Programming Language Syntax: Top-down Parsing

Read: Scott, Chapter 2.3.2 and 2.3.3
Lecture Outline

- Top-down parsing (also called LL parsing)
  - LL(1) parsing table
  - FIRST, FOLLOW, and PREDICT sets
  - LL(1) grammars

- Bottom-up parsing (also called LR parsing)
  - A brief overview, no detail
Recursive Descent

\[
\text{\underline{start} } \rightarrow \text{ expr } $$
\]
\[
\text{expr } \rightarrow \text{ term } \text{ term\_tail} \quad \text{term\_tail } \rightarrow + \text{ term } \text{ term\_tail} \mid \varepsilon
\]
\[
\text{term } \rightarrow \text{ id } \text{ factor\_tail} \quad \text{factor\_tail } \rightarrow \ast \text{ id } \text{ factor\_tail} \mid \varepsilon
\]

\text{start()}
  \begin{align*}
  \text{case lookahead() of} \\
  \text{id: expr(); match($$) ($$ - end-of-input marker)} \\
  \text{otherwise PARSE\_ERROR}
  \end{align*}

\text{expr()}
  \begin{align*}
  \text{case lookahead() of} \\
  \text{id: term(); term\_tail()} \\
  \text{otherwise PARSE\_ERROR}
  \end{align*}

\text{term\_tail()}
  \begin{align*}
  \text{case lookahead() of} \\
  \text{+ : match('+' ); term(); term\_tail()} \\
  \text{$$ : skip} \quad \text{Predicting epsilon production } \text{term\_tail } \rightarrow \varepsilon \\
  \text{otherwise: PARSE\_ERROR}
  \end{align*}
Recursive Descent

\[
\begin{align*}
\text{start} & \rightarrow \text{expr} \, $$ \\
\text{expr} & \rightarrow \text{term} \, \text{term\_tail} \\
\text{term} & \rightarrow \text{id} \, \text{factor\_tail} \\
\text{term\_tail} & \rightarrow + \, \text{term} \, \text{term\_tail} | \varepsilon \\
\text{factor\_tail} & \rightarrow * \, \text{id} \, \text{factor\_tail} | \varepsilon
\end{align*}
\]

\text{term()}
\begin{align*}
\text{case lookahead() of} \\
\text{id: match(‘id’); factor\_tail()} \\
\text{otherwise: PARSE\_ERROR}
\end{align*}

\text{factor\_tail()}
\begin{align*}
\text{case lookahead() of} \\
* & : \text{match(‘*’); match(‘id’); factor\_tail();} \\
+, $$ & : \text{skip} \\
\text{otherwise PARSE\_ERROR}
\end{align*}
Recursive Descent Parsing

- Parse \( \text{id} + \text{id} \times \text{id} \)
LL(1) Parsing Table

- But how does the parser “predict”?
  - E.g., how does the parser know to expand a $factor\_tail$ by $factor\_tail \rightarrow \varepsilon$ on $+$ and $$?
- It uses the LL(1) parsing table
  - One dimension is nonterminal to expand
  - Other dimension is lookahead token
    - We are interested in one token of lookahead
  - Entry “nonterminal on token” contains the production to apply or contains nothing
**LL(1) Parsing Table**

- **One dimension:** nonterminal to expand
- **Other dimension:** lookahead token

<table>
<thead>
<tr>
<th></th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>α</td>
</tr>
</tbody>
</table>

- E.g., entry “nonterminal A on terminal a” contains production $A \rightarrow α$
- Meaning: when parser is at nonterminal $A$ and lookahead token is $a$, then parser expands $A$ by production $A \rightarrow α$
### LL(1) Parsing Table

**Goal**: Algorithm that constructs the LL(1) table for a given grammar.

**Grammar Rules**:

- `start → expr $$`
- `expr → term term_tail`
- `term → id factor_tail`
- `term_tail → + term term_tail | ε`
- `factor_tail → * id factor_tail | ε`

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>expr</strong></td>
<td>expr $$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>term</strong></td>
<td>term term_tail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>term_tail</strong></td>
<td></td>
<td>+ term term_tail</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>factor_tail</strong></td>
<td></td>
<td></td>
<td>* id factor_tail</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**:
- The LL(1) parsing table is used to determine the action to be taken at each symbol in the input.
- The symbols in the first column are the terminals and non-terminals of the grammar.
- The symbols in the second column are the tokens in the input.
- The symbols in the third, fourth, and fifth columns are the actions to be taken:
  - `—` means no action.
  - `$∈$` means shift.
  - `$•$` means reduce.

---

Programming Languages CSCI 4430, A. Milanova
Intuition

- Top-down parsing
  - Parse tree is built from the top to the leaves
  - Always expand the leftmost nonterminal

What production applies for `factor_tail` on `+`?

+ does not belong to an expansion of `factor_tail`.
However, `factor_tail` has an epsilon production and `+` belongs to an expansion of `term_tail` which follows `factor_tail`. Thus, predict the epsilon production.
**Intuition**

- **Top-down parsing**
  - Parse tree is built from the top to the leaves
  - Always expand the leftmost nonterminal

```
expr -> term term_tail
term_tail -> + term term_tail | ε
term -> id factor_tail
factor_tail -> * id factor_tail | ε
```

What production applies for `term_tail` on `+`?
+ is the **first** symbol in expansions of `+ term term_tail`.

Thus, predict production `term_tail -> + term term_tail`
LL(1) Tables and LL(1) Grammars

- We can construct an LL(1) parsing table for any context-free grammar.
  - In general, the table will contain multiply-defined entries. That is, for some nonterminal and lookahead token, more than one production applies.

- A grammar whose LL(1) parsing table has no multiply-defined entries is said to be LL(1) grammar.
  - LL(1) grammars are a very special subclass of context-free grammars. Why?
FIRST and FOLLOW sets

Let $\alpha$ be any sequence of nonterminals and terminals

- $\text{FIRST}(\alpha)$ is the set of terminals $a$ that begin the strings derived from $\alpha$. E.g., $\text{expr} \ \text{\symbol{$\$\$}} \Rightarrow^* \text{id} \ldots$, thus $\text{id}$ in $\text{FIRST}(\text{expr} \ \text{\symbol{$\$\$}})$

- If there is a derivation $\alpha \Rightarrow^* \epsilon$, then $\epsilon$ is in $\text{FIRST}(\alpha)$

Let $A$ be a nonterminal

- $\text{FOLLOW}(A)$ is the set of terminals $b$ (including special end-of-input marker $\text{\symbol{$\$\$}}$) that can appear immediately to the right of $A$ in some sentential form:

$$ \text{start} \Rightarrow^* \ldots A b \ldots \Rightarrow^* \ldots $$
Computing FIRST

- Apply these rules until no more terminals or $\varepsilon$ can be added to any FIRST($\alpha$) set

  1. If $\alpha$ starts with a terminal $a$, then \( \text{FIRST}(\alpha) = \{ a \} \)
  2. If $\alpha$ is a nonterminal $X$, where $X \rightarrow \varepsilon$, then add $\varepsilon$ to FIRST($\alpha$)
  3. If $\alpha$ is a nonterminal $X \rightarrow Y_1 Y_2 \ldots Y_k$ then add $a$ to FIRST($X$) if for some $i$, $a$ is in FIRST($Y_i$) and $\varepsilon$ is in all of FIRST($Y_1$), \ldots FIRST($Y_{i-1}$). If $\varepsilon$ is in all of FIRST($Y_1$), \ldots FIRST($Y_k$), add $\varepsilon$ to FIRST($X$).

- Everything in FIRST($Y_1$) - $\{ \varepsilon \}$ is surely in FIRST($X$)
- If $Y_1$ does not derive $\varepsilon$, then we add nothing more;
  Otherwise, we add FIRST($Y_2$) - $\{ \varepsilon \}$, and so on

Similarly, if $\alpha$ is $Y_1 Y_2 \ldots Y_k$, we’ll repeat the above


**Warm-up Exercise**

\[
\begin{align*}
\text{start} & \rightarrow \text{expr} \quad \text{expr} \rightarrow \text{term} \ \text{term}_\text{tail} \\
\text{term} & \rightarrow \text{id} \ \text{factor}_\text{tail} \\
\text{term}_\text{tail} & \rightarrow + \ \text{term} \ \text{term}_\text{tail} \mid \epsilon \\
\text{factor}_\text{tail} & \rightarrow * \ \text{id} \ \text{factor}_\text{tail} \mid \epsilon
\end{align*}
\]

FIRST(\(\text{term}\)) = \{ \text{id} \}

FIRST(\(\text{expr}\)) = \{ \text{id}, \epsilon \}

FIRST(\(\text{start}\)) = \{ \text{id} \}

FIRST(\(\text{term}_\text{tail}\)) = \{ +, \epsilon \}

FIRST(\(+\ \text{term} \ \text{term}_\text{tail}\)) = \{ \}

FIRST(\(\text{factor}_\text{tail}\)) = \{ \}

Programming Languages CSCI 4430, A. Milanova
Exercise

Compute FIRST sets:

\[
\begin{align*}
\text{FIRST}(xS) &= \{x, z, y\} \\
\text{FIRST}(AS) &= \{x, z, y\} \\
\text{FIRST}(BCD) &= \{z, v, w, y\} \\
\text{FIRST}(zS) &= \{z, y\} \\
\text{FIRST}(vS) &= \{v\} \\
\text{FIRST}(wS) &= \{w\}
\end{align*}
\]
Computing FOLLOW

- Apply these rules until nothing can be added to any FOLLOW(A) set

  (1) If there is a production \( A \rightarrow \alpha B \beta \), then everything in FIRST(\( \beta \)) except for \( \varepsilon \) should be added to FOLLOW(\( B \))

  \[
  \text{start} \Rightarrow \ast \ldots A \ldots \Rightarrow \ldots \alpha B \beta \ldots \Rightarrow \ldots \alpha B \beta \ldots
  \]

  (2) If there is a production \( A \rightarrow \alpha B \), or a production \( A \rightarrow \alpha B \beta \), where FIRST(\( \beta \)) contains \( \varepsilon \), then everything in FOLLOW(A) should be added to FOLLOW(\( B \))

  \[
  \text{start} \Rightarrow \ast \ldots A \beta \ldots \Rightarrow \ldots \alpha B \beta \ldots
  \]
Warm-up

\[
\begin{align*}
\text{start} & \rightarrow \text{expr} \; \$$ \\
\text{expr} & \rightarrow \text{term} \; \text{term\_tail} \\
\text{term} & \rightarrow \text{id} \; \text{factor\_tail} \\
\text{term\_tail} & \rightarrow + \; \text{term} \; \text{term\_tail} \mid \varepsilon \\
\text{factor\_tail} & \rightarrow \ast \; \text{id} \; \text{factor\_tail} \mid \varepsilon
\end{align*}
\]

\[
\begin{align*}
\text{FOLLOW}(\text{expr}) & = \{ \; \$$ \; \} \\
\text{FOLLOW}(\text{term\_tail}) & = \{ \; \$$ \; \} \\
\text{FOLLOW}(\text{term}) & = \{ \; \$$, +\; \} \\
\text{FOLLOW}(\text{factor\_tail}) & = \{ \; \$$, +\; \}
\end{align*}
\]
Exercise

Compute FOLLOW sets:

FOLLOW(A) = $xyz$
FOLLOW(B) = $\emptyset$, v, w, z
FOLLOW(C) = $\emptyset$, w, z
FOLLOW(D) = $\emptyset$, y, z
FOLLOW(S) = $\emptyset$, $\$$, v, w, y, z

Programming Languages CSCI 4430, A. Milanova
PREDICT Sets

PREDICT(A → α) =

\[
\begin{cases}
\text{FIRST}(\alpha) & \text{if } \alpha \text{ does not derive } \varepsilon \\
(FIRST(\alpha) - \{\varepsilon\}) \cup \text{FOLLOW}(A) & \text{if } \alpha \text{ derives } \varepsilon
\end{cases}
\]
Constructing LL(1) Parsing Table

- Algorithm uses PREDICT sets:
  
  foreach production $A \rightarrow \alpha$ in grammar $G$
  
  foreach terminal $a$ in PREDICT($A \rightarrow \alpha$)
  
  add $A \rightarrow \alpha$ into entry $\text{parse\_table}[A,a]$

- If each entry in $\text{parse\_table}$ contains at most one production, then $G$ is said to be LL(1)
Exercise

Compute PREDICT sets:

PREDICT(S \rightarrow x \ S) = \{x\}

PREDICT(S \rightarrow A \ y) = \{y\}

PREDICT(A \rightarrow BCD) = \{x, y, v, w\}

PREDICT(A \rightarrow \epsilon) = \{y\}

... etc...

\begin{align*}
\text{start} & \rightarrow S \ \$$ \\
S & \rightarrow x \ S \ | \ A \ y \\
A & \rightarrow BCD \ | \ \epsilon \\
B & \rightarrow z \ S \ | \ \epsilon \\
C & \rightarrow v \ S \ | \ \epsilon \\
D & \rightarrow w \ S
\end{align*}
Writing an LL(1) Grammar

- Most context-free grammars are not LL(1) grammars
- Obstacles to LL(1)-ness
  - **Left recursion** is an obstacle. Why?
  - **Common prefixes** are an obstacle. Why?

```latex
expr \rightarrow \text{expr } + \text{term} \mid \text{term}

term \rightarrow \text{term } * \text{id} \mid \text{id}
```

```latex
stmt \rightarrow \text{if } b \text{ then } stmt \text{ else } stmt \mid \text{if } b \text{ then } stmt \mid a
```
Removal of Left Recursion

- Left recursion can be removed from a grammar mechanically
- Started from this left-recursive expression grammar:

\[
\begin{align*}
expr &\to expr + term | term \\
term &\to term * id | id
\end{align*}
\]

- After removal of left recursion, we obtain this equivalent grammar, which is LL(1):

\[
\begin{align*}
expr &\to term term_tail \\
term_tail &\to + term term_tail | \varepsilon \\
term &\to id factor_tail \\
factor_tail &\to * id factor_tail | \varepsilon
\end{align*}
\]
Removal of Common Prefixes

- Common prefixes can be removed mechanically as well by using **left-factoring**
- Original if-then-else grammar:

```
stmt → if b then stmt else stmt | if b then stmt | a
```

- After left-factoring:

```
stmt → if b then stmt else_part | a
else_part → else stmt | ε
```
Exercise

- Compute FIRSTs:
  \[
  \text{FIRST}(stmt \ $$), \ \text{FIRST}(if \ b \ \text{then} \ stmt \ \text{else}_\text{part}), \\
  \text{FIRST}(a), \ \text{FIRST}(\text{else} \ stmt)
  \]

- Compute FOLLOW:
  \[
  \text{FOLLOW}(\text{else}_\text{part})
  \]

- Compute PREDICT sets for all 5 productions
- Construct the LL(1) parsing table. Is this grammar an LL(1) grammar?
Exercise

- Compute FIRSTs:

\[
\text{FIRST}(stmt $$) = \{ \text{if}, \text{a} \} \\
\text{FIRST}(\text{if } b \text{ then } stmt \text{ else}_\text{part}) = \{ \text{if}, \text{b}, \text{a} \} \\
\text{FIRST}(a) = \{ \text{a} \} \\
\text{FIRST}(\text{else } stmt) = \{ \text{else} \}
\]
Exercise

- Compute FOLLOW:

\[
\text{FOLLOW}(\text{else_part}) = \{ \$\$ \}, \text{else} \]

\[
\text{FOLLOW}(\text{stmt}) = \text{FOLLOW}(\text{else-part})
\]

\[
\text{else}
\]

\[
\text{else-part} \rightarrow \text{else stmt} \mid \varepsilon
\]

\[
\text{else-part} \rightarrow \varepsilon
\]

\[
\text{else-part} \rightarrow \text{else stmt} \mid \varepsilon
\]
Exercise

- Construct the LL(1) parsing table

- Is this grammar an LL(1) grammar?

No!
Exercise

if b then if b then a else a
Lecture Outline

- Top-down parsing (also called LL parsing)
  - LL(1) parsing table
  - FIRST, FOLLOW, and PREDICT sets
  - LL(1) grammars

- Bottom-up parsing (also called LR parsing)
  - A brief overview, no detail
Bottom-up Parsing

- Terminals are seen in the order of appearance in the token stream
  \[ \text{id}, \text{id}, \text{id} ; \]
  \[
  \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow
  \]

- Parse tree is constructed
  - From the leaves to the top
  - A rightmost derivation in reverse

\[
\text{list} \rightarrow \text{id} \text{ list}_\text{tail} \\
\text{list}_\text{tail} \rightarrow \text{, id list}_\text{tail} \mid ;
\]
## Bottom-up Parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id, id, id;</td>
<td>shift</td>
</tr>
<tr>
<td>id</td>
<td>, id, id;</td>
<td>shift</td>
</tr>
<tr>
<td>id,</td>
<td>id, id;</td>
<td>shift</td>
</tr>
<tr>
<td>id, id</td>
<td>, id;</td>
<td>shift</td>
</tr>
<tr>
<td>id, id,</td>
<td>id;</td>
<td>shift</td>
</tr>
<tr>
<td>id, id, id</td>
<td>;</td>
<td>shift</td>
</tr>
<tr>
<td>id, id, id;</td>
<td></td>
<td>reduce by</td>
</tr>
</tbody>
</table>

\[
\text{list} \rightarrow \text{id} \ \text{list\_tail} \\
\text{list\_tail} \rightarrow , \ \text{id} \ \text{list\_tail} \ | ;
\]
# Bottom-up Parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id, id, id list_tail</td>
<td>reduce by list_tail → , id list_tail</td>
<td></td>
</tr>
<tr>
<td>id, id list_tail</td>
<td>reduce by list_tail → , id list_tail</td>
<td></td>
</tr>
<tr>
<td>id list_tail</td>
<td>reduce by list → id list_tail</td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>ACCEPT</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
list → id list_tail
list_tail → , id list_tail | ;
```
Bottom-up Parsing

- Also called LR parsing
- LR parsers work with LR(k) grammars
  - L stands for “left-to-right” scan of input
  - R stands for “rightmost” derivation
  - k stands for “need k tokens of lookahead”
- We are interested in LR(0) and LR(1) and variants in between
- LR parsing is better than LL parsing!
  - Accepts larger class of languages
  - Just as efficient!
LR Parsing

- The parsing method used in practice
  - LR parsers recognize virtually all PL constructs
  - LR parsers recognize a much larger set of grammars than predictive parsers
  - LR parsing is efficient

- LR parsing variants
  - SLR (or Simple LR)
  - LALR (or Lookahead LR) – yacc/bison generate LALR parsers
  - LR (Canonical LR)
  - SLR < LALR < LR
Main Idea

- **Stack ⇐ Input**
- **Stack**: holds the part of the input seen so far
  - A string of both terminals and nonterminals
- **Input**: holds the remaining part of the input
  - A string of terminals
- **Parser performs two actions**
  - **Reduce**: parser pops a “suitable” production right-hand-side off top of stack, and pushes production’s left-hand-side on the stack
  - **Shift**: parser pushes next terminal from the input on top of the stack
Example

- Recall the grammar

\[
\begin{align*}
expr & \rightarrow expr + term \mid term \\
term & \rightarrow term * id \mid id
\end{align*}
\]

- This is not LL(1) because it is left recursive
- LR parsers can handle left recursion!

- Consider string

\[
\text{id + id * id}
\]
### id + id*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id+id*id</td>
<td>id+id*id</td>
<td>shift id</td>
</tr>
<tr>
<td>id</td>
<td>+id*id</td>
<td>reduce by term → id</td>
</tr>
<tr>
<td>term</td>
<td>+id*id</td>
<td>reduce by expr → term</td>
</tr>
<tr>
<td>expr</td>
<td>+id*id</td>
<td>shift +</td>
</tr>
<tr>
<td>expr+</td>
<td>id*id</td>
<td>shift id</td>
</tr>
<tr>
<td>expr+id</td>
<td>*id</td>
<td>reduce by term → id</td>
</tr>
</tbody>
</table>

```
eexpr → expr + term | term
term → term * id | id
```
### expr + term

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr</code> + <code>term</code></td>
<td><code>*id</code></td>
<td>shift <code>*</code></td>
</tr>
<tr>
<td><code>expr</code> + <code>term</code></td>
<td><code>id</code></td>
<td>shift <code>id</code></td>
</tr>
<tr>
<td><code>expr</code> + <code>term</code></td>
<td><code>*id</code></td>
<td>reduce by <code>term</code> → <code>term</code> <code>*id</code></td>
</tr>
<tr>
<td><code>expr</code></td>
<td></td>
<td>reduce by <code>expr</code> → <code>expr</code> + <code>term</code></td>
</tr>
<tr>
<td><code>expr</code></td>
<td></td>
<td>ACCEPT, SUCCESS</td>
</tr>
</tbody>
</table>

### Grammar Rules

- `expr` → `expr` + `term` | `term`
- `term` → `term` `*id` | `id`
id + id*id

Sequence of reductions performed by parser

id+id*id

A rightmost derivation in reverse

term+id*id

• The stack (e.g., expr) concatenated with remaining input (e.g., +id*id) gives a sentential form (expr+id*id) in the rightmost derivation

expr+id*id

expr+term*id

expr+term

expr

expr → expr + term | term

term → term * id | id
The End