CS 335: Top-down Parsing

Swarnendu Biswas

Semester 2022-2023-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

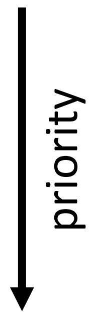
Example Expression Grammar

$$Start \rightarrow Expr$$

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$

$$Term \rightarrow Term \times Factor \mid Term \div Factor \mid Factor$$

$$Factor \rightarrow (Expr) \mid \mathbf{num} \mid \mathbf{name}$$



Derivation of $\text{name} + \text{name} \times \text{name}$

Sentential Form	Input
$Expr$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Expr + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Term + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Factor + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + Term$	$\text{name} \uparrow + \text{name} \times \text{name}$
$\text{name} + Term$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + Term \times Factor$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + Factor \times Factor$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{name} \times Factor$	$\text{name} + \uparrow \text{name} \times \text{name}$
$\text{name} + \text{name} \times Factor$	$\text{name} + \text{name} \uparrow \times \text{name}$
$\text{name} + \text{name} \times Factor$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \text{name} \uparrow$

input pointer

Derivation of $\text{name} + \text{name} \times \text{name}$

Sentential Form	Input
$Expr$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Expr + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Term + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$Factor + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + Term$	$\uparrow \text{name} + \text{name} \times \text{name}$
$\text{name} + Term$	$\text{name} \uparrow + \text{name} \times \text{name}$
$\text{name} + Term$	$\text{name} + \uparrow \text{name} \times \text{name}$

The current input terminal being scanned is called the lookahead symbol

$\text{name} + \text{name} \times Factor$	$\text{name} + \text{name} \uparrow \times \text{name}$
$\text{name} + \text{name} \times Factor$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \uparrow \text{name}$
$\text{name} + \text{name} \times \text{name}$	$\text{name} + \text{name} \times \text{name} \uparrow$

Derivation of $\mathrm{name} + \mathrm{name} \times \mathrm{name}$

$$\begin{tikzcd}
Start & \overset{lm}{\Rightarrow} & Expr & \overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & & \overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & & \overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & & \overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & \\
 & & & & Expr & + & Term & & Expr \arrow[d] & + & Term & & Expr \arrow[d] & + & Term & & Expr \arrow[d] & + & Term \\
 & & & & & & & & Term & & & & \mathrm{Term} \arrow[d] & & & & \mathrm{Term} \arrow[d] & & \\
 & & & & & & & & & & & & Factor & & & & Factor \arrow[d] & & \\
 & & & & & & & & & & & & & & & & \mathrm{name} & &
\end{tikzcd}$$

Derivation of $\mathbf{name} + \mathbf{name} \times \mathbf{name}$

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & \mathit{Expr} \arrow[dl] \arrow[d] \arrow[dr] & \\
& \mathit{Expr} \arrow[d] & + & \mathit{Term} \\
& \mathrm{Term} \arrow[d] & & \\
& \mathit{Factor} \arrow[d] & & \\
& \mathrm{name} & &
\end{tikzcd}$$

continued from previous slide

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & \mathit{Expr} \arrow[dl] \arrow[d] \arrow[drr] & & & \\
& \mathit{Expr} \arrow[d] & + & & \mathit{Term} \arrow[dl] \arrow[d] \arrow[dr] & \\
& \mathrm{Term} \arrow[d] & & \mathit{Term} & \times & \mathit{Factor} \\
& \mathit{Factor} \arrow[d] & & & & \\
& \mathrm{name} & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & \mathit{Expr} \arrow[dl] \arrow[d] \arrow[drr] & & & \\
& \mathit{Expr} \arrow[d] & + & & \mathit{Term} \arrow[dl] \arrow[d] \arrow[dr] & \\
& \mathrm{Term} \arrow[d] & & \mathit{Term} \arrow[d] & \times & \mathit{Factor} \\
& \mathit{Factor} \arrow[d] & & \mathit{Factor} & & \\
& \mathrm{name} & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & \mathit{Expr} \arrow[dl] \arrow[d] \arrow[drr] & & & \\
& \mathit{Expr} \arrow[d] & + & & \mathit{Term} \arrow[dl] \arrow[d] \arrow[dr] & \\
& \mathrm{Term} \arrow[d] & & \mathit{Term} \arrow[d] & \times & \mathit{Factor} \\
& \mathit{Factor} \arrow[d] & & \mathit{Factor} \arrow[d] & & \\
& \mathrm{name} & & \mathrm{name} & &
\end{tikzcd}$$

Derivation of $\mathbf{name} + \mathbf{name} \times \mathbf{name}$

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & & \\
& Expr \arrow[d] & + & Term \arrow[dl] \arrow[d] \arrow[dr] & \\
& \text{Term} \arrow[d] & Term \arrow[d] & \times & Factor \\
& Factor \arrow[d] & Factor \arrow[d] & & \\
& \text{name} & \text{name} & &
\end{tikzcd}$$

$$\begin{tikzcd}
\overset{lm}{\Rightarrow} & & Expr \arrow[dl] \arrow[d] \arrow[dr] & & \\
& Expr \arrow[d] & + & Term \arrow[dl] \arrow[d] \arrow[dr] & \\
& \text{Term} \arrow[d] & Term \arrow[d] & \times & Factor \arrow[d] \\
& Factor \arrow[d] & Factor \arrow[d] & & \text{name} \\
& \text{name} & \text{name} & &
\end{tikzcd}$$

General Idea of Top-down Parsing

Start with the root (i.e., start symbol) of the parse tree

Grow the tree downwards by expanding productions at the lower levels of the tree

- Select a nonterminal and extend it by adding children corresponding to the right side of some production for the nonterminal

Repeat till

- Lower fringe **consists only terminals and the input is consumed**

Top-down parsing finds a leftmost derivation for an input string

General Idea of Top-down Parsing

Start with the root of the parse tree

Grow the tree by expanding productions at the lower levels of the tree

- Extend a nonterminal by adding children corresponding to the right side of some production for the nonterminal

Repeat till

- Lower fringe consists only terminals and the input is consumed
- Mismatch in the lower fringe and the remaining input stream implies
  1. Wrong choice of productions while expanding nonterminals, selection of a production may involve trial-and-error
  2. Input character stream is not part of the language

Top-down Parsing Algorithm

```
root = node for the Start symbol
curr = root
push(null) // Stack
word = getNextWord()

while (true):
  if curr ∈ Nonterminal:
    pick next rule $A \longrightarrow \beta_1\beta_2 \ldots \beta_n$ to expand curr
    create nodes for $\beta_1$, $\beta_2$, …, $\beta_n$ as children of curr
    push($\beta_n$,$\beta_{n-1}$,…, $\beta_1$)
    curr = $\beta_1$
  if curr == word:
    word = getNextWord()
    curr = pop() // consumed
  if word == eof and curr == null:
    accept input
  else
    backtrack()
```

Implementing Backtracking

- A large subset of CFGs can be parsed without backtracking
    - The grammar may require transformations

- Steps in backtracking
    - Set `curr` to parent and delete the children
    - Expand the node `curr` with **untried rules** if any
        - Create child nodes for each symbol in the right hand of the production
        - Push those symbols onto the stack in reverse order
        - Set `curr` to the first child node
    - **Move** `curr` **up the tree** if there are no untried rules
    - Report a syntax error when there are no more moves

Cost of Backtracking

Backtracking is expensive

- Parser expands a nonterminal with the wrong rule
- Mismatch between the lower fringe of the parse tree and the input is detected
- Parser undoes the last few actions
- Parser tries other productions if any

Derivation of name + name × name

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	$Term \rightarrow Factor$
7	$Factor \rightarrow (Expr)$
8	$Factor \rightarrow$ num
9	$Factor \rightarrow$ name

Rule #	Sentential Form	Input
	$Expr$	↑ name + name × name
1	$Expr + Term$	↑ name + name × name
3	$Term + Term$	↑ name + name × name
6	$Factor + Term$	↑ name + name × name
9	name + $Term$	↑ name + name × name
	name + $Term$	name ↑ +name × name
	name + $Term$	name +↑ name × name
4	name + $Term \times Factor$	name +↑ name × name
6	name + $Factor \times Factor$	name +↑ name × name
9	name + name × $Factor$	name +↑ name × name
	name + name × $Factor$	name + name ↑× name
	name + name × $Factor$	name + name ×↑ name
9	name + name × name	name + name ×↑ name
	name + name × name	name + name × name ↑

Derivation of name + name × name

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	[illegible]
7	[illegible]
8	$Factor \rightarrow$ num
9	$Factor \rightarrow$ name

Rule #	Sentential Form	Input
	$Expr$	↑ name + name × name
1	$Expr + Term$	↑ name + name × name
3	$Term + Term$	↑ name + name × name
6	$Factor + Term$	↑ name + name × name
9	name + $Term$	↑ name + name × name
	name + $Term$	name ↑ +name × name
[illegible]	[illegible]	× name
[illegible]	[illegible]	× name
6	name + $Factor \times Factor$	name +↑ name × name
9	name + name × $Factor$	name +↑ name × name
	name + name × $Factor$	name + name ↑× name
	name + name × $Factor$	name + name ×↑ name
9	name + name × name	name + name ×↑ name
	name + name × name	name + name × name ↑

How does a top-down parser choose which rule to apply?

Selecting a Production

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	$Term \rightarrow Factor$
7	
8	
9	

Rule #	Sentential Form	Input
	$Expr$	$\uparrow$ name + name × name
1	$Expr + Term$	$\uparrow$ name + name × name
1	$Expr + Term + Term$	$\uparrow$ name + name × name
1	$Expr + Term + Term + \cdots$	$\uparrow$ name + name × name
1	…	$\uparrow$ name + name × name
1	…	$\uparrow$ name + name × name

A top-down parser can loop indefinitely with left-recursive grammar

Left Recursion

- A grammar is left-recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - **Direct** left recursion: There is a production of the form $A \rightarrow A\alpha$
  - **Indirect** left recursion: First symbol on the right-hand side of a rule can derive the symbol on the left

We can often reformulate a grammar to avoid left recursion

Remove Direct Left Recursion

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \ldots | \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon$$

Remove Direct Left Recursion

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

$\Rightarrow$

$E \rightarrow TE'$
$E' \rightarrow +TE'$
$T \rightarrow FT'$
$T' \rightarrow * FT'$
$F \rightarrow (E) \mid \mathbf{id}$

Non-Left-Recursive Expression Grammar

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Expr + Term$
2	$Expr \rightarrow Expr - Term$
3	$Expr \rightarrow Term$
4	$Term \rightarrow Term \times Factor$
5	$Term \rightarrow Term \div Factor$
6	$Term \rightarrow Factor$
7	$Factor \rightarrow (Expr)$
8	$Factor \rightarrow$ num
9	$Factor \rightarrow$ name

Rule #	Production
0	$Start \rightarrow Expr$
1	$Expr \rightarrow Term\ Expr'$
2	$Expr' \rightarrow + Term\ Expr'$
3	$Expr' \rightarrow - Term\ Expr'$
4	$Expr' \rightarrow \epsilon$
5	$Term \rightarrow Factor\ Term'$
6	$Term' \rightarrow \times Factor\ Term'$
7	$Term' \rightarrow \div Factor\ Term'$
8	$Term' \rightarrow \epsilon$
9	$Factor \rightarrow (Expr)$
10	$Factor \rightarrow$ num
11	$Factor \rightarrow$ name

Indirect Left Recursion

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- There is a left recursion because $S \rightarrow Aa \rightarrow Sda$

Eliminating Left Recursion

- **Input**: Grammar G with no cycles or ϵ-productions
- **Algorithm**

```
Arrange nonterminals in some order $A_1, A_2, \dots, A_n$
for $i \leftarrow 1 \dots n$
    for $j \leftarrow 1$ to $i - 1$
        If ∃ a production $A_i \rightarrow A_j\gamma$
            Replace $A_i \rightarrow A_j\gamma$ with one or more productions that expand $A_j$
    Eliminate the immediate left recursion among the $A_i$ productions
```

Loop invariant at the start of outer iteration i

$$\forall k < i, \text{no production expanding } A_k \text{ has } A_l \text{ in its righthand side for all } l < k$$

Eliminating Indirect Left Recursion

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

$\Rightarrow$

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

Avoid Backtracking

- Parser is to select the next rule
    - Compare the `curr` symbol and the next input symbol called the lookahead
    - Use the lookahead to disambiguate the possible production rules

- Backtrack-free grammar is a CFG for which a leftmost, top-down parser can always predict the correct rule with one word lookahead
    - Also called a predictive grammar

FIRST Set

- **Intuition**
  - Each alternative for the leftmost nonterminal leads to a **distinct** terminal symbol
  - Which rule to choose becomes obvious by comparing the **next word** in the input stream

- Given a string γ of terminal and nonterminal symbols, FIRST*(γ)* is the set of all terminal symbols that can begin any string derived from γ
  - We also need to keep track of which symbols can produce the empty string
  - $\text{FIRST}: (NT \cup T \cup \{\epsilon, \text{EOF}\}) \rightarrow (T \cup \{\epsilon, \text{EOF}\})$

Steps to Compute FIRST Set

1. If X is a terminal, then $\mathrm{FIRST}(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production, then $\epsilon \in \mathrm{FIRST}(X)$
3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production
   - I. Everything in $\mathrm{FIRST}(Y_1)$ is in $\mathrm{FIRST}(X)$
   - II. If for some i, $a \in \mathrm{FIRST}(Y_i)$ and $\forall 1 \leq j < i$, $\epsilon \in \mathrm{FIRST}(Y_j)$, then $a \in \mathrm{FIRST}(X)$
   - III. If $\epsilon \in \mathrm{FIRST}(Y_1 \ldots Y_k)$, then $\epsilon \in \mathrm{FIRST}(X)$

FIRST Set

- Generalize FIRST relation to string of symbols

$$\text{FIRST}(X\gamma) \rightarrow \text{FIRST}(X) \text{ if } X \nrightarrow \epsilon$$

$$\text{FIRST}(X\gamma) \rightarrow \text{FIRST}(X) \cup \text{FIRST}(\gamma) \text{ if } X \rightarrow \epsilon$$

Compute FIRST Set

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +Term\ Expr'$

$\quad |-Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term'$

$\quad |\div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid \mathrm{num} \mid \mathrm{name}$

$\mathrm{FIRST}(Expr) = \{(, \mathrm{name}, \mathrm{num}\}$

$\mathrm{FIRST}(Expr') = \{+, -, \epsilon\}$

$\mathrm{FIRST}(Term) = \{(, \mathrm{name}, \mathrm{num}\}$

$\mathrm{FIRST}(Term') = \{\epsilon, \times, \div\}$

$\mathrm{FIRST}(Factor) = \{(, \mathrm{name}, \mathrm{num}\}$

FOLLOW Set

- $\mathrm{FOLLOW}(X)$ is the set of terminals that can immediately follow X
    - That is, $t \in \mathrm{FOLLOW}(X)$ if there is any derivation containing Xt

$$\begin{tikzcd}
 & S \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\alpha & A \arrow[dl] \arrow[dr] & a & \beta \\
c & \dots & \gamma &
\end{tikzcd}$$

Terminal c is in $\mathrm{FIRST}(A)$ and a is in $\mathrm{FOLLOW}(A)$

Steps to Compute FOLLOW Set

1. Place \$ in $\mathrm{FOLLOW}(S)$ where S is the start symbol and \$ is the end marker
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\mathrm{FIRST}(\beta)$ except ϵ is in $\mathrm{FOLLOW}(B)$
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where $\mathrm{FIRST}(\beta)$ contains ϵ, then everything in $\mathrm{FOLLOW}(A)$ is in $\mathrm{FOLLOW}(B)$

Compute FOLLOW Set

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +Term\ Expr'$

$\quad\quad |-Term\ Expr' \mid \epsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term'$

$\quad\quad |\div\ Factor\ Term' \mid \epsilon$

$Factor \rightarrow (Expr) \mid \mathrm{num} \mid \mathrm{name}$

$\mathrm{FOLLOW}(Expr) = \{\$,)\}$

$\mathrm{FOLLOW}(Expr') = \{\$,)\}$

$\mathrm{FOLLOW}(Term) = \{\$, +, -,)\}$

$\mathrm{FOLLOW}(Term') = \{\$, +, -,)\}$

$\mathrm{FOLLOW}(Factor) = \{\$, +, -, \times, \div,)\}$

Conditions for Backtrack-Free Grammar

- Consider a production $A \rightarrow \beta$

$$\text{FIRST}^{+} = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

- For any nonterminal A where $A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$, a backtrack-free grammar has the property

$$\text{FIRST}^{+}(A \rightarrow \beta_i) \cap \text{FIRST}^{+}\left(A \rightarrow \beta_j\right) = \phi, \qquad \forall 1 \leq i, j \leq n, i \neq j$$

Backtracking

$Start \rightarrow Expr$

$Expr \rightarrow Term Expr'$

$Expr' \rightarrow +Term Expr'$

$\quad |-Term Expr' \mid \epsilon$

$Term \rightarrow Factor Term'$

$Term' \rightarrow \times Factor Term'$

$\quad |\div Factor Term' \mid \epsilon$

$Factor \rightarrow (\ Expr\ ) \mid \mathrm{num}$

$Factor \rightarrow$ name

$\quad |$ name [$Arglist$]

$\quad |$ name ($Arglist$)

$Arglist \rightarrow Expr\ \ MoreArgs$

$MoreArgs \rightarrow , Expr\ MoreArgs$

$\quad | \ \epsilon$

Not all grammars are backtrack free

Left Factoring

- Left factoring is the process of extracting and isolating common prefixes in a set of productions

$$Factor \rightarrow name\ Arguments$$
$$Arguments \rightarrow [\ ArgList\ ]\ |\ (\ ArgList\ )\ |\ \epsilon$$

- Algorithm

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ \dots\ |\alpha\beta_n|\gamma_1|\gamma_2|\ \dots\ |\gamma_j$$

$$\Downarrow$$

$$A \rightarrow \alpha B|\gamma_1|\gamma_2|\ \dots\ |\gamma_j$$
$$B \rightarrow \beta_1|\beta_2|\ \dots\ |\beta_n$$

Key Insight in Using Top-Down Parsing

- Efficiency depends on the accuracy of selecting the correct production for expanding a nonterminal
  - Parser may not terminate in the worst case
- A large subset of the context-free grammars can be parsed without backtracking

Recursive-Descent Parsing

Recursive-Descent Parsing

- Recursive-descent parsing is a form of top-down parsing that **may** require backtracking
- Consists of a set of procedures, one for each nonterminal

```
void A() {
    Choose an A-production $A \rightarrow X_1 X_2 \dots X_k$
    for $i \leftarrow 1 \dots k$
        if $X_i$ is a nonterminal
            call procedure $X_i()$
        else if $X_i$ equals the current input symbol $a$
            advance the input to the next symbol
        else
            // error
}
```

Limitations with Recursive-Descent Parsing

- Consider a grammar with two productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$
- Suppose $\text{FIRST}(\gamma_1) \cap \text{FIRST}(\gamma_2) \neq \phi$
    - Say a is the common terminal symbol
- Function corresponding to X will not know which production to use on input token a

Recursive-Descent Parsing with Backtracking

- To support backtracking
    - All productions should be tried in some order
    - Failure for some production implies we need to try remaining productions
    - Report an error only when there are no other rules

Predictive Parsing

- Special case of recursive-descent parsing that does not require backtracking
  - Lookahead symbol unambiguously determines which production rule to use
  - Advantage is that the algorithm is simple and the parser can be constructed by hand

$$
\begin{aligned}
stmt &\rightarrow \textbf{expr} \; ; \\
&\quad \mid \textbf{if} \; (\; expr \;) stmt \\
&\quad \mid \textbf{for} \; (\; optexpr \; ; \; optexpr \; ; \; optexpr \;) \; stmt \\
&\quad \mid \textbf{other} \\
optexpr &\rightarrow \epsilon \mid \textbf{expr}
\end{aligned}
$$

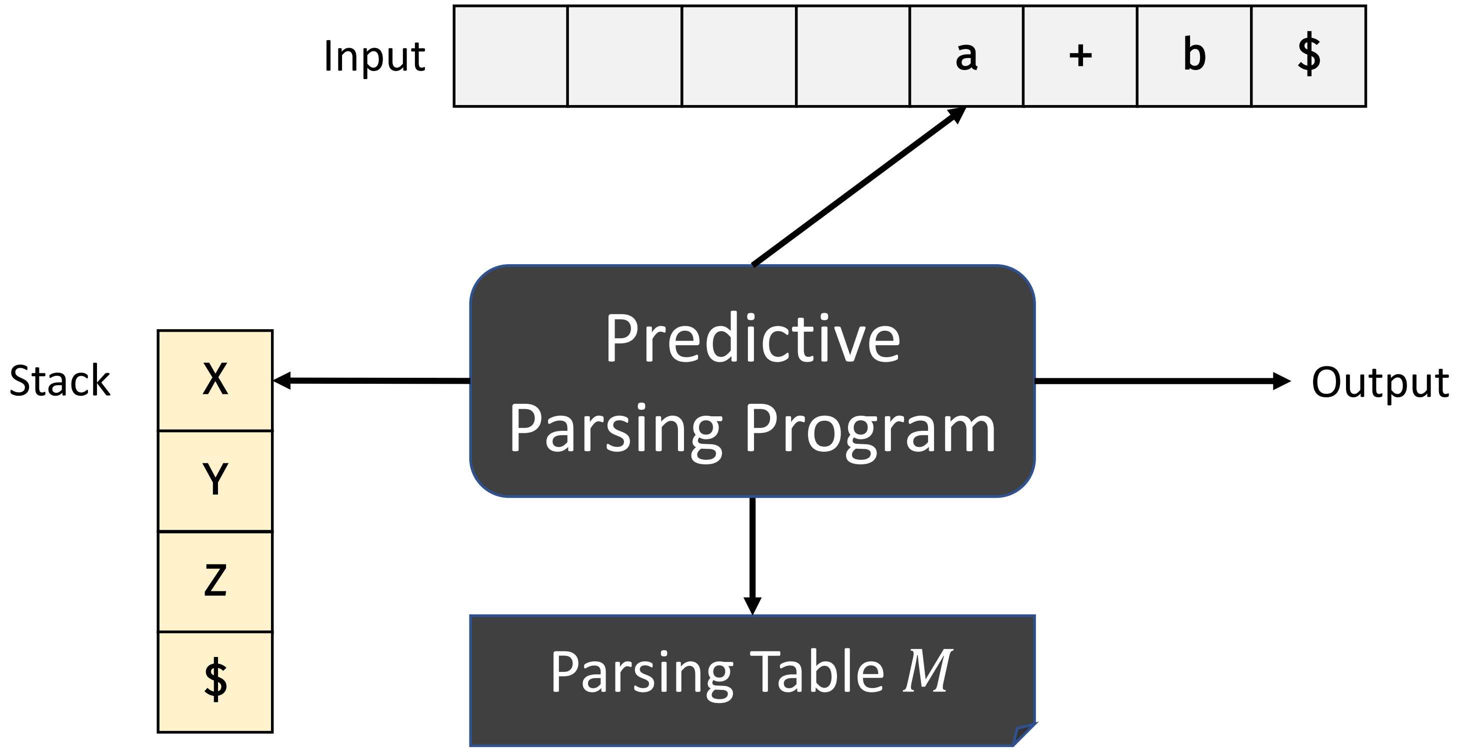
Pseudocode for a Predictive Parser

```
void stmt() {
  switch(lookahead) {
    case expr:
      match(expr); match(‘;’); break;
    case if:
      match(if); match(‘(‘); match(expr); match(‘)’); stmt(); break;
    case for:
      match(for); match(‘(‘); optexpr(); match(‘;’); optexpr();
      match(‘;’); optexpr(); match(‘)’); stmt(); break;
    case other:
      match(other); break;
    default:
      report(“syntax error”);
  }
}
```

LL(1) Grammars

- Class of grammars for which no backtracking is required
  - First L stands for left-to-right scan, second L stands for leftmost derivation
  - There is one lookahead token
- In LL(k), k stands for k lookahead tokens
  - Predictive parsers accept LL(k) grammars
  - Every LL(1) grammar is a LL(2) grammar

Nonrecursive Table-Driven Predictive Parser



Predictive Parsing Algorithm

- **Input**: String w and parsing table M for grammar G
- **Algorithm**:

```
Let $a$ be the first symbol in $w$
Let $X$ be the symbol at the top of the stack
while $X \neq \$$:
    if $X == a$:
        pop the stack and advance the input
    else if $X$ is a terminal or $M[X, a]$ is an error entry:
        error
    else if $M[X, a] == X \rightarrow Y_1 Y_2 \dots Y_k$:
        output the production
        pop the stack
        push $Y_k Y_{k-1} \dots Y_1$ onto the stack
    $X \leftarrow$ top stack symbol
```

Construction of a Predictive Parsing Table

- **Input**: Grammar G

- **Algorithm**:
  - For each production $A \to \alpha$ in G,
    - For each terminal a in $\mathrm{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$
    - If ϵ is in $\mathrm{FIRST}(\alpha)$, then for each terminal b in $\mathrm{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$
    - If ϵ is in $\mathrm{FIRST}(\alpha)$ and \$ is in $\mathrm{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, \$]$
    - No production in $M[A, a]$ indicates error

Predictive Parsing Table

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{\epsilon, \times\}$

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\$,)\}$

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\$, +,)\}$

$\text{FOLLOW}(F) = \{\$, +, \times,)\}$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow * FT' \mid \epsilon$
$F \rightarrow (E) \mid \textbf{id}$

Nonterminal	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Working of Predictive Parser

Matched	Stack	Input	Action
	$E\$$	**id** $+$ **id** $*$ **id**$\$$	
	$TE'\$$	**id** $+$ **id** $*$ **id**$\$$	Output $E \rightarrow TE'$
	$FT'E'\$$	**id** $+$ **id** $*$ **id**$\$$	Output $T \rightarrow FT'$
	id$T'E'\$$	**id** $+$ **id** $*$ **id**$\$$	Output $F \rightarrow$ **id**
id	$T'E'\$$	$+$**id** $*$ **id**$\$$	Match **id**
id	$E'\$$	$+$**id** $*$ **id**$\$$	Output $T' \rightarrow \epsilon$
id	$+TE'\$$	$+$**id** $*$ **id**$\$$	Output $E' \rightarrow +TE'$
id $+$	$TE'\$$	**id** $*$ **id**$\$$	Match $+$
id $+$	$FT'E'\$$	**id** $*$ **id**$\$$	Output $T \rightarrow FT'$
id $+$	**idT**$'E'\$$	**id** $*$ **id**$\$$	Output $F \rightarrow$ **id**

Working of Predictive Parser

Matched	Stack	Input	Action
…			
id $+$	**id**$T'E'\$$	**id** $*$ **id**$\$$	Output $F \rightarrow$ **id**
id $+$ **id**	$T'E'\$$	$*$ **id**$\$$	Match **id**
id $+$ **id**	$*\,FT'E'\$$	$*$ **id**$\$$	Output $T' \rightarrow * \, FT'$
id $+$ **id**$*$	$FT'E'\$$	**id**$\$$	Match $*$
id $+$ **id**$*$	**id**$T'E'\$$	**id**$\$$	Output $F \rightarrow$ **id**
id $+$ **id**$*$**id**	$T'E'\$$	$\$$	Match **id**
id $+$ **id**$*$**id**	$E'\$$	$\$$	Output $T' \rightarrow \epsilon$
id $+$ **id**$*$**id**	$\$$	$\$$	Output $E' \rightarrow \epsilon$

Predictive Parsing

- Grammars whose predictive parsing tables contain no duplicate entries are called LL(1)
  - No left-recursive or ambiguous grammar can be LL(1)
- If grammar G is left-recursive or is ambiguous, then parsing table M will have at least one multiply-defined cell
- Some grammars cannot be transformed into LL(1)
  - The adjacent grammar is ambiguous

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

Predictive Parsing Table

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

Nonterminal	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$		$T \rightarrow FT'$		

What do we do when we see an `else`?

Error Recovery in Predictive Parsing

- Error conditions
  - Terminal on top of the stack does not match the next input symbol
  - Nonterminal A is on top of the stack, a is the next input symbol, and $M[A, a]$ is error
- Choices
  1. Raise an error and quit parsing
  2. Print an error message, try to recover from the error, and continue with compilation

Error Recovery in Predictive Parsing

- Panic mode – skip over symbols until a token in a set of synchronizing (synch) tokens appears
  - Add all tokens in $\mathrm{FOLLOW}(A)$ to the synch set for A, parsing can continue if the parser sees an input symbol in $\mathrm{FOLLOW}(A)$
  - Add symbols in $\mathrm{FIRST}(A)$ to the synch set for A, parsing can continue with the nonterminal A that is at the top of the stack
  - Add keywords that can begin constructs
  - …
- Other error handling policies
  - Skip input if the table does not have an entry
  - Pop nonterminal if the table entry is synch

Predictive Parsing Table with Synchronizing Tokens

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\$,)\}$

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\$, +,)\}$

$\text{FOLLOW}(F) = \{\$, +, \times,)\}$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow * FT' \mid \epsilon$
$F \rightarrow (E) \mid \textbf{id}$

Nonterminal	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Error Recover Moves by Predictive Parser

Stack	Input	Remark
$E\$$	$+\mathbf{id} * +\mathbf{id}\$$	Entry is blank, Error, skip $+$
$E\$$	$\mathbf{id} * +\mathbf{id}\$$	
$TE'\$$	$\mathbf{id} * +\mathbf{id}\$$	
$FTE'\$$	$\mathbf{id} * +\mathbf{id}\$$	
$\mathbf{id}TE'\$$	$\mathbf{id} * +\mathbf{id}\$$	
$T'E'\$$	$* +\mathbf{id}\$$	
$* FT'E'\$$	$* +\mathbf{id}\$$	
$FT'E'\$$	$+\mathbf{id}\$$	Error, $M[F, +] = \text{synch}$, pop F
$T'E'\$$	$+\mathbf{id}\$$	
$E'\$$	$+\mathbf{id}\$$	

Error Recover Moves by Predictive Parser

Stack	Input	Remark
$E'\$$	$+\mathbf{id}\$$	…continuation
$+TE'\$$	$+\mathbf{id}\$$	
$TE'\$$	$\mathbf{id}\$$	
$FT'E'\$$	$\mathbf{id}\$$	
$\mathbf{id}T'E'\$$	$\mathbf{id}\$$	
$T'E'\$$	$\$$	
$E'\$$	$\$$	
$\$$	$\$$	

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 4.4.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 3.3.