Predictive Parsing

Roadmap (Where are we?)

- Specifying syntax
  - Context-free grammars
  - Ambiguity

- (Back Tracking) Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal

- Predictive top-down parsing
  - FIRST, FOLLOW, FIRST+
  - The LL(1) condition
  - Table-driven LL(1) parsers
  - Recursive descent parsers
  - Left factoring

Predictive Parsing

Basic idea

given \( A \to \alpha \mid \beta \), the parser should be able to choose between \( \alpha \) \& \( \beta \)

First sets

For some \( \alpha \to \gamma \), define \( \text{First}(\alpha) \) as the set of tokens that appear as the first symbol in some string that derives from \( \alpha \).

That is, \( a \in \text{First}(\alpha) \iff \alpha \Rightarrow^* a \gamma \), for some \( \gamma \)

The LL(1) Property

If \( A \to \alpha \) and \( A \to \beta \) both appear in the grammar, we would like \( \text{First}(\alpha) \cap \text{First}(\beta) = \emptyset \)

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

This is almost correct
Will also need FOLLOW

The FIRST Set

\[ \alpha \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build FIRST(\( \alpha \)) for all grammar symbols \( \alpha \):

1. if \( \alpha \) is a terminal (token), FIRST(\( \alpha \)) := \{ \( \alpha \) \}
2. if \( \alpha \) ::= \( \varepsilon \), then \( \varepsilon \in \text{FIRST}(\alpha) \)
3. iterate until no more terminals or \( \varepsilon \) can be added to any FIRST(\( \alpha \)):
   - if \( \alpha \to \gamma_1 \gamma_2 \ldots \gamma_k \) then
     - \( a \in \text{FIRST}(\alpha) \) if \( a \in \text{FIRST}(X_i) \) and \( \varepsilon \in \text{FIRST}(X_j) \) for all \( 1 \leq j < i \)
     - \( \varepsilon \in \text{FIRST}(\alpha) \) if \( \varepsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq k \)

Note: if \( \varepsilon \in \text{FIRST}(Y_i) \), then FIRST(\( Y_i \)) is irrelevant, for \( 1 < i \)

The FOLLOW Set

For a non-terminal \( A \), define FOLLOW(\( A \)) as

\( \text{FOLLOW}(A) := \text{the set of terminals that can appear immediately to the right of } A \text{ in some sentential form.} \)

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set.
The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Place $ in FOLLOW(<goal>) // $ = EOF

   iterate until no more terminals or ε can be added
   to any FOLLOW(X):

2. If A $\rightarrow$ αβ then
   put (FIRST(β) - ε) in FOLLOW(B)

3. If A $\rightarrow$ αβ then
   put FOLLOW(A) in FOLLOW(B)

4. If A $\rightarrow$ αβ and ε $\in$ FIRST(β) then
   put FOLLOW(A) in FOLLOW(B)

Predictive Parsing

If A $\rightarrow$ α and A $\rightarrow$ β and ε $\in$ FIRST(α), then we need to ensure
that FIRST(β) is disjoint from FOLLOW(A), too

Define FIRST(β) for rule A $\rightarrow$ β as

- FIRST(β), if ε $\in$ FIRST(β)
- FIRST(β) - ε, otherwise

Recall Top-down Parsing

A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm
Construct the root node of the parse tree
Repeat until the fringe of the parse tree matches the input string
1. At a node labeled A, select a production with A on its lhs and, for each
   symbol on its rhs, construct the appropriate child
2. When a terminal symbol is added to the fringe and it doesn’t match the
   fringe, backtrack
3. Find the next node to be expanded (label = NT)
   - For LL(1) grammars
     → we can use FIRST to pick the right production in step 1
     → Using 1 token of lookahead

LL(1) Parser Example

Is the following grammar LL(1)?

S ::= a $b | ε$

FIRST(aSb) = { a }
FIRST(e) = { ε }
FIRST(eSb) = { a }
FIRST(e) = { FIRST(e) - { ε } } ∪ FOLLOW(S) = { $, b }$

LL(1)? YES, since { a } ∩ { $, b } = ∅
LL(1) Parser Example

Building Table-driven Top Down Parsers

Building the complete table
• Need a row for every NT & a column for every T
• Need an algorithm to build the table

Filling in TABLE[x,y], X ∈ NT, y ∈ T
• entry is the rule X ::= y, if y ∈ FIRST(y)
• entry is error otherwise (can treat empty entry as implicit error)
If any entry is defined multiple times, G is not LL(1)

This is the LL(1) table construction algorithm

Recursive Descent Parsers

• Description
  → Top-down parser built from a set of mutually-recursive procedures
  → Each procedure usually implements a nonterminal from the grammar

• Implementation
  → Backtracking
  → Choose different production if current choice fails
  → LL(1) Predictive
    • Compare lookahead token to FIRST sets to select production
  → Utility function
    match(t) {
      if (lookahead == t) lookahead = next_token();
      else error();
    }

LL(1) Skeleton Parser

Table-driven LL(1) Parser Example

Table-driven LL(1) Parser Example

Table-driven LL(1) Parser Example

Table-driven LL(1) Parser Example
Recursive Descent LL(1) Parser Implementation

- Terminals
  → Terminals in the input stream appear as token lookahead
  → Can advance lookahead token using: lookahead ← next_token()

- Nonterminals
  → Every NT is associated with a parsing procedure
  → The parsing procedure for \( A \in NT \), proc \( A \)
    - Responsible for parsing and consuming any string that can be derived from \( A \)
    - Choose to replace \( A \) with \( \beta \) for production \( A \rightarrow \beta \)
    - If lookahead \( \notin FIRST(\beta) \)
      → Error if lookahead token does not match any production

- Parser is invoked by calling proc \( S \) for start symbol \( S \)

Recursive Descent Parser Example 1

- Given grammar \( S \rightarrow xyz | abc \)
  → FIRST(\(xyz\)) = \{x\}, FIRST(\(abc\)) = \{a\}
- Parser
  ```java
  parse_S() {
    if (lookahead == "x") {
      match("x"); match("y"); match("z"); // \( S \rightarrow xyz \)
    } else if (lookahead == "a") {
      match("a"); match("b"); match("c"); // \( S \rightarrow abc \)
    } else error();
  }
  ```

Recursive Descent Parser Example 2

- Given grammar \( S \rightarrow A | B \)
  \( A \rightarrow x | y \)
  \( B \rightarrow z \)
  → FIRST(\(A\)) = \{x, y\}, FIRST(\(B\)) = \{z\}
- Parser
  ```java
  parse_S() {
    if ((lookahead == "x") || (lookahead == "y"))
      parse_A(); // \( S \rightarrow A \)
    else if (lookahead == "z")
      parse_B(); // \( S \rightarrow B \)
    else error();
  }
  ```
  ```java
  parse_A() {
    if (lookahead == "x")
      match("x"); // \( A \rightarrow x \)
    else if (lookahead == "y")
      match("y"); // \( A \rightarrow y \)
    else error();
  }
  ```
  ```java
  parse_B() {
    if (lookahead == "z")
      match("z"); // \( B \rightarrow z \)
    else error();
  }
  ```

Recursive Descent Parser Example 3

- Given grammar \( S \rightarrow a S b | \epsilon \)
  → FIRST(\(aSb\)) = \{a\}, FIRST(\(\epsilon\)) = \{b, \$\}
- Parser
  ```java
  parse_S() {
    if (lookahead == "a") {
      match("a"); parse_S(); match("b"); // \( S \rightarrow a S b \)
    } else if ((lookahead == "b") || (lookahead == \$))
      ; // \( S \rightarrow \epsilon \)
    else error();
  }
  ```

Left Factoring

What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

The Algorithm

\[ \text{If } A \in NT, \text{ find the longest prefix } \alpha \text{ that occurs in two or more right-hand sides of } A \]
\[ \text{if } \alpha \neq \epsilon \text{ then replace all of the } A \text{ productions, } A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n, \text{ with} \]
\[ A \rightarrow \alpha Z | \gamma \]
\[ Z \rightarrow \beta_1 | \beta_2 | \ldots | \beta_m \]
\[ \text{where } Z \text{ is a new element of NT} \]
Repeat until no common prefixes remain

Left Factoring

A graphical explanation for the same idea
Left Factoring

Consider the following fragment of the expression grammar

\[
\text{Factor} \rightarrow \text{Identifier} \\
| \text{Factor} \{ \text{ExpList} \} \\
| \text{Factor} \{ \text{ExpList} \} \\
| \epsilon
\]

After left factoring, it becomes

\[
\text{Factor} \rightarrow \text{Identifier} \{ \text{ExpList} \} \\
| \{ \text{ExpList} \} \\
| \epsilon
\]

This form has the same syntax, with the LL(1) property.

LL(1) Example - Grammar & Left Factoring

<table>
<thead>
<tr>
<th>Original Grammar</th>
<th>Left Factored Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Goal} \rightarrow \text{Expr} \phantom{\text{Factor Term}'}</td>
<td>\text{Goal} \rightarrow \text{Expr}</td>
</tr>
<tr>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr}</td>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr'}</td>
</tr>
<tr>
<td>\text{Term} \rightarrow \text{Factor Term}</td>
<td>\text{Term} \rightarrow \text{Factor Term'}</td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \text{id}</td>
<td>\text{Factor} \rightarrow \text{id}</td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \epsilon</td>
<td>\text{Factor} \rightarrow \epsilon</td>
</tr>
</tbody>
</table>

LL(1) Example - First Sets

<table>
<thead>
<tr>
<th>Grammar</th>
<th>FIRST Sets</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Goal} \rightarrow \text{Expr} \phantom{\text{Factor Term}'}</td>
<td>\text{Goal} \rightarrow \text{Expr} { \text{num, id} }</td>
<td>\text{Goal} \rightarrow \text{num, id}</td>
</tr>
<tr>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr}</td>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr'} { \text{num, id} }</td>
<td>\text{Expr} \rightarrow \text{num, id}</td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \epsilon</td>
<td>\text{Factor} \rightarrow \epsilon</td>
<td></td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \text{id}</td>
<td>\text{Factor} \rightarrow \text{id}</td>
<td></td>
</tr>
</tbody>
</table>

LL(1) Example - Follow Sets

<table>
<thead>
<tr>
<th>Grammar</th>
<th>FIRST Sets</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Goal} \rightarrow \text{Expr}</td>
<td>\text{Goal} \rightarrow { \text{num, id} } { \text{$} }</td>
<td>\text{Follow Sets}</td>
</tr>
<tr>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr}</td>
<td>\text{Expr} \rightarrow { \text{num, id} } { \text{$} }</td>
<td></td>
</tr>
<tr>
<td>\text{Term} \rightarrow \text{Factor Term}</td>
<td>\text{Term} \rightarrow { \text{$} }</td>
<td></td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \text{id}</td>
<td>\text{Factor} \rightarrow { \text{$} }</td>
<td></td>
</tr>
</tbody>
</table>

LL(1) Example - LL(1) Table

<table>
<thead>
<tr>
<th>Grammar</th>
<th>FIRST Sets</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Goal} \rightarrow \text{Expr}</td>
<td>\text{Goal} \rightarrow { \text{$} }</td>
<td>\text{Follow Sets}</td>
</tr>
<tr>
<td>\text{Expr} \rightarrow \text{Term} \text{ Expr}</td>
<td>\text{Expr} \rightarrow { \text{$} }</td>
<td></td>
</tr>
<tr>
<td>\text{Term} \rightarrow \text{Factor Term}</td>
<td>\text{Term} \rightarrow { \text{$} }</td>
<td></td>
</tr>
<tr>
<td>\text{Factor} \rightarrow \text{id}</td>
<td>\text{Factor} \rightarrow { \text{$} }</td>
<td></td>
</tr>
</tbody>
</table>
Question
By eliminating left recursion and left factoring, can we transform an arbitrary CFG to a form where it meets the $LL(1)$ condition?
(and can be parsed predictively with a single token lookahead?)

Answer
Given a CFG that doesn’t meet the $LL(1)$ condition, it is undecidable whether or not an equivalent $LL(1)$ grammar exists.

Example
$(a^n b^n \mid n \geq 1) \cup (a^n b^{2n} \mid n \geq 1)$ has no $LL(1)$ grammar

Left Factoring (Generality)

Language that Cannot Be $LL(1)$

Example
$G \rightarrow aAb$
\[ | aAbb \]
$A \rightarrow aAb$
\[ | \epsilon \]
$B \rightarrow aAbb$
\[ | \epsilon \]

Problem: need an unbounded number of $a$ characters before you can determine whether you are in the $A$ group or the $B$ group.