**Top-Down Parsing**

- **Goal:**
  - Find a leftmost derivation for an input string, or
  - Construct a parse tree for the input starting from the root and creating nodes of the parse tree in preorder (parent, then children)
- Discussed deterministic special case – predictive parsing – in 2.4.
- General case is nondeterministic (backtracking)
- Of more theoretical than practical interest

---

**Recursive-Descent Parsing**

- Requires backtracking
- Consider grammar
  
  
  \[ \begin{align*}
  S & \rightarrow cAd \\
  A & \rightarrow ab \mid a \\
  \end{align*} \]
- Parse input string \( w = cad \):

  \[ \begin{array}{c}
  c \\
  a \\
  d \\
  \end{array} \]

---

**Recursive-Descent Parsing**

- Requires backtracking
- Consider grammar
  
  
  \[ \begin{align*}
  S & \rightarrow cAd \\
  A & \rightarrow ab \mid a \\
  \end{align*} \]
- Parse input string \( w = cad \):

  \[ \begin{array}{c}
  c \\
  a \\
  d \\
  \end{array} \]
Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
- Parse input string \( w = cad \):

```
S
\( \rightarrow \)
c A d
\( \rightarrow \)
c a d
↓
```

Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
- Parse input string \( w = cad \):

```
S
\( \rightarrow \)
c A d
\( \rightarrow \)
c a d
↓
```

Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
- Parse input string \( w = cad \):

```
S
\( \rightarrow \)
c A d
\( \rightarrow \)
c a d
↓
```

Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
- Parse input string \( w = cad \):

```
S
\( \rightarrow \)
c A d
\( \rightarrow \)
c a d
↓
```

Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
- Parse input string \( w = cad \):

```
S
\( \rightarrow \)
c A d
\( \rightarrow \)
c a d
↓
```
Recursive-Descent Parsing

* Requires backtracking
* Consider grammar
  
  \[ S \rightarrow cAd \]
  
  \[ A \rightarrow ab | a \]

* Parse input string \( w = cad \):

  \[
  S \quad c \quad A \quad d \\
  \quad a \quad b \quad FAIL \\
  c \quad a \quad d \\
  \]

---

Recursive-Descent Parsing

* Requires backtracking
* Consider grammar
  
  \[ S \rightarrow cAd \]
  
  \[ A \rightarrow ab | a \]

* Parse input string \( w = cad \):

  \[
  S \quad c \quad A \quad d \\
  \quad c \quad a \quad d \\
  \]

---

Recursive-Descent Parsing

* Requires backtracking
* Consider grammar
  
  \[ S \rightarrow cAd \]
  
  \[ A \rightarrow ab | a \]

* Parse input string \( w = cad \):

  \[
  S \quad c \quad A \quad d \\
  \quad c \quad a \quad d \\
  \]
Recursive-Descent Parsing

- Requires backtracking
- Consider grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab | a \]
- Parse input string \( w = cad \):

\[
\begin{array}{c|c|c|c}
\text{symbol} & S & A & d \\
\hline
\text{c} & & & \\
\text{a} & & & \\
\text{d} & & & \\
\text{a} & & & \\
\text{d} & & & \\
\end{array}
\]

SUCCEED

Nonrecursive Predictive Parsing

- Maintain stack explicitly, instead of relying on runtime support for recursion.
- Components
  - Input buffer: \( w $ \)
  - Stack: terminals and nonterminals
  - Parsing table:
    - nonterminal \times input symbol \rightarrow production
  - Output stream: derivation

Nonrecursive Predictive Parsing

- Table \( M \) determines action based on stack symbol \( X \) and input symbol \( a \).
- Initial stack is start symbol on top of $.
- Possibilities are
  1. \( X = a = $ \): halt successfully
  2. \( X = a \neq $ \): pop \( X \) and advance input pointer
  3. \( X = \text{nonterminal} \): Consult table entry \( M[X, a] \).
     If empty, report error. Else pop \( X \) and push table entry.

Predictive Parsing Algorithm

set input pointer \( \pi \) to first symbol of \( w $ \\
repeat
  \text{Let } X \text{ be the top stack symbol and } a \text{ the symbol pointed to by } \pi \text{.}
  \text{if } X \text{ is a terminal or } $ \text{ then}
    \text{if } X = a \text{ then}
      \text{pop } X \text{ from the stack and advance } \pi
    \text{else error ()}
  \text{else /* } X \text{ is a nonterminal */}
    \text{if } M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k \text{ then begin}
      \text{pop } X \text{ from the stack}
      \text{push } Y_k Y_{k-1} \ldots Y_1 \text{ onto the stack with } Y_1 \text{ on top}
      \text{output the production } X \rightarrow Y_1 Y_2 \ldots Y_k
    \text{end}
    \text{else error ()}
  \text{end error ()}
until X = $ /* stack is empty */
Predictive Parsing Example

- Grammar (note elimination of left recursion):
  \[ E \to T E' 
  E' \to + T E' | \in 
  T \to F T' 
  T' \to \ast F T' | \in 
  F \to (E) | \text{id} \]

- Input: \text{id} + \text{id} * \text{id}

- Table:

| Nonterminal | Input Symbol | \in | \ast | ( | | |
|-------------|--------------|-----|-----|---|---|
| \text{E}    | T            | \in | \ast | ( | | |
| \text{T}    | F            | \in | \ast | ( | | |
| \text{F}    | (            | \in | \ast | ( | | |

FIRST and FOLLOW

- Recall FIRST from Chapter 2: \text{FIRST(\alpha)} is set of terminals that begin strings derived from \alpha.
- Together with FOLLOW, helps us build parse table from grammar.
- \text{FOLLOW(A)} is set of terminals \ast that can appear immediately to the right of \ast \text{A} in some sentential form; i.e., \ast \text{A} \Rightarrow \ast \text{A} \ast \beta.

Computing FIRST

1. If \text{X} is terminal, then \text{FIRST(X)} is \{X\}.
2. If \text{X} \Rightarrow \in \ast is a production, add \in \ast to \text{FIRST(X)}.
3. If \text{X} \Rightarrow Y_1 \ldots Y_k is a production, place \ast to \text{FIRST(X)} if for some i, \ast is in \text{FIRST(Y_i)} and \in \ast is in all of \text{FIRST(Y_1) \ldots FIRST(Y_k)}; that is, for all j \neq i, \text{Y_j} \Rightarrow \ast \in \ast. If \in \ast is in \text{FIRST(Y_i)} for all j = 1, 2, \ldots, k, then add \in \ast to \text{FIRST(X)}. For example, everything in \text{FIRST(Y_1')} is surely in \text{FIRST(X)}. If \text{Y_i} does not derive \in \ast, then we add nothing more to \text{FIRST(X)}, but if \text{Y_i} \Rightarrow \ast \in \ast, then we add \text{FIRST(Y_1')} and so on.

Computing FOLLOW

1. Place \$ in \text{FOLLOW(S)}, where S is the start symbol.
2. If there is a production \text{A} \Rightarrow \ast \text{B}, then everything in \text{FIRST(B)} except for \in \ast is placed in \text{FOLLOW(A)}.
3. If there is a production \text{A} \Rightarrow \ast \text{B} or a production \text{A} \Rightarrow \ast \text{B} \ast \text{C} \ast \text{D} \ast \text{E} \ast \text{F}, where \text{FIRST(B)} contains \in \ast (i.e., \ast \Rightarrow \in \ast \beta), then everything in \text{FOLLOW(A)} is in \text{FOLLOW(B)}.

Exercise: Compute FIRST, FOLLOW for nonterminals in grammar.
Construction of Predictive Parse Tables

**Input:** Grammar $G$  
**Output:** Parsing table $M$

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal $a$ in FIRST($\alpha$), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If $\varepsilon$ is in FIRST($\alpha$) and $\varepsilon$ is in FOLLOW($A$), add $A \rightarrow \alpha$ to $M[A, \varepsilon]$.
4. Make each undefined entry of $M$ be error.

LL(1) Grammars

- Ambiguous grammars will have more than one entry $M[A, a]$ for some nonterminal $A$, terminal $a$.
- E.g., ambiguous if / then / else grammar: 
  
  $S \rightarrow a\varepsilon \varepsilon$ [a]  
  $S \rightarrow \varepsilon$ [a]  
  $S \rightarrow \varepsilon$ [b]

- This grammar produces a table $M$ containing entry $M[S', \epsilon] = \{S' \rightarrow \epsilon, S' \rightarrow \epsilon S\}$ (because FOLLOW($S'$) = \{\epsilon, S\}).

LL(1) Grammars

- A grammar without such duplicate entries is called LL(1).
- First $L$ means “read input Left to right”.
- Second $L$ means “build Leftmost derivation”.
- 1 means one symbol of lookahead in input to make decisions.
- No ambiguous or left-recursive grammar can be LL(1).
- More technically: Grammar $G$ is LL(1) iff for $A \rightarrow \alpha | \beta$,
  1. For no terminal $a$ do both $\alpha$ and $\beta$ derive strings beginning with $a$.
  2. At most one of $\alpha$ and $\beta$ can derive the empty string.
  3. If $\beta \Rightarrow^* \varepsilon$, then $\alpha$ does not derive any string beginning with a terminal in FOLLOW($A$).
- So what to do when $M$ has multiply-defined entries?
  - Can try to make $G$ LL(1) by eliminating left recursion, and left factoring the result – may produce an LL(1) grammar.
  - Won’t work for some grammars, like our if / then / else example.
  - For such grammars, we may be able to eliminate all but one of the multiple entries; e.g., change $M[S', \epsilon] = \{S' \rightarrow \epsilon, S' \rightarrow \epsilon S\}$ to $M[S', \epsilon] = S' \rightarrow \epsilon S$.
  - But this must be done on a case-by case basis; there are no universal rules.