Predictive Parsers

Can we avoid backtracking? Yes, if for a given input symbol and given non-terminal, we can choose the alternative appropriately.

This is possible if the first terminal of every alternative in a production is unique:

\[
A \rightarrow a B C \mid b B D \mid b c e
\]

B \rightarrow c \mid b c e

D \rightarrow d

Parsing an input "abcd" has no backtracking.

Left factoring to enable predication:

\[
A \rightarrow \alpha \beta \mid \alpha \gamma
\]

\[
A \rightarrow \alpha A' \mid \alpha
\]

\[
A' \rightarrow \beta \mid \gamma
\]

For predicative parsers, must eliminate left recursion

LL(k) Parsing

**LL(k)**

- L — left to right scan
- L — leftmost derivation
- k — k symbols of lookahead

In practice, k = 1

It is table-driven and efficient.

LL(k) Parser Structure

**Sample Parse Table**

| \(E\) | \(E\rightarrow TX E\) | \(E\rightarrow TX\) |
| \(X\) | \(X\rightarrow +E\) | \(X\rightarrow \epsilon\) |
| \(T\) | \(T\rightarrow int\ Y\) | \(T\rightarrow \{ E\}\) |
| \(Y\) | \(Y\rightarrow \times \ T\) | \(Y\rightarrow \epsilon\) |

Implementation with 2-D parse table:

- A row for each non-terminal
- A column for all possible terminals and \(\$\) (the end of input marker)
- Every table entry contains at most one production
- Required for a grammar to be LL(1)
- No backtracking

Fixed action for each (non-terminal, input symbol) combination

**Push RHS in Reverse Order**

\[X\rightarrow \text{symbol at the top of the syntax stack}\]

\[a\rightarrow \text{current input symbol}\]

Parsing based on \(X, a\):

- If \(X = a = \$, then parser halts with "success"
- If \(X = \# = \$, then input rejected
- If \(X \neq a\), then
  - Case (a): If \(X = T\), then parser halts with "talled," input rejected
  - Case (b): If \(X \in N, M[X,a] = \epsilon\), then input rejected
  - pop X and push RHS to stack in reverse order

Example:

\[
X \rightarrow aBcD$
\]

\[
B \rightarrow c$
\]

\[
D \rightarrow d$
\]

\[
S \rightarrow \$
\]
**LL(1) Grammars**

Remove left recursive and perform left factoring.

Given the grammar:

\[ E \rightarrow T + E | T \]
\[ T \rightarrow \text{int} \times T | \text{int} | (E) \]

The grammar has no left recursion but requires left factoring.

After rewriting grammar, we have:

\[ E \rightarrow TX \]
\[ X \rightarrow +E | \epsilon \]
\[ T \rightarrow \text{int} Y | (E) \]
\[ Y \rightarrow \times T | \epsilon \]

**LL(1) Parsing**

Input Tokens: \[ \text{int} \times \text{int} \]

Parse table:

<table>
<thead>
<tr>
<th>Input</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(E \rightarrow TX)</td>
<td>(E \rightarrow TX)</td>
<td>(E \rightarrow TX)</td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>(X \rightarrow +E)</td>
<td>(X \rightarrow \epsilon)</td>
<td>(X \rightarrow \epsilon)</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>(T \rightarrow \text{int} Y)</td>
<td>(T \rightarrow (E))</td>
<td>(T \rightarrow (E))</td>
<td>$</td>
</tr>
<tr>
<td>Y</td>
<td>(Y \rightarrow \times T)</td>
<td>(Y \rightarrow \epsilon)</td>
<td>(Y \rightarrow \epsilon)</td>
<td>$</td>
</tr>
</tbody>
</table>
### LL(1) Parsing

**Parse table**

<table>
<thead>
<tr>
<th></th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → ( E )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → * T Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>

**Input Tokens:**

- int * int $
LL(1) Parsing

Input Tokens:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E -&gt; TX</td>
<td>$</td>
<td>E -&gt; TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X -&gt; +E</td>
<td>X -&gt; E</td>
<td>X -&gt; E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T -&gt; int Y</td>
<td>T -&gt; E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y -&gt; * T</td>
<td>Y -&gt; E</td>
<td>Y -&gt; E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parse table

Action List

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>int</td>
<td>int $</td>
</tr>
<tr>
<td>T X S</td>
<td>int</td>
<td>* int $</td>
</tr>
<tr>
<td>int Y X S</td>
<td>int</td>
<td>* int $</td>
</tr>
<tr>
<td>int Y X S</td>
<td>* int</td>
<td>Y -&gt; * T</td>
</tr>
<tr>
<td>* T X S</td>
<td>* int</td>
<td>int $</td>
</tr>
<tr>
<td>T X S</td>
<td>int</td>
<td>int $</td>
</tr>
<tr>
<td>int Y X S</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>Y X S</td>
<td>0</td>
<td>Y -&gt; E</td>
</tr>
<tr>
<td>X S</td>
<td>0</td>
<td>X -&gt; E</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>Halt and accept</td>
</tr>
</tbody>
</table>

Accept!

Constructing the Parse Table

We need to know what non-terminals to place our productions in the table?

We know that we have restricted our grammars so that left recursion is eliminated and they have been left factored. That means that each production is uniquely recognizable by the first terminal that production would derive.

Thus, we can construct our table from 2 sets:

- For each symbol A, the set of terminals that can begin a string derived from A. This set is called the FIRST set of A
- For each non-terminal A, the set of terminals that can appear after a string derived from A is called the FOLLOW set of A
Constructing LL(1) Parse Table

To construct the parse table, we check each $A \rightarrow \alpha$:

- For each terminal $a$, $A \rightarrow \alpha$ adds $A \rightarrow \alpha$ to $M[A, a]$.
- If $\epsilon \in \text{First}(\alpha)$, then for each terminal $b$ in $\text{Follow}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
- If $\epsilon \in \text{First}(\alpha)$ and $\epsilon \in \text{Follow}(A)$, then add $A \rightarrow \alpha$ to $M[A, \epsilon]$.

Example

Grammar:

- $E \rightarrow \epsilon$
- $E \rightarrow T X$
- $T \rightarrow \text{int } Y \mid ( E )$
- $X \rightarrow + E$
- $X \rightarrow \epsilon$

First Set:

- $T \rightarrow \text{int }
- \epsilon$
- $X \rightarrow +
- \epsilon$
- $Y \rightarrow \ast$
- $Y \rightarrow \epsilon$

Follow Set:

- $T \rightarrow \text{int }$
- $X \rightarrow +$
- $Y \rightarrow \ast$
- $Y \rightarrow \epsilon$
- $E \rightarrow \epsilon$

Constructing LL(1) Parse Table

For each terminal $a = \text{First}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.

Grammar:

- $E \rightarrow \epsilon$
- $E \rightarrow T X$
- $T \rightarrow \text{int } Y \mid ( E )$
- $X \rightarrow + E$
- $X \rightarrow \epsilon$

First Set:

- $T \rightarrow \text{int }$
- $X \rightarrow +$
- $Y \rightarrow \ast$
- $Y \rightarrow \epsilon$

Follow Set:

- $T \rightarrow \text{int }$
- $X \rightarrow +$
- $Y \rightarrow \ast$
- $Y \rightarrow \epsilon$
- $E \rightarrow \epsilon$

10/17/2012

5
Constructing LL(1) Parse Table

If \( \epsilon \in \text{First}(\alpha) \), then for each terminal \( b \in \text{Follow}(A) \), add \( A \rightarrow \alpha \) to \( M[A, b] \).

Grammar:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow T \ X )</td>
<td>( E \rightarrow T \ X )</td>
<td>( E \rightarrow T \ X )</td>
</tr>
<tr>
<td>( X \rightarrow + \ E )</td>
<td>( X \rightarrow + \ E )</td>
<td>( X \rightarrow + \ E )</td>
</tr>
<tr>
<td>( T \rightarrow \text{int} \ Y )</td>
<td>( T \rightarrow \text{int} \ Y )</td>
<td>( T \rightarrow \text{int} \ Y )</td>
</tr>
<tr>
<td>( Y \rightarrow * T )</td>
<td>( Y \rightarrow * T )</td>
<td>( Y \rightarrow * T )</td>
</tr>
<tr>
<td>( Y \rightarrow \epsilon )</td>
<td>( Y \rightarrow \epsilon )</td>
<td>( Y \rightarrow \epsilon )</td>
</tr>
</tbody>
</table>

Is a Grammar LL(1)?

Observation

If a grammar is LL(1), then each of its LL(1) table entries contain at most one rule. Otherwise, it is not LL(1).

Two methods to determine if a grammar is LL(1) or not:

1. Construct LL(1) table, and check if there is a multi-rule entry or
2. Check each rule as if the table were being constructed:

\( G \) is LL(1) iff for a rule \( A \rightarrow \alpha | \beta \):

a) \( \text{First}(\alpha) \cap \text{First}(\beta) = \emptyset \)

b) at most one of \( \alpha \) and \( \beta \) can derive \( \epsilon \)

c) If \( \beta \) derives \( \epsilon \), then \( \text{First}(\alpha) \cap \text{Follow}(\beta) = \emptyset \)

Removing Ambiguity

To remove ambiguity, it is possible to rewrite the grammar.

For the “if-then-else” example, how to rewrite?

May not even need to rewrite in this case, we can just use the \( X \rightarrow \text{else} \ S \) production over the \( X \rightarrow \epsilon \).

However, by changing the grammar:

- It might make the other phases of the compiler more difficult
- It becomes harder to determine semantics and generate code
- It is less appealing to programmers

Ambiguous Grammars

Some grammars may need more than one token of lookahead (k). However, some grammars are not LL regardless of how the grammar is changed.

\( S \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S \text{ else } S \mid \ldots \)

\( C \rightarrow b \)

change to

\( S \rightarrow \text{if } C \text{ then } S \ X \mid \ldots \)

\( X \rightarrow \text{else} \ S \mid \epsilon \)

\( C \rightarrow b \)

problem sentence: “if b then if b then a else a”

“else” \( \in \text{First}(X) \)

First(\( X \)) \( \subset \text{Follow}(S) \)

X \( \rightarrow \text{else} \ S \mid \epsilon \)

“else” \( \in \text{Follow}(X) \)

LL(1) Summary

LL(1) parsers operate in linear time and at most linear space relative to the length of input because:

Time — each input symbol is processed constant number of times

Space — stack is smaller than the input (in case we remove \( X \rightarrow \epsilon \))
Summary

First and Follow sets are used to construct predictive parsing tables

Intuitively, First and Follow sets guide the choice of rules:
- For non-terminal $A$ and input $t$, use a production rule $A \rightarrow \alpha$ where $t \in \text{First}(\alpha)$
- For non-terminal $A$ and input $t$, if $A \rightarrow \alpha$ and $t \in \text{Follow}(A)$, use the production $A \rightarrow \alpha$ where $\epsilon \in \text{First}(\alpha)$

What is $LL(0)$?

Why are $LL(2)$ ... $LL(k)$ not widely used?