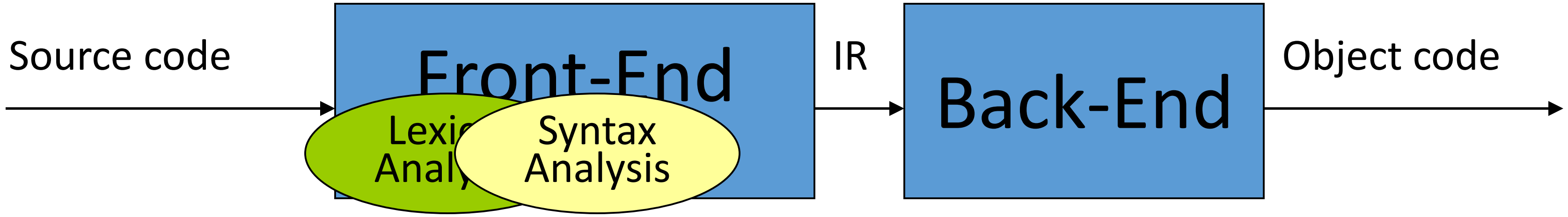
Parsing
(Syntax Analysis)

Introduction to Parsing (Syntax Analysis)



Lexical Analysis:

- Reads characters of the input program and produces tokens.

  But: Are they syntactically correct? Are they valid sentences of the input language?

Outline

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

Grammars

- Every programming language has **precise grammar rules** that describe **the syntactic structure of well-formed programs**
  - E.g., In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.

- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars

- Context-free grammars are usually used for syntax specification of programming languages

What is syntax analysis/parsing

- A parser for a grammar of a programming language
  - **verifies** that the string of tokens for a program in that language can indeed be generated from that grammar
  - **reports** any syntax errors in the program
  - **constructs** a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on context-free grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context Free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)	(2)	(3)	(4)
$E \rightarrow E + E$	$S \rightarrow 0S0$	$S \rightarrow aSb$	$S \rightarrow aB \mid bA$
$E \rightarrow E * E$	$S \rightarrow 1S1$	$S \rightarrow \epsilon$	$A \rightarrow a \mid aS \mid bAA$
$E \rightarrow (E)$	$S \rightarrow 0$		$B \rightarrow b \mid bS \mid aBB$
$E \rightarrow id$	$S \rightarrow 1$		
	$S \rightarrow \epsilon$		

Derivations

- $E \Rightarrow^{E \rightarrow E+E} E + E \Rightarrow^{E \rightarrow id} id + E \Rightarrow^{E \rightarrow id} id + id$
  is a derivation of the terminal string $id + id$ from E

- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \rightarrow E + E$, $E \rightarrow id$, and $E \rightarrow id$, are applied at steps 1,2, and, 3 respectively

- The above derivation is represented in short as,
  $E \Rightarrow^{*} id + id$, and is read as S **derives** $id + id$

Context Free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is
  $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S

- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Examples

(1)

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

$L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'

(2)

$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow \epsilon$

$L(G_2)$ = Set of palindromes over 0 and 1

Examples

(3)

$S \rightarrow aSb$

$S \rightarrow \epsilon$

L(G_3) = {a^n b^n | n>=1}

(4)

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

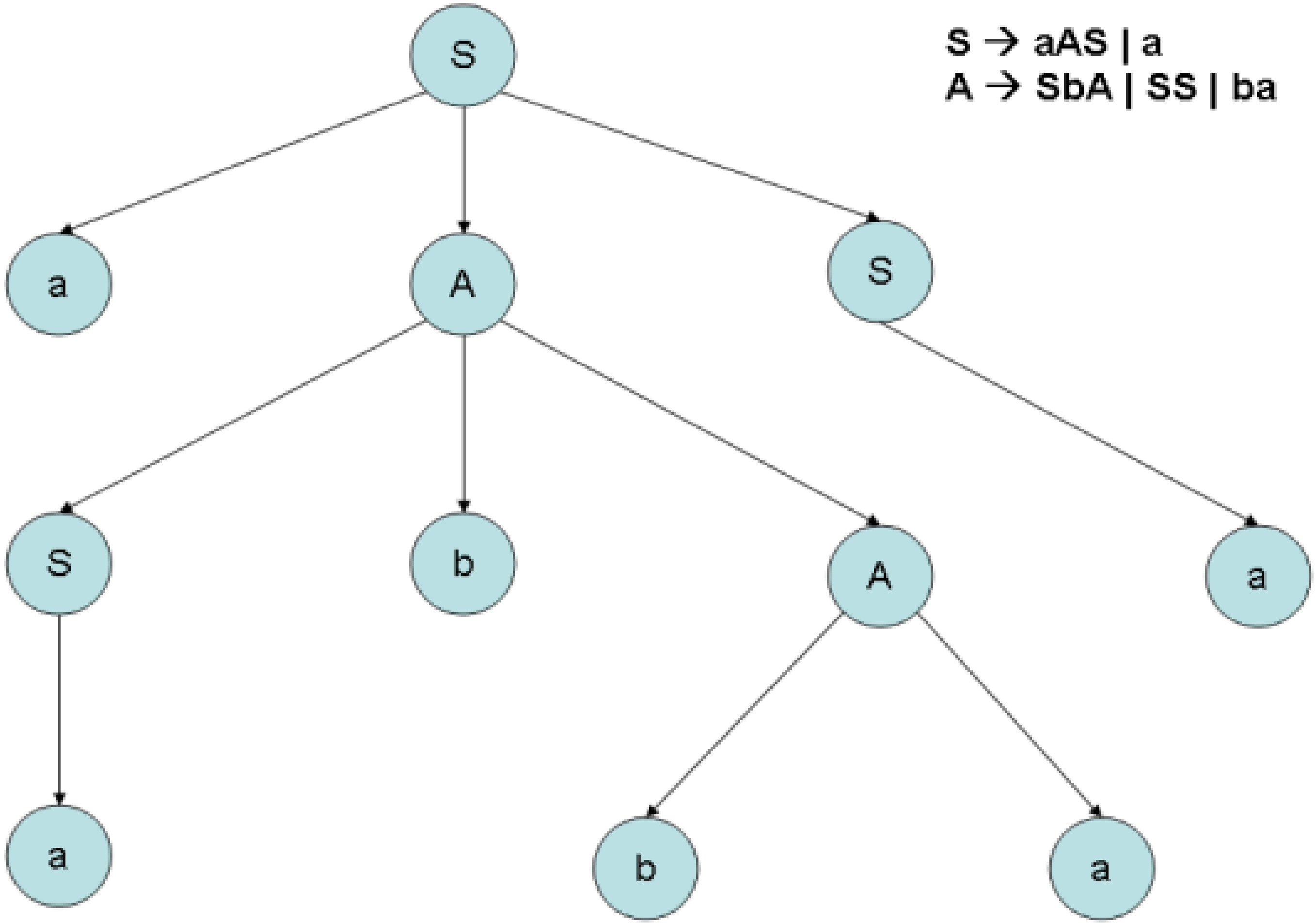
L(G_4) = {x| x has equal number of a's and b's}

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in P$, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Example: Derivation Tree

S → aAS | a
A → SbA | SS | ba



S => aAS => aSbAS => aabAS => aabbaS => aabbaa

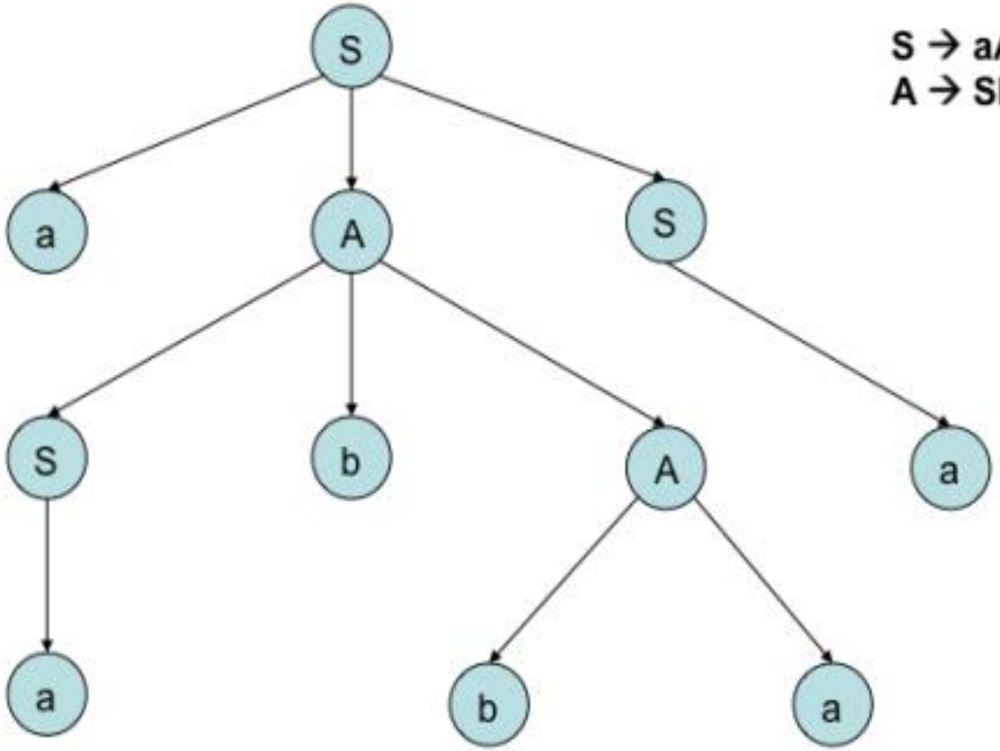
Leftmost and Rightmost Derivations

- Leftmost: at each step in a derivation, a production is applied to the **leftmost nonterminal**.

- Rightmost: at each step in a derivation, a production is applied to the **rightmost nonterminal**.

- If **w ∈ L(G)** for some **G**, then **w** has at least one parse tree and corresponding to a parse tree, **w** has unique *leftmost* and *rightmost* derivations

Example- Leftmost and Rightmost Derivations



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

Find parse tree, leftmost, rightmost derivation for: x-2*y

1. Goal $\rightarrow$ Expr
2. Expr $\rightarrow$ Expr op Expr
3. | number
4. | id
5. Op $\rightarrow$ +
6. | -
7. | *
8. | /

Ambiguity

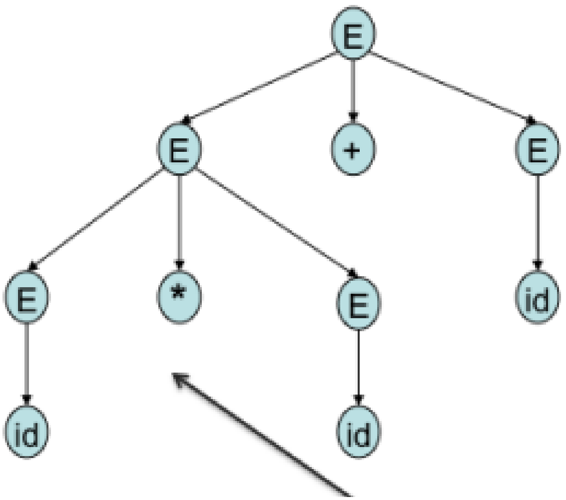
- If some word **w** in ***L(G)*** has two or more parse trees, then **G** is said to be ambiguous

- A **CFL** for which every **G** is ambiguous, is said to be an inherently ambiguous **CFL**

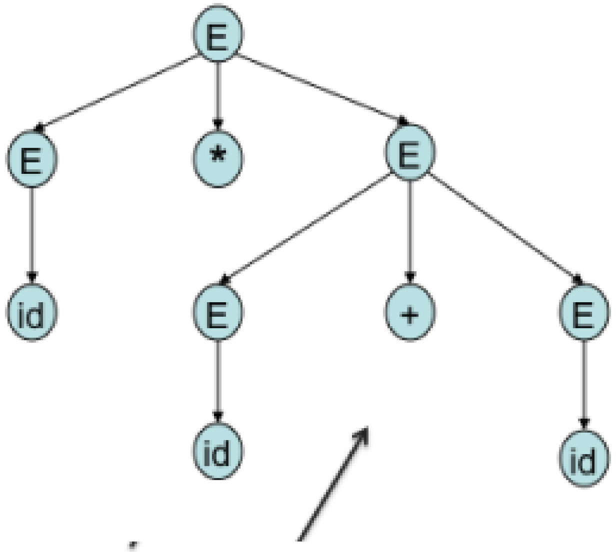
Ambiguity - Example

Is this grammar ambiguous?

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$



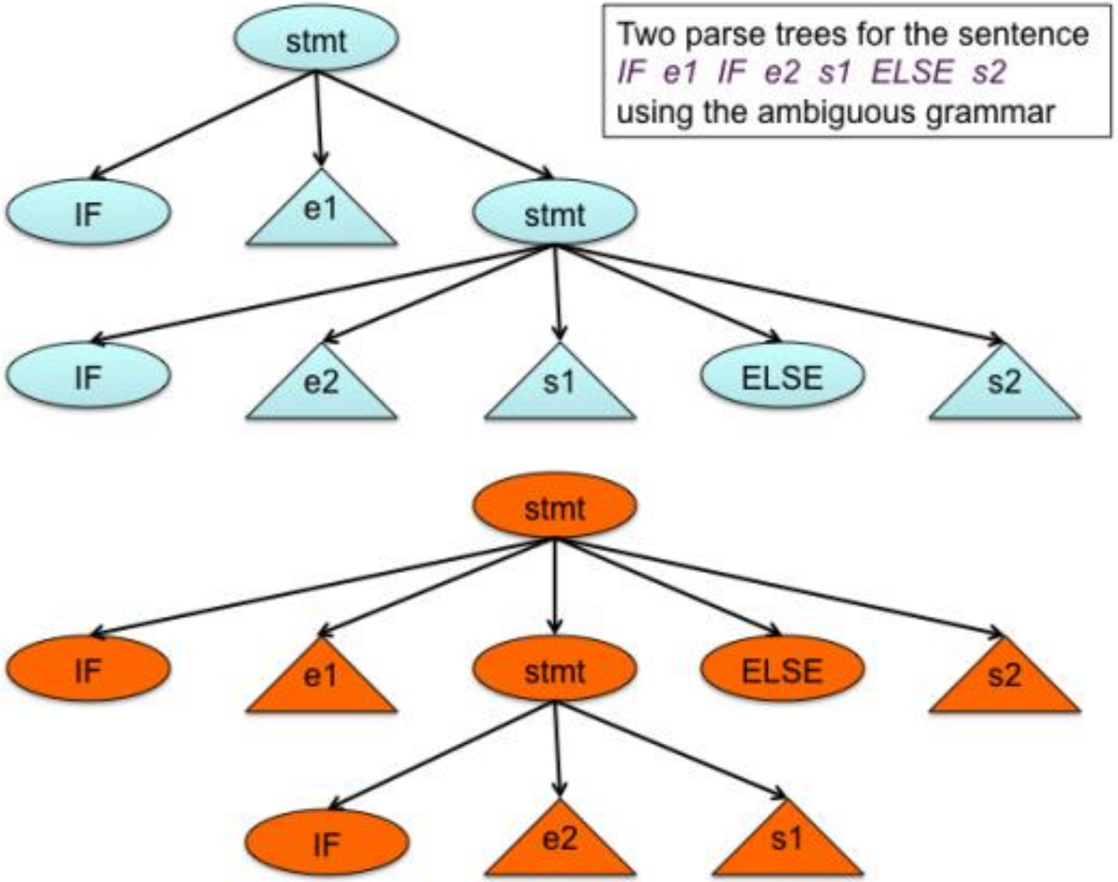
E => E+E => E*E+E => id*E+E => id*id+E => id*id+id



E => E*E => id*E => id*E+E => id*id+E => id*id+id

Ambiguity- Example

$$stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$$



Eliminating Ambiguity

- For most parsers, desirable that the grammar is made unambiguous (cannot uniquely determine which parse-tree to select otherwise)

- Use *disambiguating rules* (to discard undesirable parse trees), leaving only one tree for each sentence.

Eliminating Ambiguity

- An ambiguous grammar can be rewritten to eliminate ambiguity
- Eg:

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} + \mathbf{E} \\ &\mid \mathbf{E} * \mathbf{E} \\ &\mid \mathbf{id} \end{aligned}$$

- Parse tree(s) for id+id+id, and for id+id*id
- *Associativity* and *precedence* not taken into account (restrict recursion, introduce levels).

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T} \\ \mathbf{T} &\rightarrow \mathbf{T} * \mathbf{F} \mid \mathbf{F} \\ \mathbf{F} &\rightarrow \mathbf{id} \end{aligned}$$

Unambiguous Grammar

Ambiguity

A grammar that produces more than one parse tree for some sentence is ambiguous.

Example:

- Stmt $\rightarrow$ if Expr then Stmt | if Expr then Stmt else Stmt | …other…
- What are the derivations of:
  - **if E1 then if E2 then S1 else S2**
- Rewrite the grammar to avoid the problem
- Match each else to innermost unmatched if

Left-Recursive Grammars

- **Definition**: A grammar is ***left-recursive*** if it has a non-terminal symbol ***A***, such that there is a derivation $\boldsymbol{A \Rightarrow A a}$, for some string a.
- A left-recursive grammar can cause a (top-down) parser to go into an ***infinite loop***.
- **Eliminating left-recursion**: In many cases, it is sufficient to replace

  $A \to Aa \mid b$ with $A \to bA'$ and $A' \to aA' \mid \varepsilon$

- **Example**:

  $$Sum \to Sum{+}number \mid number$$

  would become:

  $$Sum \to number\ \ Sum'$$

  $$Sum' \to {+}number\ \ Sum' \mid \varepsilon$$

Eliminating Left Recursion

<u>General algorithm</u>: works for non-cyclic, no ε-productions grammars

1. Arrange the non-terminal symbols in order: $A_1, A_2, A_3, \ldots, A_n$
2. For i=1 to n do
for j=1 to i-1 do
I) replace each production of the form $A_i \rightarrow A_j\gamma$ *with*
the productions $A_i \rightarrow \delta_1\,\gamma \mid \delta_2\,\gamma \mid \ldots \mid \delta_k\,\gamma$
where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *are all the current* A_j *productions*
II) eliminate the immediate left recursion among the A_i

Example- Eliminating Left Recursion

Example:

1. $\textbf{\textit{Goal}} \rightarrow \textit{Expr}$
2. $\textbf{\textit{Expr}} \rightarrow \textit{Expr} + \textit{Term}$
3. $\quad | \ \textit{Expr} - \textit{Term}$
4. $\quad | \ \textit{Term}$
5. $\textbf{\textit{Term}} \rightarrow \textit{Term} * \textit{Factor}$
6. $\quad | \ \textit{Term} \,/\, \textit{Factor}$
7. $\quad | \ \textit{Factor}$
8. $\textbf{\textit{Factor}} \rightarrow \textit{number}$
9. $\quad | \ \textit{id}$

Applying the transformation:

$\textbf{\textit{Expr}} \rightarrow \textit{Term Expr}'$

$\textbf{\textit{Expr}}' \rightarrow +\textit{Term Expr}' \ | - \textit{Term Expr}' \ | \ \varepsilon$

$\textbf{\textit{Term}} \rightarrow \textit{Factor Term}'$

$\textbf{\textit{Term}}' \rightarrow *\textit{Factor Term}' \ | \ /\, \textit{Factor Term}' \ | \ \varepsilon$

($\textbf{\textit{Goal}} \rightarrow \textit{Expr}$ and $\textbf{\textit{Factor}} \rightarrow \textit{number} \ | \ \textit{id}$ remain unchanged)

Left Factoring

- Useful for transforming a grammar to be suitable for (predictive) top-down parsing

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \end{array}$$

- **Example**: On seeing input "if" we cannot immediately tell which production to choose to expand *stmt*.

- When the choice between two A-productions in unclear, we may be able to defer the decision until sufficient input is seen (to make correct choice)

- If $\boldsymbol{A} \rightarrow xB_1 + xB_2$ and the input begins with "x" should A be expanded to xB_1 or xB_2 is unclear
- Left Factored: $\boldsymbol{A} \rightarrow xA'$ $\quad A' \rightarrow B_1 + B_2$

Left Factoring

Algorithm:

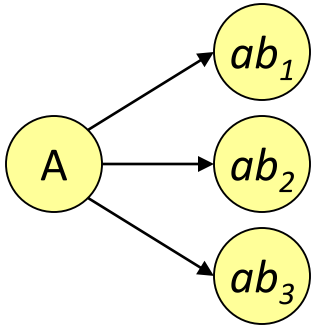
1. For each non-terminal A, find the longest prefix, say a, common to two or more of its alternatives

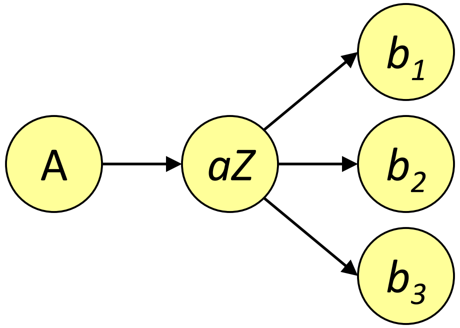
2. if $a \neq \varepsilon$ then replace all the A productions, $A \rightarrow ab_1|ab_2|ab_3|\ldots|ab_n|\gamma$, where γ is anything that does not begin with a, with $A \rightarrow aZ \mid \gamma$ and $Z \rightarrow b_1|b_2|b_3|\ldots|b_n$

Repeat the above until no common prefixes remain

Example: $A \rightarrow ab_1 \mid ab_2 \mid ab_3$ would become $A \rightarrow aZ$ and $Z \rightarrow b_1|b_2|b_3$

Note the graphical representation:





Parsing techniques

- **Top-down parsers**:
  - Construct the top node of the tree and then the rest in pre-order. (depth-first)
  - Pick a production & try to match the input; if you fail, backtrack.
  - Essentially, we try to find a **leftmost** derivation for the input string (which we scan left-to-right).
  - predictive parsing (backtrack-free).

- **Bottom-up parsers**:
  - Construct the tree for an input string, beginning at the leaves and working up towards the top (root).
  - Bottom-up parsing, using left-to-right scan of the input, tries to construct a **rightmost** derivation in reverse.
  - Handle a large class of grammars.

Top-down paring

Top-down Parsing

- Constructing a parse-tree for the input string starting from the root
- At each step of a top-down parser:
  - Determine the production to be applied for a non-terminal (say A)
  - Matching the terminal symbols in the production body with the input string

- Recursive-descent parsing (general form of top-down parsing)
  - May require backtracking to find the correct production to be applied

- Predictive parsing
  - Chooses correct production by looking ahead of the input a fixed number of symbols
  - No backtracking required

Recursive-Descent Parsing

- Consists of a set of procedures one for each non-terminal.
- Execution begins with the procedure for the start symbol.

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

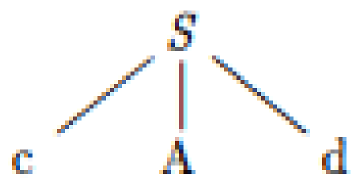
Typical procedure for a non-terminal in a top-down parser

- General recursive-descent parser may require backtracking
- Unique A production cannot be chosen (must try different productions)
- Failure at line 7 (return to line 1, try another A production)

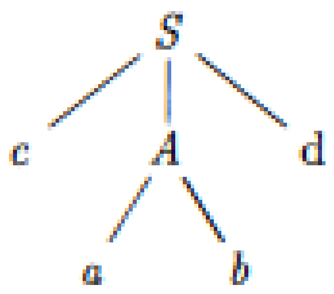
Example- Recursive-Descent Parsing

$$\begin{aligned} S &\rightarrow c\, A\, d \\ A &\rightarrow a\, b \mid a \end{aligned}$$

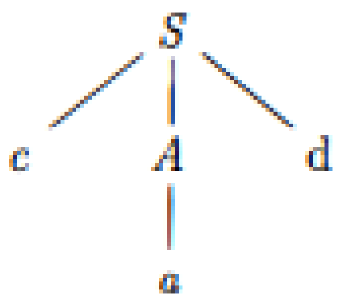
- To construct a parse tree top-down for the input string ***“cad”***
- Input pointer pointing to “***c***” initially



- Advance input pointer to “a”



- Advance input pointer to “d”
- Does not match with “b”
- Reset input pointer to position 2,
- go back, check another alternative for A



- Leaf “a” matches 2nd inp symbol
- Leaf “d” matches 3rd symbol
- **Halt**, announce ***successful***

Example (2)- Recursive-Descent Parsing

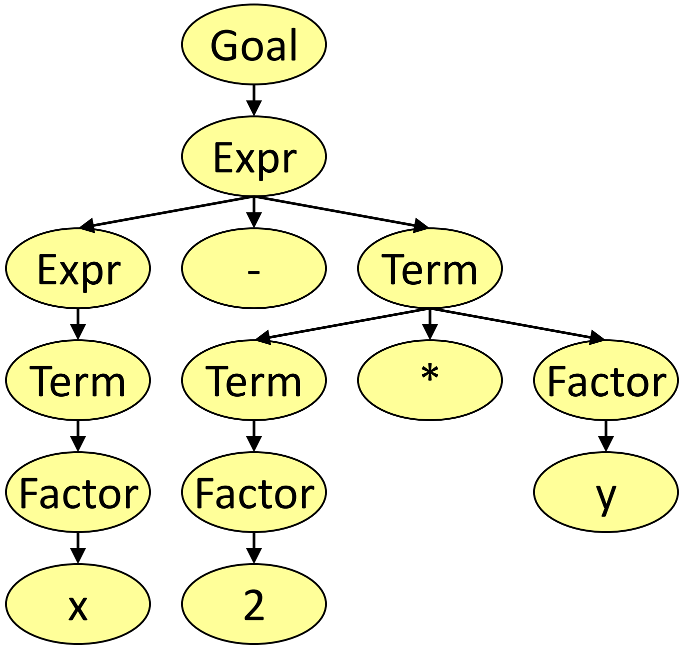
Example:

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. $\quad\quad | \; Expr - Term$
4. $\quad\quad | \; Term$
5. $Term \rightarrow Term * Factor$
6. $\quad\quad | \; Term \, / \, Factor$
7. $\quad\quad | \; Factor$
8. $Factor \rightarrow number$
9. $\quad\quad | \; id$

Example: Parse *x-2*y*

Steps

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + … + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Left-recursive grammar: Wrong choice can lead to non-termination

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.
- **<u>Idea</u>**: look ahead in input and use context to pick correctly.

- How much ***lookahead*** is needed?
  - In general, an arbitrarily large amount.
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

Predictive Parsing (First Sets)

- ***FIRST*** sets:
  - For any symbol A, *FIRST(A)* is defined as *the set of terminal symbols that appear as the first symbol of one or more strings derived from A*.

    E.g.:

$$\begin{aligned}
&\textbf{\textit{Goal}} \rightarrow Expr \\
&\textbf{\textit{Expr}} \rightarrow Term\ Expr' \\
&\textbf{\textit{Expr'}} \rightarrow +Term\ Expr' \ \mid -Term\ Expr' \ \mid \varepsilon \\
&\textbf{\textit{Term}} \rightarrow Factor\ Term' \\
&\textbf{\textit{Term'}} \rightarrow *Factor\ Term' \ \mid /\,Factor\ Term' \ \mid \varepsilon \\
&\textbf{\textit{Factor}} \rightarrow number \mid id
\end{aligned}$$

  - $FIRST(Expr') = \{+, -, \varepsilon\}$
  - $FIRST(Term') = \{*, /, \varepsilon\}$
  - $FIRST(Factor) = \{number, id\}$

FIRST Sets

If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
$FIRST(\alpha) = \{a \mid a \in T,\ and\ \alpha \Rightarrow^* ax,\ x \in T^*\}$
$FIRST(\epsilon) = \{\epsilon\}$

Computation of FIRST

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 $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, 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, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Computation of FIRST

1. For any terminal symbol ‘**a**’,   First(a) = { a }

2. If **X** is nonterminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then

   - If $\in\ \notin$ **First(Y_1)** , then **First(X) = First(Y_1)**
   - If $\in$ in **First(Y_1)**, then **First(X) = {First(Y_1) – $\in$ } $\cup$ First($Y_2 \ldots Y_k$)**
   - **…………………**
   - If **FORALL i in {1,..,k};** $\in$ in **First(Y_i) then add $\in$ to First(X)**

3. If **X** $\rightarrow$ $\in$, is a production, then add **$\in$ to** First(X)

Examples: Computing First Set

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 (A) = ?

$FIRST(S') = FIRST(S) = \{a, c\}$ 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

For non-terminal **A**;
Follow(A) is the set of terminals **a**
that can appear immediately to the right of **A** in some ***sentential form***.

If A is any nonterminal, then

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*,\ a \in T \cup \{\$\}\}$$

Examples: Computing Follow Set

Consider the following grammar

$S' \to S\$,\ S \to aAS \mid c,\ A \to ba \mid SB,\ B \to bA \mid S$

- Follow(S) = ?
- Follow(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

$S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$,$

$S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Predictive Parsing

- **Basic idea**:
  - For any production $A \rightarrow a \mid b$ we would like to have a distinct way of choosing the correct production to expand.
- ***FIRST* sets**:
  - For any symbol A, *FIRST(A)* is defined as the set of terminal symbols that appear as the first symbol of one or more strings derived from A.

    E.g. (grammar in prev. slide): ***FIRST***(*Expr')={+,-,ε}*, ***FIRST***(*Term')={*,/,ε}*, ***FIRST***(*Factor)={number, id}*
- **The LL(1) property**:
  - If $A \rightarrow a$ and $A \rightarrow b$ both appear in the grammar, we would like to have: $\mathbf{FIRST(a)} \cap \mathbf{FIRST(b)} = \emptyset$.
  - This would allow the parser to make a correct choice with a lookahead of exactly ***one*** symbol!

Parsing- LL(1) grammars

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar
- 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)

LL(1) grammars

- Predictive parsers (that require no back-tracking) can be constructed for a class of grammars called **LL(1)** grammars
  - L: Scanning input from left to right
  - L: produce a left-most derivation
  - "1": use 1 input symbol as lookahead at each step

- **LL(1)** grammars covers most programming constructs

- No ***ambiguous*** or ***left-recursive*** grammar can be **LL(1)**

LL(1) grammars

- A grammar **G** is **LL(1)** *iff* whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two distinct productions of **G** the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$. Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

LL(1) grammars

- A grammar **G** is **LL(1)** *iff* whenever **A → α** and **A → β** are two distinct productions of **G** the following conditions hold:

  Condition 1 &2: **First(α)** and **First(β)** are disjoint sets

  Condition 3: if ε is in **First(β)**, then **First(α)** and **Follow(A)** are disjoint sets (likewise if ε in First(**α**))

- For **LL(1)** grammars, predictive parsers can be constructed (by looking only at current input symbol production to apply for a non-terminal can be selected)

Predictive Parsing Table

- Predictive parsing table M[A,a], a two-dimensional array
  - **A**: non-terminal
  - **a**: terminal or **$** (input endmark)

Idea:

- Choose production **A → α** if the current input symbol is in **First(α)**
- Only issue is when **α= ε** (or ε can be derived from α)
  - Again choose **A → α** if the current input symbol is in **Follow(A)**, or if **$** is reached and **$** is in **Follow(A)**

Constructing Predictive Parsing table (using First/Follow)

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$.

2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

Example- Predictive Parsing Table

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E’						
T						
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E’		$E' \rightarrow +T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T						
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E’		$E' \rightarrow +T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$			$T \rightarrow F T'$		
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E’		$E' \rightarrow +T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F						

Example- Predictive Parsing Table

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

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E’		$E' \rightarrow +T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$			$T \rightarrow F T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Example- Predictive Parsing Table

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

- **Non-blanks indicate production to use to expand a non-terminal**
- **Blanks are error entries**

Parsing table, LL(1) grammar

- For every **LL(1)** grammar, each parse table entry uniquely identifies a production or signals an error

- For any grammar **G**, parse table can be constructed (**M** may have entries multiply defined)
  - E.g., if G is ambiguous or left-recursive, there will be at least one multiply defined entry in M !!

Non-recursive predictive parsing

- Non-recursive predictive parser built by maintaining an explicit stack

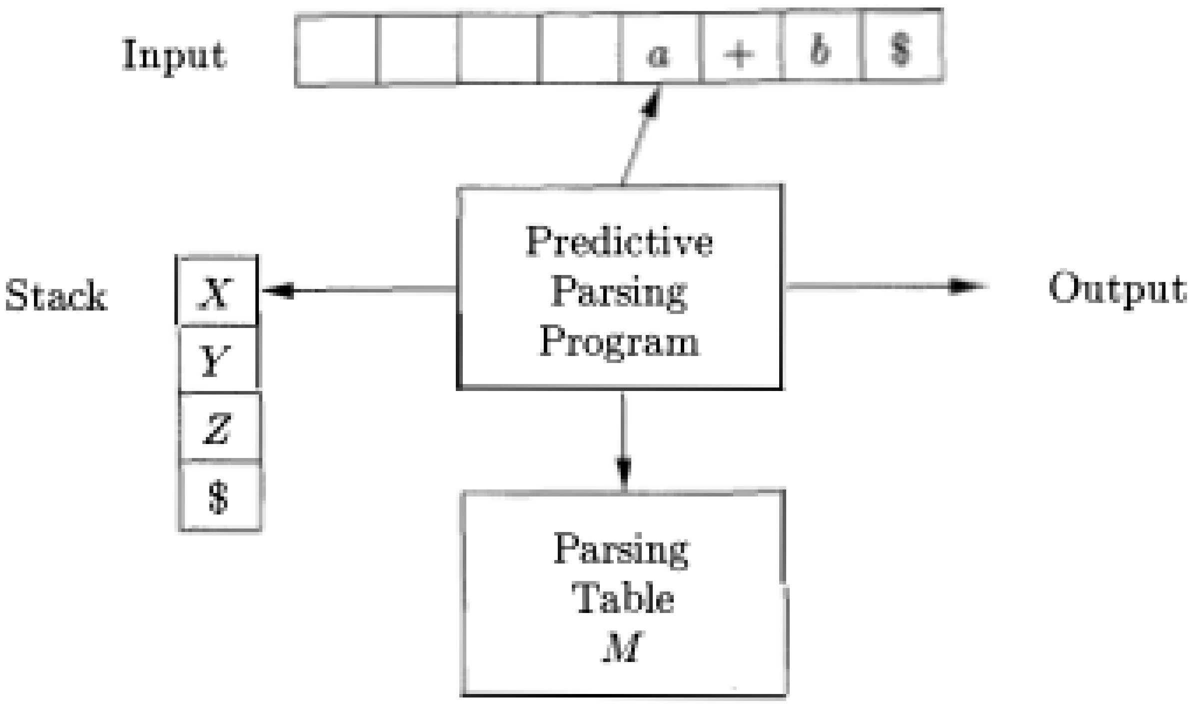
- Parser mimics a leftmost derivation

- Let "***w***" denote the sequence of input that has been matched so far. Then the stack holds a sequence of grammar symbols **α** such that:

$$S \Rightarrow^{*}_{lm} \mathbf{w\alpha}$$

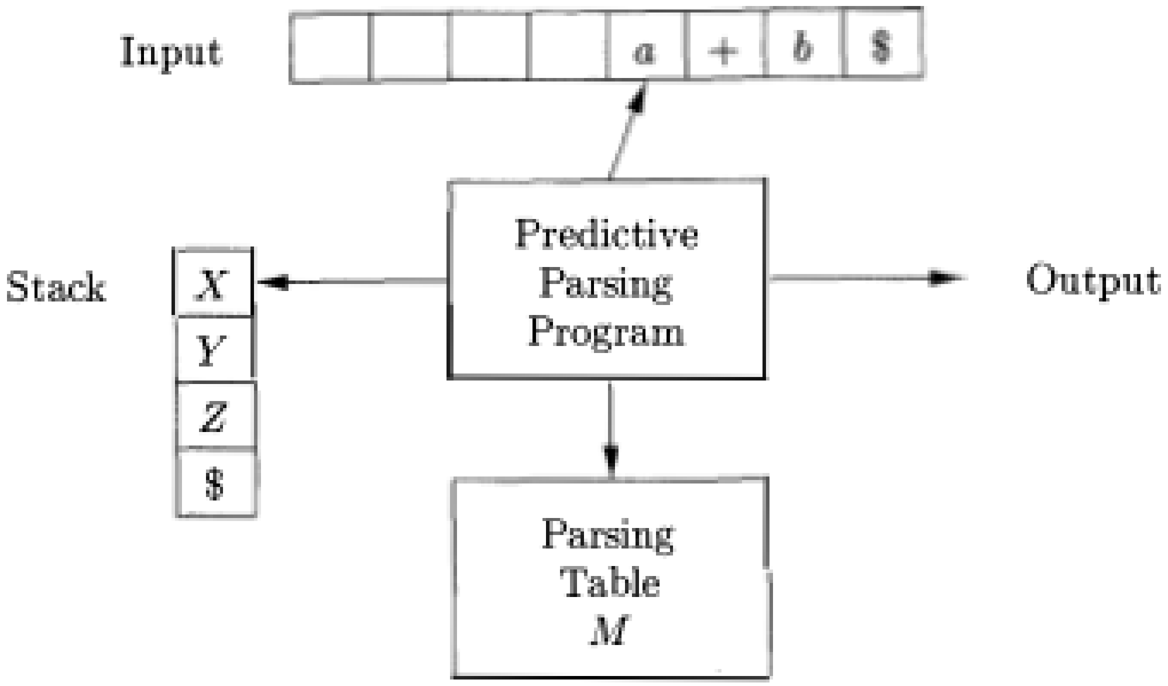
Non-recursive predictive parsing

- **Table driven parser**
  - **Input buffer** (input to be parsed followed by an end marker $)
  - **Stack** containing a sequence of grammar symbols ($ marks bottom of stack)
  - **Parsing table** (constructed using the prev. algo)



Non-recursive predictive parsing

- Take "**x**" symbol on top of the stack, and the current input "**a**".
  - X is non-terminal: Apply rule as per entry **M[X,a]**
  - X is terminal: Check for match between **X** and current input symbol **a**



Predictive parsing algorithm

- Behavior of parser can be described in terms of its configurations (**stack content** and **remaining input**)

- **Initial configuration**: stack containing S (**start symbol**) above **$**, remaining input is complete input (w$)

set ip to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
  if (X is a) pop the stack and advance ip;
  else if (X is a terminal) $error()$;
  else if ($M[X,a]$ is an error entry) $error()$;
  else if ($M[X,a] = X \to Y_1 Y_2 \cdots Y_k$) {
    output the production $X \to Y_1 Y_2 \cdots Y_k$;
    pop the stack;
    push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
  }
  set X to the top stack symbol;
}

Moves made by predictive parser for: id+id*id

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E’		$E' \rightarrow +T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Moves made by predictive parser for: id+id*id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\, TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\, TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\, FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\, FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Recap

- Ambiguity, left-recursion, left-factoring
- **Top-down parsing**
  - Recursive-Descent Parsing (with backtracking)
  - Predictive parsing – LL(1)

Exercise

- Construct the predictive parsing table for the below grammar:

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				