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
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 $A \rightarrow \alpha$
- Meaning: when parser is at nonterminal $A$ and lookahead token is $a$, then parser expands $A$ by production $A \rightarrow \alpha$
# LL(1) Parsing Table

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

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>expr $$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>expr</td>
<td>term term</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>id factor</td>
<td>-</td>
<td>+ term term</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>id factor</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>factor</td>
<td>-</td>
<td>$$</td>
<td>* id factor</td>
<td>$$</td>
</tr>
</tbody>
</table>
**Intuition**

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

```latex
\begin{align*}
\text{expr} & \rightarrow \text{term} \text{ term}\_\text{tail} \\
\text{term}\_\text{tail} & \rightarrow + \text{ term} \text{ term}\_\text{tail} \ | \varepsilon \\
\text{term} & \rightarrow \text{id} \text{ factor}\_\text{tail} \\
\text{factor}\_\text{tail} & \rightarrow * \text{id} \text{ factor}\_\text{tail} \ | \varepsilon
\end{align*}
```

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

```
<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr</code>  → <code>term term_tail</code></td>
</tr>
<tr>
<td><code>term_tail</code>  → <code>+ term term_tail</code></td>
</tr>
<tr>
<td><code>term</code> → <code>id factor_tail</code></td>
</tr>
<tr>
<td><code>factor_tail</code> → <code>* id factor_tail</code></td>
</tr>
</tbody>
</table>
```

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., $\text{expr} \Rightarrow^* \text{id}...$, thus $\text{id}$ in $\text{FIRST(expr)}$
  - 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 $$) that can appear immediately to the right of $A$ in some sentential form:
    
    $\text{start} \Rightarrow^* ...A_{\text{something}}b... \Rightarrow^*...$
Computing FIRST

- Apply these rules until no more terminals or ε can be added to any FIRST(α) set
  1. If α starts with a terminal a, then FIRST(α) = { a }
  2. If α is a nonterminal X, where X → ε, then add ε to FIRST(α)
  3. If α is a nonterminal X → Y₁ Y₂ ... Yₖ then add a to FIRST(X) if for some i, a is in FIRST(Yᵢ) and ε is in all of FIRST(Y₁), ... FIRST(Yᵢ₋₁). If ε is in all of FIRST(Y₁), ... FIRST(Yₖ), add ε to FIRST(X).
    - Everything in FIRST(Y₁) is surely in FIRST(X)
    - If Y₁ does not derive ε, then we add nothing more;
      Otherwise, we add FIRST(Y₂), and so on
  Similarly, if α is Y₁ Y₂ ... Yₖ, we’ll repeat the above
Warm-up Exercise

\[
\begin{align*}
\text{FIRST}(\text{term}) &= \{ \text{id} \} \\
\text{FIRST}(\text{expr}) &=  \\
\text{FIRST}(\text{start}) &=  \\
\text{FIRST}(\text{term_tail}) &=  \\
\text{FIRST}(+ \text{ term term_tail}) &=  \\
\text{FIRST}(\text{factor_tail}) &= 
\end{align*}
\]
Exercise

Compute FIRST sets:

FIRST($x$ $S$) = FIRST($S$) = 
FIRST($A$ $y$) = FIRST($A$) = 
FIRST($BCD$) = FIRST($B$) = 
FIRST($z$ $S$) = FIRST($C$) = 
FIRST($v$ $S$) = FIRST($D$) = 
FIRST($w$ $S$) =
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($B$)

(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($B$)

Notation:
$A, B, S$ are nonterminals.
$\alpha, \beta$ are arbitrary sequences of terminals and nonterminals.
Warm-up

```
start → expr $$
expr → term term_tail

term → id factor_tail

term_tail → + term term_tail | ε
factor_tail → * id factor_tail | ε
```

FOLLOW(expr) = { $$ } 
FOLLOW(term_tail) = 
FOLLOW(term) = 
FOLLOW(factor_tail) = 
Exercise

Compute FOLLOW sets:

FOLLOW(A) =
FOLLOW(B) =
FOLLOW(C) =
FOLLOW(D) =
FOLLOW(S) =
PREDICT Sets

\[
PREDICT(A \rightarrow \alpha) = \begin{cases} 
\text{FIRST}(\alpha) & \text{if } \alpha \text{ does not derive } \varepsilon \\
(\text{FIRST}(\alpha) - \{\varepsilon\}) \cup \text{FOLLOW}(A) & \text{if } \alpha \text{ derives } \varepsilon 
\end{cases}
\]
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 parse_table[$A,a$]

- If each entry in parse_table contains at most one production, then $G$ is said to be LL(1)
Exercise

Compute PREDICT sets:
PREDICT(S → x S) =
PREDICT(S → A y) =
PREDICT(A → BCD) =
PREDICT(A → ε) =

… etc…
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?
    - expr → expr + term | term
    - term → term * id | id
  - Common prefixes are an obstacle. Why?
    - stmt → if b then stmt else stmt | if b then stmt | a
Removal of Left Recursion

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

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

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

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

- Common prefixes can be removed mechanically as well, by using left-factoring

- Original if-then-else grammar:

  \[
  stmt \rightarrow \textbf{if } b \textbf{ then } stmt \textbf{ else } stmt \mid \\
  \textbf{if } b \textbf{ then } stmt \mid \\
  a
  \]

- After left-factoring:

  \[
  stmt \rightarrow \textbf{if } b \textbf{ then } stmt \textbf{ else}_\text{part} \mid a \\
  \text{else}_\text{part} \rightarrow \textbf{else } stmt \mid \varepsilon
  \]
Exercise

- Compute FIRSTs:
  FIRST($stmt $$), FIRST(if b then stmt else_part), FIRST(a), FIRST(else stmt)

- Compute FOLLOW:
  FOLLOW(else_part)

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

- Compute FIRSTs:

FIRST($stmt$$) =

FIRST($if\ b\ then\ stmt\ else\ part$) =

FIRST($a$) =

FIRST($else\ stmt$) =
Exercise

- Compute FOLLOW:

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


text:

\[
\begin{align*}
\text{start} & \rightarrow \text{stmt} \\
\text{stmt} & \rightarrow \text{if b then stmt else part} \mid a \\
\text{else part} & \rightarrow \text{else stmt} \mid \varepsilon
\end{align*}
\]
Exercise

- Construct the LL(1) parsing table

- Is this grammar an LL(1) grammar?

\[
\begin{align*}
\text{start} & \rightarrow \text{stmt} \quad $$ \\
\text{stmt} & \rightarrow \text{if} \ b \ \text{then} \ \text{stmt} \ \text{else_part} \ | \ \text{a} \\
\text{else_part} & \rightarrow \text{else} \ \text{stmt} \ | \ \varepsilon
\end{align*}
\]
Exercise
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 right-most derivation in reverse

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

### Stack | Input | Action
--- | --- | ---
id | id,id,id; | shift
id | ,id,id; | shift
id | id,id; | shift
id,id | ,id; | shift
id,id | id; | shift
id,id,id | ; | shift
id,id,id; | | reduce by 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</td>
<td>list_tail</td>
<td>reduce by</td>
</tr>
<tr>
<td>id, id</td>
<td>list_tail</td>
<td>list_tail → , id list_tail</td>
</tr>
<tr>
<td>id</td>
<td>list_tail</td>
<td>reduce by</td>
</tr>
<tr>
<td>id</td>
<td>list_tail</td>
<td>list_tail → , id list_tail</td>
</tr>
<tr>
<td>id</td>
<td>list_tail</td>
<td>reduce by</td>
</tr>
<tr>
<td>id</td>
<td>list_tail</td>
<td>list → id list_tail</td>
</tr>
<tr>
<td>list</td>
<td></td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

**Grammar Rules:**

- `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 $\leftarrow$ 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

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

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

- Consider string

\[
id + id \ast id
\]
\[ \text{id} + \text{id} \ast \text{id} \]

<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 ( \text{term} \rightarrow \text{id} )</td>
</tr>
<tr>
<td>term</td>
<td>+id*id</td>
<td>reduce by ( \text{expr} \rightarrow \text{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 ( \text{term} \rightarrow \text{id} )</td>
</tr>
</tbody>
</table>

\[
\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term} \\
\text{term} \rightarrow \text{term} \ast \text{id} \mid \text{id}
\]
### id + id*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr+term</td>
<td>*id</td>
<td>shift *</td>
</tr>
<tr>
<td>expr+term*</td>
<td>id</td>
<td>shift id</td>
</tr>
<tr>
<td>expr+term*id</td>
<td></td>
<td>reduce by term (\rightarrow) term *id</td>
</tr>
<tr>
<td>expr+term</td>
<td></td>
<td>reduce by expr (\rightarrow) expr+term</td>
</tr>
<tr>
<td>expr</td>
<td></td>
<td>ACCEPT, SUCCESS</td>
</tr>
</tbody>
</table>

```plaintext
expr \(\rightarrow\) expr + term | term
term \(\rightarrow\) term * id | id
```
**id + id\*id**

Sequence of reductions performed by parser

- A rightmost derivation in reverse

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

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