### Table-Driven Parsing

- It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls [1].
- The non-recursive parser looks up the production to be applied in a parsing table.
- The table can be constructed directly from LL(1) grammars.

### Table-Driven Parsing

- **An input buffer**
  - Contains the input string
  - The string can be followed by $, an end marker to indicate the end of the string
- **A stack**
  - Contains symbols with $ on the bottom, with the start symbol initially on the top
- **A parsing table** (2-dimensional array $M[A, a]$)
- **An output stream** (production rules applied for derivation)
Input: a string w, a parsing table M for grammar G

Output: if w is in L(G), a leftmost derivation of w; otherwise, an error indication

Method:

set ip to point to the first symbol of w$
repeat
let X be the top stack symbol and a the symbol pointed to by ip;
if X is a terminal or $, then
   if X = a then
      pop X from the stack and advance ip
   else error()
else /* X is a non-terminal */
   if M[X, a] = X→Y₁Y₂…Yₖ, then
      pop X from the stack
      push Yₖ, …, Y₂, Y₁ on to the stack
      output the production X→Y₁Y₂…Yₖ
   else error()
until X =$

An Example

• Input String: id + id * id
• Input parsing table for the following grammar:

<table>
<thead>
<tr>
<th>E</th>
<th>E'</th>
<th>T</th>
<th>T'</th>
<th>F</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>E'</td>
<td>E</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>T'</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

An Example

• Input String: id + id * id
• Input parsing table for the following grammar:

<table>
<thead>
<tr>
<th>NON-TERMINAL</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>E'</td>
<td>E'</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>$</td>
</tr>
<tr>
<td>T'</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>$</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>$</td>
</tr>
</tbody>
</table>
**LL Parsing**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id + id * id$</td>
<td></td>
</tr>
<tr>
<td>$E'T$</td>
<td>id + id * id$</td>
<td>E → TE'</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>id + id * id$</td>
<td>T → FT'</td>
</tr>
<tr>
<td>$E'Tid$</td>
<td>id + id * id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T$</td>
<td>+ id * id$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$$</td>
<td>$$</td>
<td>E' → ε</td>
</tr>
</tbody>
</table>

---

**Construction of Parsing Table**

- Two functions used to fill in a predicative parsing table for $G$
  - **FIRST**
    - For non-terminal $A$, FIRST($A$) is the set of terminals that begin the strings derived from $A$
  - **FOLLOW**
    - For non-terminal $A$, FOLLOW($A$) is the set of terminals that appear immediately to the right of $A$. If $A$ can be the rightmost symbol, $\$$ can be included in FOLLOW($A$)
Algorithm to compute FIRST(X)

• If X is terminal, then FIRST(X) = \{X\}
• If X \rightarrow \epsilon is a production, then \epsilon \in FIRST(X)
• If X is non-terminal, and X \rightarrow Y_1 Y_2 \ldots Y_k, then place a in FIRST(X), if for some i, a is in FIRST(Y_i), and \epsilon is in all of FIRST(Y_1), \ldots, FIRST(Y_{i-1}). Place \epsilon in FIRST(X) if for all i, FIRST(X_i) contains \epsilon

Revisit the example

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

FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}
FIRST(E')={+, \epsilon}\nFIRST(T')={*, \epsilon}\n
Algorithm to compute FOLLOW(X)

- Place $ in FOLLOW(S)
- If there is a production $A -> \alpha B\beta$, then \{FIRST(\beta) - \varepsilon\} \subseteq FOLLOW(B)
- If there is a production $A -> \alpha B$, or a production $A -> \alpha B\beta$, where FIRST(\beta) contains \varepsilon, then FOLLOW(A) \subseteq FOLLOW(B)

Revisit the example

E -> TE'
E' -> +TE' | \varepsilon
T -> FT'
T' -> *FT' | \varepsilon
F -> (E) | id

FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}
FIRST(E')={+, \varepsilon\}
FIRST(T')={*, \varepsilon\}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW(T')
    = FIRST(E') - \varepsilon U FOLLOW(E')
    = {+, ), $}
FOLLOW(F) = FIRST(T') - \varepsilon U FOLLOW(T')
    = {*, +, ) $}
Algorithm to create a parsing table

Input: Grammar G
Output: Parsing table M
Method:
1. for each production $A \rightarrow \alpha$, do steps 2 and 3
2. for each terminal $a$ in FIRST($\alpha$), add $A \rightarrow \alpha$ to $M[A, a]$
3. if $\epsilon$ is in FIRST($\alpha$), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal $b$ in FOLLOW($A$). If $\$$ is in FOLLOW($A$), add $A \rightarrow \alpha$ to $M[A, \$$]
4. make each undefined entry of $M$ be error

Revisit the example

| FIRST(E) = | FOLLOW(E) = | E -> TE' | $E' \rightarrow + TE' \mid \epsilon$
|-----------|-------------|-------------------|
| FIRST(T) = | FOLLOW(E') = $\{(), \$$\}$ | T -> FT' | $T' \rightarrow * FT' \mid \epsilon$
| FIRST(F) = $\{(, \ id\}$ | FOLLOW(T) = $\{+, \$$\}$ | F -> (E) | $id$
| FIRST(E')={+, $\epsilon$} | FOLLOW(T') = $\{+, \$$\}$ | | |
| FIRST(T')={$*, \$\} | FOLLOW(F) = $\{*, +, \$$\}$ | | |

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Input Symbol</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-up Parsing

- Construct a parse tree for an input string beginning at the leaves, and working up towards the root
  - E.g., reducing a string $w$ to the start symbol

An Example

- Consider the grammar:
  - $S \rightarrow aABe$
  - $A \rightarrow Abc \mid b$
  - $B \rightarrow d$
- Input string: abbcde
- How to build a parse tree bottom-up?
Bottom-up Parsing

- Scan the string to look for a substring that matches the right side of some production
  - E.g., b matches A, while d matches B
- Choose the leftmost b and replace it with A, obtaining “aAbcde”
- Now “Abc”, “b”, and “d” match the right side of some rules
- Choose the leftmost longest substring to replace, obtaining “aAde”

Bottom-up Parsing

- Replace d with B, getting “aABe”
- Replace the whole string with S
LR(1) Parsing

- LR(1) Grammar
- Input String: id + id * id
- There is still a parsing table involved (not shown here)
- A stack is also used to help parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id + id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>id .</td>
<td>id + id * id$</td>
<td>Reduce by F-&gt;id</td>
</tr>
<tr>
<td>F</td>
<td>F + id * id$</td>
<td>Reduce by T-&gt;F</td>
</tr>
<tr>
<td>T</td>
<td>T + id * id$</td>
<td>Reduce by E-&gt;T</td>
</tr>
<tr>
<td>E</td>
<td>E + id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>E +</td>
<td>id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>E + id .</td>
<td>* id$</td>
<td>Reduce by F-&gt;id</td>
</tr>
<tr>
<td>E + F</td>
<td>* id$</td>
<td>Reduce by T-&gt;F</td>
</tr>
<tr>
<td>E + T</td>
<td>* id$</td>
<td>shift</td>
</tr>
<tr>
<td>E + T *</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>E + T * id</td>
<td>$</td>
<td>Reduce by F-&gt;id</td>
</tr>
<tr>
<td>E + T * F</td>
<td>$</td>
<td>Reduce by T-&gt;T*F</td>
</tr>
<tr>
<td>E + T</td>
<td>$</td>
<td>Reduce by E-&gt;E+T</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Homework

• Exercises 1.1, 2.4, 2.12
• Hint:
  – For 2.4, please refer to slides about conversions from RE to minimized DFA
• Due Date: 09/20 11:59pm
• Submit the electronic copy to Canvas.