Implementing A Top-Down Parser

- LL(1) grammar example:
  \[ S \rightarrow ES' \]
  \[ S' \rightarrow \epsilon \mid + S \]
  \[ E \rightarrow \text{num} \mid (S) \]
- Use a predictive parsing table:

```
<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES</td>
<td>+S</td>
<td>ES</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
- Implement a recursive-descent parser using mutually recursive procedures `parse_E()`, `parse_ES()`, `parse_ES_prime()`

How to Construct Parsing Tables

- Need an algorithm that generates a predictive parse table from a grammar

```
<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES</td>
<td>+S</td>
<td>ES</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Constructing Parse Tables

- Parsing table tells us, for each non-terminal and each look-ahead symbol what production to use
- \( \text{FIRST}(\gamma) \) for arbitrary string of terminals and non-terminals \( \gamma \) = set of symbols that might begin the fully expanded version of \( \gamma \)
- \( \text{FOLLOW}(X) \) for a non-terminal \( X \) = set of symbols that might follow the derivation of \( X \)

Parse Table Entries

- Consider a production \( X \rightarrow \gamma \)
- Add \( \rightarrow \gamma \) to the \( X \) row for symbols in FIRST(\( \gamma \))

```
<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES</td>
<td>+S</td>
<td>ES</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
- If \( \gamma \) can derive \( \epsilon \) (\( \gamma \) is nullable), add \( \rightarrow \gamma \) for each symbol in FOLLOW(\( X \))
- Grammar is LL(1) if no conflicting entries
Computing nullable, FIRST

- X is nullable if it can derive the empty string:
  - if: X → ε
  - if: X → Y₁...Yₙ where all Y_i are nullable
  - Algorithm: assume all non-terminals non-nullable, apply rules repeatedly until no change

- Computing FIRST(γ)
  - FIRST(X) ⊇ FIRST(γ) if X → γ
  - FIRST(aβ) = { a }
  - FIRST(Xβ) ⊇ FIRST(X)
  - FIRST(Xβ) ⊇ FIRST(β) if X is nullable
  - Algorithm: Assume FIRST(γ) = {} for all γ, apply rules repeatedly to build FIRST sets.

Computing FOLLOW

- Compute FOLLOW(X):
  - FOLLOW(S) = { $ } if S → $ 
  - if X → αYβ, FOLLOW(Y) ⊇ FIRST(β)
  - if X → αYβ and β is nullable (or non-existent), FOLLOW(Y) ⊇ FOLLOW(X)

- Algorithm: Assume FOLLOW(X) = {} for all X, apply rules repeatedly to build FOLLOW sets

- Common theme: iterative analysis. Start with initial assignment, apply rules until no change

Example

- nullable
  - only S' is nullable

- FIRST
  - FIRST(E S') = { num, ( )
  - FIRST(S) = { * }:
  - FIRST(num) = { num, ( )
  - FIRST( S') = { ( )
  - FIRST(S') = { * }

- FOLLOW
  - FOLLOW(S) = { $, ) }
  - FOLLOW(S') = { ( ) };
  - FOLLOW(E) = { ( * ), $ }

LL Grammars and Associativity

- We have been using grammar for language of "sums with parentheses" e.g., (1-(3+4))+5

- Started with simple, right-associative grammar:
  S → E + S | E
  E → num | ( S )

- Transformed it to an LL(1) grammar by left-factoring:
  S → ES'
  S' → E | E + S
  E → num | ( S )

- What if we start with a left-associative grammar?
  S → S + E | E
  E → num | ( S )

Left vs. Right Associativity

Right recursion: right-associative
S → E + S  
S → E  
E → num  

Left recursion: left-associative
S → S + E  
S → E  
E → num

Left Recursion

- Left-recursive grammars don’t work with top-down parsing: we don’t know where to stop the recursion

derived string  lookahead  read not read
S 1 1 + 2 + 3 + 4
S → E 1 1 + 2 + 3 + 4
S + E 1 1 + 2 + 3 + 4
S → E + E 1 1 + 2 + 3 + 4
E + E → E 1 1 + 2 + 3 + 4
E + E + E 2 1 + 2 + 3 + 4
1 + E + E 3 1 + 2 + 3 + 4
1 + 2 + E + E 4 1 + 2 + 3 + 4
1 + 2 + 3 + 4 8 1 + 2 + 3 + 4
Left-Recursive Grammars

- Left-recursive grammars are not LL(1)!
  \[ S \rightarrow S \alpha \]
  \[ S \rightarrow \beta \]
- \( \text{FIRST}(\beta) \subseteq \text{FIRST}(S\alpha) \)
- If \( \beta \) is nullable, then so is \( S\alpha \)
- Both productions will appear in the table at row \( S \) in all the columns corresponding to symbols in \( \text{FIRST}(\beta) \) if \( \beta \) is not nullable, or to symbols in \( \text{FOLLOW}(S) \) if \( \beta \) is nullable

Eliminate Left Recursion

- Method for left-recursion elimination:
  Replace
  \[ A \rightarrow A \alpha_1 \mid \ldots \mid A \alpha_m \]
  \[ A \rightarrow \beta_1 \mid \ldots \mid \beta_n \]
  with
  \[ A \rightarrow \beta_1 B \mid \ldots \mid \beta_n B \]
  \[ B \rightarrow \alpha_1 B \mid \ldots \mid \alpha_m B \mid \varepsilon \]
- (See the complete algorithm in the Dragon Book)

Creating an LL(1) Grammar

- Start with a left-recursive grammar:
  \[ S \rightarrow S + E \]
  \[ S \rightarrow E \]
  and apply left-recursion elimination algorithm:
  \[ S \rightarrow ES' \]
  \[ S' \rightarrow +E S' \mid \varepsilon \]
- Start with a right-recursive grammar:
  \[ S \rightarrow E + S \]
  \[ S \rightarrow E \]
  and apply left-factoring to eliminate common prefixes:
  \[ S \rightarrow ES' \]
  \[ S' \rightarrow +S \mid \varepsilon \]

Top-Down Parsing Summary

- Language grammar
  \[ \text{Left-recursion elimination} \]
  \[ \text{Left-factoring} \]
  \[ LL(1) \text{grammar} \]
  \[ \text{predictive parsing table} \]
  \[ \text{recursive-descent parser} \]
  \[ \text{parser with AST generation} \]