q$	***(id+id)$**	Match *****
$F\ R\ Q$	**(id+id)$**	Predict $F \rightarrow$ **(** E **)**
(E **)** $R\ Q$	**(id+id)$**	Match **(**
E **)** $R\ Q$	**id+id)$**	Predict $E \rightarrow T\ Q$
$T\ Q$ **)** $R\ Q$	**id+id)$**	Predict $T \rightarrow F\ R$
$F\ R\ Q$ **)** $R\ Q$	**id+id)$**	Predict $F \rightarrow$ **id**
id $R\ Q$ **)** $R\ Q$	**id+id)$**	Match **id**
$R\ Q$ **)** $R\ Q$	**+id)$**	Predict $R \rightarrow \varepsilon$
Q **)** $R\ Q$	**+id)$**	Predict $Q \rightarrow$ **+** $T\ Q$
+ $T\ Q$ **)** $R\ Q$	**+id)$**	Match **+**
$T\ Q$ **)** $R\ Q$	**id)$**	Predict $T \rightarrow F\ R$
$F\ R\ Q$ **)** $R\ Q$	**id)$**	Predict $F \rightarrow$ **id**
id $R\ Q$ **)** $R\ Q$	**id)$**	Match **id**
$R\ Q$ **)** $R\ Q$	**)$**	Predict $R \rightarrow \varepsilon$
Q **)** $R\ Q$	**)$**	Predict $Q \rightarrow \varepsilon$
) $R\ Q$	**)$**	Match **)**
$R\ Q$	**$**	Predict $R \rightarrow \varepsilon$
Q	**$**	Predict $Q \rightarrow \varepsilon$
Empty	**$**	Accept

The Problem of Left Recursion

- Left recursive grammars fail to be LL(1) or even LL(*k*)
  - A left recursive production puts an LL parser into infinite loop
  - If a left recursive production is predicted then
    - Nonterminal on LHS is replaced with RHS of production
    - The same nonterminal will appear again on top of parser stack
    - The same production is predicted again
    - Iteration goes forever
- Left recursion is commonly used to
  - Make an operation left associative
    - *Expr* $\rightarrow$ *Expr* **addop** *Term* | *Term*
  - Specify a list of identifiers, statements, etc.
    - *StmtList* $\rightarrow$ *StmtList* **;** *Statement* | *Statement*
- We need to eliminate left recursion to make a grammar LL(1)

Eliminating Immediate Left Recursion

- The simplest case of left recursion is **immediate left recursion**
  - General Form: $A \to A\,\alpha \mid \beta$
  - The above productions of A generate strings of the form $\beta\alpha^n$, $n \geq 0$
  - We introduce a new nonterminal and use right recursion as follows:

$$
\begin{aligned}
A &\to \beta\, Atail \\
Atail &\to \alpha\, Atail \mid \varepsilon
\end{aligned}
$$

- In general, if many immediate left recursive productions exist
  - General Form: $A \to A\,\alpha_1 \mid A\,\alpha_2 \mid \dots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$
  - We introduce a new nonterminal and use right recursion

$$
\begin{aligned}
A &\to \beta_1\, Atail \mid \beta_2\, Atail \mid \dots \mid \beta_m\, Atail \\
Atail &\to \alpha_1\, Atail \mid \alpha_2\, Atail \mid \dots \mid \alpha_n\, Atail \mid \varepsilon
\end{aligned}
$$

- For example: $Expr \to Expr + Term \mid Expr - Term \mid Term$ becomes:

$$
\begin{aligned}
Expr &\to Term\;\; Exprtail \\
Exprtail &\to +\; Term\;\; Exprtail \mid -\; Term\;\; Exprtail \mid \varepsilon
\end{aligned}
$$

Eliminating Indirect Left Recursion

- In some cases, **left recursion may be indirect**

  For example: $A \to B\,\beta \mid \ldots$ and $B \to A\,\alpha \mid \ldots$

- We can do substitutions to make left recursion immediate
- Consider the following grammar:

  $A \to B\,\mathbf{a} \mid A\,\mathbf{a} \mid \mathbf{c}$

  $B \to B\,\mathbf{b} \mid A\,\mathbf{b} \mid \mathbf{d}$

- First, we remove the immediate left recursion of A

  $A \to B\,\mathbf{a}\,Atail \mid \mathbf{c}\,Atail$ $Atail \to \mathbf{a}\,Atail \mid \varepsilon$

- Second, we eliminate the indirect left recursion of $B \to A\,\mathbf{b}$

  $B \to B\,\mathbf{b} \mid B\,\mathbf{a}\,Atail\,\mathbf{b} \mid \mathbf{c}\,Atail\,\mathbf{b} \mid \mathbf{d}$

- Finally, we remove the immediate left recursion of B

  $B \to \mathbf{c}\,Atail\,\mathbf{b}\;Btail \mid \mathbf{d}\;Btail$

  $Btail \to \mathbf{b}\,Btail \mid \mathbf{a}\,Atail\,\mathbf{b}\;Btail \mid \varepsilon$

Left Factoring of Common Prefixes

- Another problem to LL parsers is to have a **common prefix**
- An **if** statement may have 2 production with a common prefix:

  IfStmt → **if** *Expr* **then** *StmtList* **end if ;**
  IfStmt → **if** *Expr* **then** *StmtList* **else** *StmtList* **end if ;**

- An LL(1) parser cannot predict which production to apply
- The solution is use **left factoring** of the **common prefix**
  - General Form: $A \to \alpha\,\beta \mid \alpha\,\gamma \mid \ldots \mid \alpha\,\zeta$
  - Left Factoring solution:

    $A \to \alpha\, Atail$
    $Atail \to \beta \mid \gamma \mid \ldots \mid \zeta$

- The left factoring of the two **if** statement productions:

  IfStmt → **if** *Expr* **then** *StmtList* *IfStmtTail*
  IfStmtTail → **else** *StmtList* **end if ;** | **end if ;**