CS 335: Top-Down Parsing

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

Example Expression Grammar

$$
\begin{aligned}
\mathit{Start} &\rightarrow \mathit{Expr} \\
\mathit{Expr} &\rightarrow \mathit{Expr} + \mathit{Term} \mid \mathit{Expr} - \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} &\rightarrow \mathit{Term} \times \mathit{Factor} \mid \mathit{Term} \div \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} &\rightarrow (\mathit{Expr}) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

$\downarrow$ priority

Derivation of **name** + **name** $\times$ **name** with Oracular Knowledge

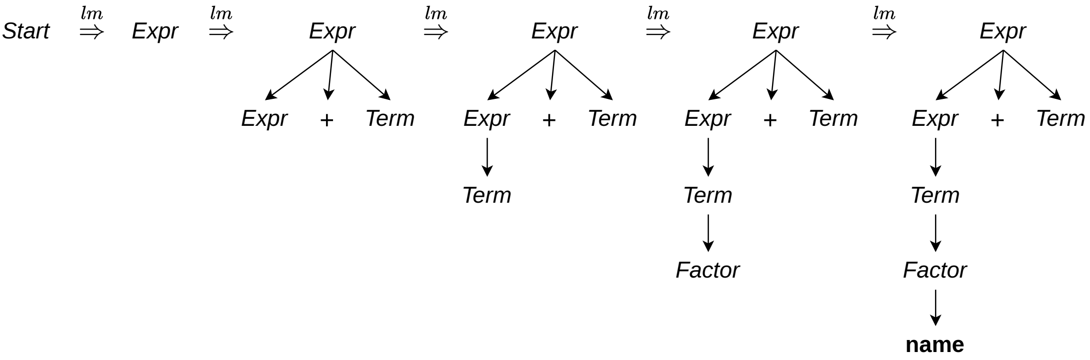
Sentential Form	Input
Expr	$\uparrow$ **name** + **name** $\times$ **name**
Expr + *Term*	$\uparrow$ **name** + **name** $\times$ **name**
Term + *Term*	$\uparrow$ **name** + **name** $\times$ **name**
Factor + *Term*	$\uparrow$ **name** + **name** $\times$ **name**
name + *Term*	$\uparrow$ **name** + **name** $\times$ **name**
name + *Term*	**name** $\uparrow$ + **name** $\times$ **name**
name + *Term*	**name** + $\uparrow$ **name** $\times$ **name**
name + *Term* $\times$ *Factor*	**name** + $\uparrow$ **name** $\times$ **name**
name + *Factor* $\times$ *Factor*	**name** + $\uparrow$ **name** $\times$ **name**
name + **name** $\times$ *Factor*	**name** + $\uparrow$ **name** $\times$ **name**
name + **name** $\times$ *Factor*	**name** + **name** $\uparrow$ $\times$ **name**
name + **name** $\times$ *Factor*	**name** + **name** $\times$ $\uparrow$ **name**
name + **name** $\times$ **name**	**name** + **name** $\times$ $\uparrow$ **name**
name + **name** $\times$ **name**	**name** + **name** $\times$ **name** $\uparrow$

Derivation of **name** + **name** × **name** with Oracular Knowledge

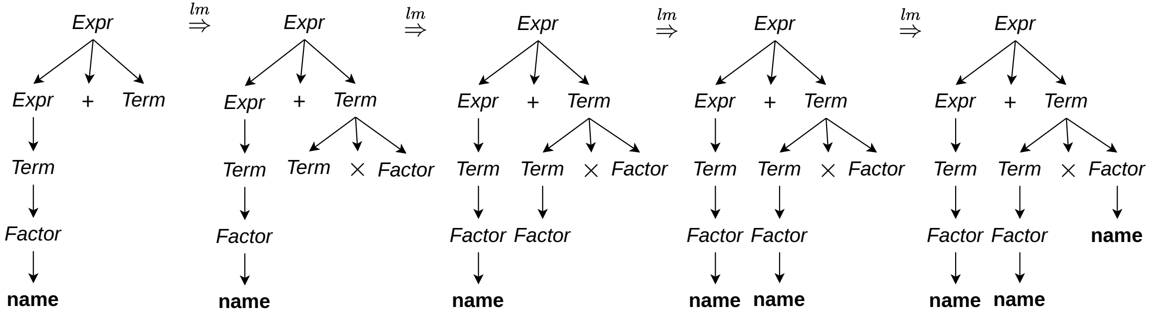
Sentential Form	Input
Expr	↑ **name** + **name** × **name**
Expr + *Term*	↑ **name** + **name** × **name**
Term + *Term*	↑ **name** + **name** × **name**
Factor + *Term*	↑ **name** + **name** × **name**
name + *Term*	↑ **name** + **name** × **name**
name + *Factor* × *Factor*	**name** + ↑ **name** × **name**
name + **name** × *Factor*	**name** + ↑ **name** × **name**
name + **name** × *Factor*	**name** + **name** ↑ × **name**
name + **name** × *Factor*	**name** + **name** × ↑ **name**
name + **name** × **name**	**name** + **name** × ↑ **name**
name + **name** × **name**	**name** + **name** × **name** ↑

The current input terminal being scanned is called the lookahead symbol

Derivation of **name** + **name** × **name** with Oracular Knowledge



Derivation of **name** + **name** × **name** with Oracular Knowledge



Top-Down Parsing

High-level idea in top-down parsing

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

(ii) Grow the tree downwards by expanding the production 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

(iii) Repeat till the lower fringe consists only of terminals and the input is consumed

- Top-down parsing finds a **leftmost derivation** for an input string
- Expands the parse tree with a **preorder depth-first** traversal

Top-Down Parsing

High-level idea in top-down parsing

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

(ii) Grow the tree downwards by expanding the production 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

(iii) Repeat till the lower fringe consists only of terminals and the input is consumed

Mismatch in the lower fringe and the remaining input stream implies

(i) Wrong choice of productions while expanding nonterminals, selection of a production may involve trial-and-error

(ii) 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 $A \rightarrow \beta_1\beta_2\ldots\beta_n$ to expand curr
    create nodes for $\beta_1, \beta_2, \ldots \beta_n$ as children of curr
    push($\beta_n\beta_{n-1}\ldots\beta_1$) // reverse order
    curr = $\beta_1$
  if curr == word
    word = getNextWord()
    curr = pop() // Consumed
  if word == EOF and curr == null
    accept input
  else
    backtrack
```

Derivation of **name** + **name** × **name**

Rule #	Production
0	*Start* → *Expr*
1	*Expr* → *Expr* + *Term*
2	*Expr* → *Expr* − *Term*
3	*Expr* → *Term*
4	*Term* → *Term* × *Factor*
5	*Term* → *Term* ÷ *Factor*
6	*Term* → *Factor*
7	*Factor* → (*Expr*)
8	*Factor* → **num**
9	*Factor* → **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* × *Factor*	**name** + ↑ **name** × **name**
4	**name** + *Factor* × *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* → *Expr*
1	*Expr* → *Expr* + *Term*
2	*Expr* → *Expr* – *Term*
3	*Expr* → *Term*
4	*Term* → *Term* × *Factor*
5	*Ter*
6	*Ter*
7	*Fac*
8	*Factor* → **num**
9	*Factor* → **name**

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

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

Deterministically Selecting a Production in 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 #	Sentential Form	Input
	$Expr$	$\uparrow$ **name** + **name** $\times$ **name**
1	$Expr + Term$	$\uparrow$ **name** + **name** $\times$ **name**
1	$Expr + Term + Term$	$\uparrow$ **name** + **name** $\times$ **name**
1	$Expr + Term + Term + \ldots$	$\uparrow$ **name** + **name** $\times$ **name**
1	$\ldots$	$\uparrow$ **name** + **name** $\times$ **name**
1	$\ldots$	$\uparrow$ **name** + **name** $\times$ **name**

Deterministically Selecting a Production in 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* × *Factor*
5	*Ter*
6	*Ter*
7	*Fac*
8	*Factor* $\rightarrow$ **num**
9	*Factor* $\rightarrow$ **name**

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* + ...	$\uparrow$ **name** + **name** × **name**
1		$\uparrow$ **name** + **name** × **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 $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α

Direct There is a production of the form $A \rightarrow A\alpha$

Indirect The first symbol on the right-hand side of a rule can derive the symbol on the left

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow Ac \mid Sd \mid \epsilon
\end{aligned}
$$

We can often reformulate a grammar to avoid left recursion

Remove Direct Left Recursion

Grammar with left recursion

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

Grammar without left recursion

$$A \rightarrow \beta_1 A^{'} \mid \beta_2 A^{'} \mid \ldots \mid \beta_n A^{'}$$
$$A^{'} \rightarrow \alpha_1 A^{'} \mid \alpha_2 A^{'} \mid \ldots \mid \alpha_m A^{'} \mid \epsilon$$

Example

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned} \quad \Longrightarrow \quad \begin{aligned} E &\rightarrow TE^{'} \\ E^{'} &\rightarrow +TE^{'} \\ T &\rightarrow FT^{'} \\ T^{'} &\rightarrow *FT^{'} \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

Non-Left-Recursive Expression Grammar

Expression Grammar with Recursion

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**

Expression Grammar without Recursion

Rule #	Production
0	$Start \rightarrow Expr$
1	$Start \rightarrow Term\ Expr'$
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**

Eliminating Indirect Left Recursion

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

```
Arrange nonterminals in some order A_1, A_2, ... A_n
for i ← 1 ... n
  for j ← 1 ... i − 1
    if ∃ a production A_i → A_j γ
      Replace A_i → A_j γ 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 the outer iteration i

$\forall k < i$, no production expanding A_k has A_l in its body (i.e., right-hand side) for all $l < k$

The algorithm establishes a topological ordering on nonterminals

Eliminating Indirect Left Recursion

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

```
Arrange nonterminals in some order A_1, A_2, ... A_n
for i ← 1 ... n
  for j ← 1 ... i − 1
    if ∃ a production A_i → A_j γ
      Replace A_i → A_j γ with one or more productions that expand A_j
  Eliminate the immediate left recursion among the A_i productions
```

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

$$\Longrightarrow$$

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

Implementing Backtracking

- A top-down parser may need to undo its actions after it detects a mismatch between the parse tree's leaves and the input
  - Implies a possible expansion with a wrong production

- 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

Backtracking is Expensive

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

A large subset of CFGs can be parsed without backtracking

The grammar may require transformations

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

Definition

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

FIRST Set

Definition

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
- 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 FIRST $(X) = \{X\}$
  2. If $X \rightarrow \epsilon$ is a production, then $\epsilon \in$ FIRST (X)
  3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then
     (i) FIRST (X) = FIRST (Y_1) provided $Y_1 \nrightarrow \epsilon$
     (ii) If for some $i \leq k$ and $1 \leq j < i$, $a \in$ FIRST (Y_i), and $\forall j, \epsilon \in$ FIRST (Y_j), then $a \in$ FIRST (X)
     (iii) If $\epsilon \in$ FIRST $(Y_1, \dots Y_k)$, then $\epsilon \in$ FIRST (X)
- Generalization of FIRST relation to string of symbols

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

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

Example of FIRST Set Computation

Grammar

$$
\begin{aligned}
\mathit{Start} &\rightarrow \mathit{Expr} \\
\mathit{Expr} &\rightarrow \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' &\rightarrow +\ \mathit{Term}\ \mathit{Expr}' \mid -\ \mathit{Term}\ \mathit{Expr}' \mid \epsilon \\
\mathit{Term} &\rightarrow \mathit{Factor}\ \mathit{Term}' \\
\mathit{Term}' &\rightarrow \times\ \mathit{Factor}\ \mathit{Term}' \mid \div\ \mathit{Factor}\ \mathit{Term}' \mid \epsilon \\
\mathit{Factor} &\rightarrow (\mathit{Expr}) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

FIRST Sets

$$
\begin{aligned}
&\text{FIRST}\,(\mathit{Start}) = \{\textbf{name}, \textbf{num}, (\,\} \\
&\text{FIRST}\,(\mathit{Expr}) = \{\textbf{name}, \textbf{num}, (\,\} \\
&\text{FIRST}\left(\mathit{Expr}'\right) = \{+, -, \epsilon\} \\
&\text{FIRST}\,(\mathit{Term}) = \{\textbf{name}, \textbf{num}, (\,\} \\
&\text{FIRST}\left(\mathit{Term}'\right) = \{\times, \div, \epsilon\} \\
&\text{FIRST}\,(\mathit{Factor}) = \{\textbf{name}, \textbf{num}, (\,\}
\end{aligned}
$$

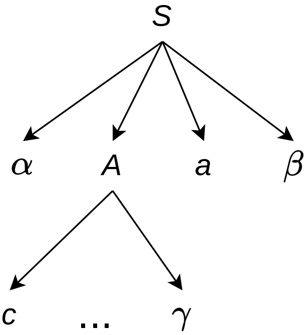
How does a parser decide when to apply the ϵ-production?

FOLLOW Set

Definition

FOLLOW (X) is the set of terminals that can immediately follow X

- That is, $t \in$ FOLLOW (X) if there is any derivation containing Xt



Terminal c is in FIRST (A) and a is in FOLLOW (A)

Steps to Compute FOLLOW Set

(i) Place \$ in FOLLOW (S) where S is the start symbol and the \$ is the end marker

(ii) If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST (β) except ϵ is in FOLLOW (B)

(iii) If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST (β) contains ϵ, then everything in FOLLOW (A) is in FOLLOW (B)

Example of FOLLOW Set Computation

Grammar

$$
\begin{aligned}
\textit{Start} &\rightarrow \textit{Expr} \\
\textit{Expr} &\rightarrow \textit{Term}\ \textit{Expr}' \\
\textit{Expr}' &\rightarrow +\ \textit{Term}\ \textit{Expr}' \mid -\ \textit{Term}\ \textit{Expr}' \mid \epsilon \\
\textit{Term} &\rightarrow \textit{Factor}\ \textit{Term}' \\
\textit{Term}' &\rightarrow \times\ \textit{Factor}\ \textit{Term}' \mid \div\ \textit{Factor}\ \textit{Term}' \mid \epsilon \\
\textit{Factor} &\rightarrow (\textit{Expr}) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

FOLLOW Sets

$$
\begin{aligned}
&\text{FOLLOW}(\textit{Start}) = \{\$\} \\
&\text{FOLLOW}(\textit{Expr}) = \{\$,)\} \\
&\text{FOLLOW}\left(\textit{Expr}'\right) = \{\$,)\} \\
&\text{FOLLOW}(\textit{Term}) = \{\$, +, -,)\} \\
&\text{FOLLOW}\left(\textit{Term}'\right) = \{\$, +, -,)\} \\
&\text{FOLLOW}(\textit{Factor}) = \{\$, +, -, \times, \div,)\}
\end{aligned}
$$

Conditions for Backtrack-Free Grammar

- Consider a production $A \rightarrow \beta$

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

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

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

Expression grammar on the previous slide is backtrack-free

Not All Grammars are Backtrack-Free

$$
\begin{aligned}
Start &\rightarrow Expr \\
Expr &\rightarrow Term\ Expr' \\
Expr' &\rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon \\
Term &\rightarrow Factor\ Term' \\
Term' &\rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon \\
Factor &\rightarrow (Expr) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

$$
\begin{aligned}
Factor &\rightarrow \textbf{name} \mid \textbf{name}[Arglist] \mid \textbf{name}\ (Arglist) \\
Arglist &\rightarrow Expr\ MoreArgs \\
MoreArgs &\rightarrow ,\ Expr\ MoreArgs \mid \epsilon
\end{aligned}
$$

Not All Grammars are Backtrack-Free

$$
\begin{aligned}
Start &\rightarrow Expr \\
Expr &\rightarrow Term\ Expr' \\
Expr' &\rightarrow +\ Term\ Expr' \mid -\ Term\ Expr' \mid \epsilon \\
Term &\rightarrow Factor\ Term' \\
Term' &\rightarrow \times\ Factor\ Term' \mid \div\ Factor\ Term' \mid \epsilon \\
Factor &\rightarrow (Expr) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

$$
\begin{aligned}
Factor &\rightarrow \textbf{name} \mid \textbf{name}[Arglist] \mid \textbf{name}\ (Arglist) \\
Arglist &\rightarrow Expr\ MoreArgs \\
MoreArgs &\rightarrow ,\ Expr\ MoreArgs \mid \epsilon
\end{aligned}
$$

Given a finite lookahead, we can always devise a non-backtrack-free grammar such that the lookahead is insufficient

Left Factoring

Definition

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

- **Algorithm**:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_j$$

$$\Downarrow$$

$$\begin{aligned} A &\rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \ldots \mid \gamma_j \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

Summarizing 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
  - Top-down approach is modeled by calls to functions, where there is one function for each nonterminal

```
void A() {
  Choose an A-production A → X_1 X_2 ... X_k
  for i ← 1 ... k
    if X_i is a nonterminal
      call function X_i
    else if X_i equals the current input symbol a
      advance the input to the next symbol
    else
      // error
}
```

Recursive-Descent Parsing with Backtracking

- Consider a grammar with two productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$
- Suppose FIRST $(\gamma_1) \cap$ FIRST $(\gamma_2) \neq \phi$
  - Let us denote one of the common terminal symbols by a
- The function for X will not know which production to use on the input token a
- To support backtracking
  - All productions should be tried in some order
  - Failure for some production implies the parser needs to try the remaining productions
  - Report an error only when there are no other rules

Predictive Parsing

Definition

Predictive parsing is a 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

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

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: { print("syntax error"); }
  }
}
```

Non-Recursive Predictive Parsing

LL(k) Grammars

Definition

A CFG $G = (T, NT, S, P)$ is LL(1) if and only if for every nonterminal $A \in NT$ where $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ such that $\beta_i \in \Sigma^*$, we have

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

- First L stands for left-to-right scan, second L stands for leftmost derivation, and k represents the number of lookahead tokens
- LL(k) grammars are the class of CFGs for which no backtracking is required
  - Predictive parsers accept LL(k) grammars
- Every LL(1) grammar is a LL(2) grammar
- Many programming language constructs are LL(1)

Definition of LL(k) Grammar

- For a given word $w \in T^*$ and non-negative integer k,
  - w/k is w if $|w| \leq k$, or
  - w/k is a string consisting of the first k symbols of w if $|w| > k$.

- A CFG $G = (T, NT, S, P)$ is LL(k) for some positive integer k if and only if given
  (i) a word $w \in T^*$ such that $|w| \leq k$,
  (ii) a nonterminal $A \in NT$, and
  (iii) a word $w_1 \in T^*$,

  there is at most one production $p \in P$ such that for some $w_2, w_3 \in T^*$,
  1. $S \Rightarrow w_1 A w_3$,
  2. $A \overset{+}{\Rightarrow} w_2$ by first applying production p,
  3. $w_2 w_3 / k = w$.

D. Rosenkrantz and R. Stearns. Properties of Deterministic Top-Down Grammars.

Definition of LL(k) Grammar

- For a given word $w \in T^*$ and non-negative integer k,
  - w/k is w if $|w| \leq k$, or
  - w/k is a string consisting of the first k symbols of w if $|w| > k$.

- A CFG $G = (T, NT, S, P)$ is LL(k) for some positive integer k if and only if given
  (i) a word $w \in T^*$ such that $|w| \leq k$,

> Stated informally in terms of parsing, an LL(k) grammar is a CFG such that for any word in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the k^{th} symbol beyond the beginning of the production.

(
th

2. $A \overset{+}{\Rightarrow} w_2$ by first applying production p,
3. $w_2w_3/k = w$.

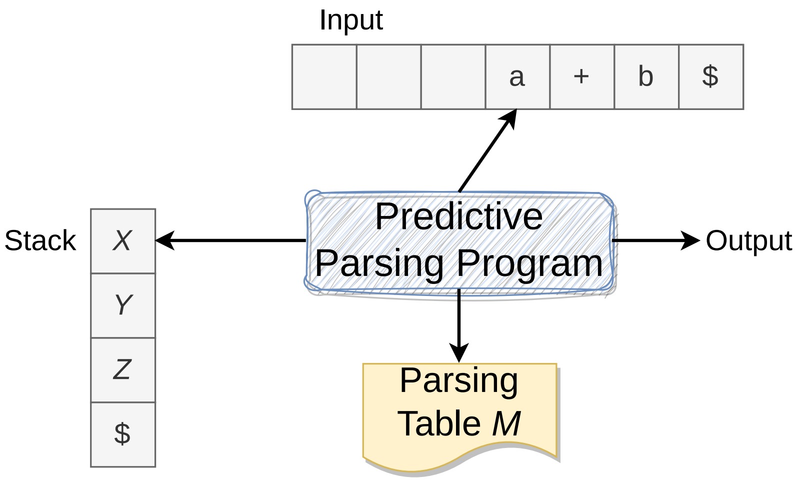
Example LL(2) Parser

$$S \rightarrow AXQ \mid AYR$$

lookahead is the set of 2-sequence tokens that indicate which alternative will succeed

```
void S() {
  if (lookahead(1) == A && lookahead(2) == X) {
    match(A); match(X); match(Q);
  } else if (lookahead(1) == A && lookahead(2) == Y) {
    match(A); match(Y); match(R);
  } else {
    // Raise error
  }
}
```

Nonrecursive Table-Driven LL(1) Parser



LL(1) Parsing Algorithm

- **Input**: String w and parsing table M for grammar G
- **Output**: A leftmost derivation of w if $w \in L(G)$; otherwise, report an error
- **Algorithm**:

```
Let a be the first symbol in w
Let X be the symbol at the top of the stack
while X != $
  if X == a
    pop the stack and advance the input
  else if X is a terminal or M[X,a] is an error entry
    report error
  else if M[X,a] == X → Y1 Y2 ... Yk
    // Expand with the production X → Y1 Y2 ... Yk
    pop the stack
    // Simulate depth-first traversal
    push Yk Yk-1 ... Y1 onto the stack
  X ← top stack symbol
```

Construction of a LL(1) Parsing Table

- **Input**: Grammar G
- **Algorithm**:

```
for each production A → α in G
  for each terminal a in FIRST(α)
    add A → α to M[A,a]
  if ϵ ∈ FIRST(α)
    for each terminal b in FOLLOW(A)
      add A → α to M[A,b]
  if ϵ ∈ FIRST(α) and $ ∈ FOLLOW(A)
      add A → α to M[A,$]

// No production in M[A,a] indicates error
```

LL(1) Parsing Table

Grammar

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

FIRST Sets

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

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

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

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

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

FOLLOW Sets

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

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

FOLLOW $(T) = \{\$, +,)\}$

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

FOLLOW $(F) = \{\$, +, *,)\}$

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	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Working of a LL(1) Parser

Stack	Input	Remark
$\$E$	$\uparrow \textbf{id} + \textbf{id} * \textbf{id}\$$	Expand $E \rightarrow TE'$
$\$E'T$	$\uparrow \textbf{id} + \textbf{id} * \textbf{id}\$$	Expand $T \rightarrow FT'$
$\$E'T'F$	$\uparrow \textbf{id} + \textbf{id} * \textbf{id}\$$	Expand $F \rightarrow \textbf{id}$
$\$E'T'\textbf{id}$	$\uparrow \textbf{id} + \textbf{id} * \textbf{id}\$$	Match **id**
$\$E'T'$	$\uparrow + \textbf{id} * \textbf{id}\$$	Expand $T \rightarrow \epsilon$
$\$E'$	$\uparrow + \textbf{id} * \textbf{id}\$$	Expand $E' \rightarrow +TE'$
$\$E'T+$	$\uparrow + \textbf{id} * \textbf{id}\$$	Match +
$\$E'T$	$\uparrow \textbf{id} * \textbf{id}\$$	Expand $T \rightarrow FT'$
$\$E'T'F$	$\uparrow \textbf{id} * \textbf{id}\$$	Expand $F \rightarrow \textbf{id}$
$\$E'T'\textbf{id}$	$\uparrow \textbf{id} * \textbf{id}\$$	Match **id**
$\$E'T'$	$\uparrow * \textbf{id}\$$	Expand $T' \rightarrow *FT'$
$\$E'T'F*$	$\uparrow * \textbf{id}\$$	Match *
$\$E'T'F$	$\uparrow \textbf{id}\$$	Expand $F \rightarrow \textbf{id}$
$\$E'T'\textbf{id}$	$\uparrow \textbf{id}\$$	Match **id**
$\$E'T'$	$\uparrow \$$	Expand $T' \rightarrow \epsilon$
$\$E'$	$\uparrow \$$	Expand $E' \rightarrow \epsilon$
$\$$	$\uparrow \$$	

More on LL(1) Parsing

- Grammars whose predictive parsing tables contain no duplicate entries are 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 below grammar is ambiguous

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

Limitations with LL(k) Parsing

LL(k) cannot see past **arbitrarily** long constructs from the left edge

$$S \rightarrow A+XQ \mid A+YR$$

Could left factor, but not always possible and natural

$$S \rightarrow A+(XQ \mid YR)$$

Programming language grammars may not be LL(k) (e.g., C function declaration vs definition)

$$\begin{aligned} \text{func} &\rightarrow \text{type ID '(' arg* ')' ';'} \\ &\rightarrow \text{type ID '(' arg* ')' '\{' body '\}'} \end{aligned}$$

Using Ambiguous Grammars

LL(1) Parsing Table for an Ambiguous Grammar

Grammar

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

FIRST Sets

$$\text{FIRST}(S) = \{i, a\}$$
$$\text{FIRST}\left(S'\right) = \{e, \epsilon\}$$
$$\text{FIRST}(E) = \{b\}$$

FOLLOW Sets

$$\text{FOLLOW}(S) = \{\$, e\}$$
$$\text{FOLLOW}\left(S'\right) = \{\$, e\}$$
$$\text{FOLLOW}(E) = \{t\}$$

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

Detecting Errors in Table-Driven Predictive Parsing

Error conditions

(i) Terminal on top of the stack does not match the next input symbol

(ii) Nonterminal A is on top of the stack, a is the next input symbol, and $M[A, a]$ is empty

Choices

(i) Raise an error and quit parsing

(ii) Print an error message, try to recover from the error, and continue with the compilation

Error Recovery in Table-Driven Predictive Parsing

Assume A is the nonterminal at the top of the stack

Panic mode recovery skips over symbols until a token in a set of synchronizing (synch) tokens is found

(i) Add all tokens in FOLLOW (A) to the synch set for A
   - Parsing can continue if the parser skips all input tokens until it sees an input symbol in FOLLOW (A)

(ii) Add symbols in FIRST (A) to the synch set for A
   - Parsing can continue with A if the parser skips all input tokens until it sees an input symbol in FIRST (A)

(iii) Add keywords that begin constructs

(iv) Skip input if the table does not have an entry

(v) ...

Using FOLLOW Sets as Synchronizing Tokens

Grammar

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

FOLLOW Sets

$$\text{FOLLOW}(E) = \text{FOLLOW}\left(E^{'}\right) = \{\$,)\}$$
$$\text{FOLLOW}(T) = \text{FOLLOW}\left(T^{'}\right) = \{\$, +,)\}$$
$$\text{FOLLOW}(F) = \{\$, +, \times,)\}$$

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	$F \rightarrow \textbf{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Error Recovery Moves by Table-Driven Predictive Parser

Stack	Input	Remark
$\$E$	$+\textbf{id}*+\textbf{id}\$$	Error, skip +
$\$E$	$\textbf{id}*+\textbf{id}\$$	Expand $E \rightarrow TE'$
$\$E'T$	$\textbf{id}*+\textbf{id}\$$	Expand $T \rightarrow FT'$
$\$E'T'F$	$\textbf{id}*+\textbf{id}\$$	Expand $F \rightarrow \textbf{id}$
$\$E'T'\textbf{id}$	$\textbf{id}*+\textbf{id}\$$	Match **id**
$\$E'T'$	$*+\textbf{id}\$$	Expand $T \rightarrow *FT'$
$\$E'T'F*$	$*+\textbf{id}\$$	Match $*$
$\$E'T'F$	$+\textbf{id}\$$	Error, $M[F,+]$ = synch, pop F
$\$E'T'$	$+\textbf{id}\$$	Expand $T \rightarrow \epsilon$
$\$E'$	$+\textbf{id}\$$	Expand $E' \rightarrow +TE'$
$\$E'T+$	$+\textbf{id}\$$	Match +
$\$E'T$	$\textbf{id}\$$	Expand $T \rightarrow FT'$
$\$E'T'F$	$\textbf{id}\$$	Expand $F \rightarrow \textbf{id}$
$\$E'T'\textbf{id}$	$\textbf{id}\$$	Match **id**
$\$E'T'$	$\$$	Expand $T' \rightarrow \epsilon$
$\$E'$	$\$$	Expand $E' \rightarrow \epsilon$
$\$$	$\$$	

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 2.4, 4.2–4.4, 2^{nd} edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Section 3.3, 2^{nd} edition, Morgan Kaufmann.