Lecture 6. Parsing (syntax analysis)

Wei Le

2015.9
The parsing problem: generate a parse tree whose in order traversal is the string

- top down parsers
- recursive descendant parsers
  - a naive recursive descendant parser: try production rules to generate a string to see if the prefix matches the prefix of the token stream
  - predictive parsers
Predictive Parsers

- A predictive parser is a recursive descent parser that does not require backtracking.
- Like recursive-descent but parser can predict which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
  - In practice, LL(1) is used
LL(1) Languages

- In recursive-descent parsers, for each non-terminal and input token there may be a choice of production.
- LL(1) means that for each non-terminal and token there is only one production that could lead to success:
  - Can be specified as a 2D table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production
  - At any time, you select a rule to build the next children for the parse tree, you have only one choice (with the help of look ahead the next token). Here, the non-terminal is an internal node of the parse tree (parent)
What if there are common prefixes?

- Recall the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

- Impossible to predict because
  - For \( T \) two productions start with \text{int}
  - For \( E \) it is not clear how to predict

- A grammar must be \text{left-factored} before use for predictive parsing
Left-Factoring Example

- Recall the grammar
  \[
  E \rightarrow T + E \mid T \\
  T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
  \]

- Factor out common prefixes of productions
  \[
  E \rightarrow T \times \\
  X \rightarrow + E \mid \epsilon \\
  T \rightarrow (E) \mid \text{int} \ Y \\
  Y \rightarrow \ast T \mid \epsilon
  \]
Table Driven Predictive Parser

- Automatically compute PREDICT table (also called Parsing Table) from grammar
- \( \text{PREDICT(} \text{nonterminal, input-token}) \Rightarrow \text{right hand side} \)
LL(1)

- If PREDICT table has at most one entry per cell
- Then the grammar is LL(1)
- There is always exactly one right choice (So its fast to parse and easy to implement)
- If multiple entries in each cell
  - Ex: common prefixes, ambiguous
  - Can rewrite grammar (sometimes)
  - Can patch table manually, if you know what to do
  - Or can use more powerful parsing technique
LL(1) Parsing Table: an Example

- **Left-factored grammar**
  
  \[
  
  \begin{align*}
  E & \rightarrow TX \\
  T & \rightarrow (E) | \text{int } Y \\
  X & \rightarrow +E | \varepsilon \\
  Y & \rightarrow *T | \varepsilon
  \end{align*}
  
  \]

- **The LL(1) parsing table** ($\$ \text{ is a special end marker}$):

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+E</td>
<td></td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*T</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>
Table Driven Predictive Parser
LL(1) Parsing Table: How to use it?

• Consider the [E, int] entry
  – “When current non-terminal is E and next input is int, use production $E \rightarrow TX$
  – This production can generate an int in the first place

• Consider the [Y,+] entry
  – “When current non-terminal is Y and current token is +, get rid of Y”
  – We’ll see later why this is so
LL(1) Parsing Table Example

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - “There is no way to derive a string starting with * from non-terminal E”
LL(1) Parsing Algorithms

- Method similar to recursive descent, except
  - For each non-terminal $S$
  - We look at the next token $a$
  - And choose the production shown at $[S, a]$
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input
initialize stack = $S$ and next (pointer to tokens)
repeat
  case stack of
    $X$, rest : if $T[X, *next] = Y_1 \ldots Y_n$
               then stack $\leftarrow Y_1 \ldots Y_n$ rest;
               else error ();
    t, rest   : if $t == *next$ ++
               then stack $\leftarrow$ rest;
               else error ();
until stack $== <$ >
### LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>int * int $</td>
<td>$T X$</td>
</tr>
<tr>
<td>$T X$</td>
<td>int * int $</td>
<td>int $</td>
</tr>
<tr>
<td>int $Y X$</td>
<td>int * int $</td>
<td>* $T$</td>
</tr>
<tr>
<td>$Y X$</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>$* T X$</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>$T X$</td>
<td>int $</td>
<td>int $</td>
</tr>
<tr>
<td>int $Y X$</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>$Y X$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$X$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
Challenge: Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

- We want to generate parsing tables from CFG
Top Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

int * int + int
Top Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
int * T
    +
  E
```

- The leaves at any point form a string $\beta A \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$

```
int * int + int
```
Top Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
  E
   +
  T
```

- The leaves at any point form a string $\beta A_\gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b\delta$
  - The prefix $\beta$ matches
  - The next token is $b$

```
int * int + int
```
Top Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
  E
 / \
/   \
T   E
 /   \
| int * T
 |   \
| int \
|   int
```

- The leaves at any point form a string $\beta A_\gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$
Constructing Predictive Parsing Tables

- Consider the state $S \rightarrow^* \beta A \gamma$
  - With $b$ the next token
  - Trying to match $\beta b \delta$

There are two possibilities:

1. $b$ belongs to an expansion of $A$
   - Any $A \rightarrow \alpha$ can be used if $b$ can start a string derived from $\alpha$

In this case we say that $b \in \text{First}(\alpha)$

Or...
2. \textit{b does not belong to an expansion of A}
   - The expansion of \textit{A} is empty and \textit{b} belongs to an expansion of \textit{γ} (e.g., \textit{bω})
   - Means that \textit{b} can appear after \textit{A} in a derivation of the form \textit{S →* βAbω}
   - We say that \textit{b ∈ Follow(A)} in this case

   - What productions can we use in this case?
     • Any \textit{A → α} can be used if \textit{α} can expand to \textit{ε}
     • We say that \textit{ε ∈ First(A)} in this case
Computing First Sets

Definition \( \text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon \} \)

1. \( \text{First}(b) = \{ b \} \)

2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   
   - Add \( \text{First}(A_1) - \{\varepsilon\} \) to \( \text{First}(X) \). Stop if \( \varepsilon \notin \text{First}(A_1) \)
   
   - Add \( \text{First}(A_2) - \{\varepsilon\} \) to \( \text{First}(X) \). Stop if \( \varepsilon \notin \text{First}(A_2) \)
   
   - \( \ldots \)
   
   - Add \( \text{First}(A_n) - \{\varepsilon\} \) to \( \text{First}(X) \). Stop if \( \varepsilon \notin \text{First}(A_n) \)
   
   - Add \( \varepsilon \) to \( \text{First}(X) \)

   (ignore \( A_i \) if it is \( X \))
First Sets Example

• Recall the grammar

\[ E \rightarrow T X \]
\[ T \rightarrow ( E ) \mid \text{int} \ Y \]
\[ X \rightarrow + E \mid \varepsilon \]
\[ Y \rightarrow * T \mid \varepsilon \]

• First sets

\[ \text{First}( ( ) ) = \{ ( ) \} \]
\[ \text{First}( ( ) ) = \{ ( ) \} \]
\[ \text{First}( \text{int} ) = \{ \text{int} \} \]
\[ \text{First}( + ) = \{ + \} \]
\[ \text{First}( * ) = \{ * \} \]
\[ \text{First}( T ) = \{ \text{int}, ( ) \} \]
\[ \text{First}( E ) = \{ \text{int}, ( ) \} \]
\[ \text{First}( X ) = \{ +, \varepsilon \} \]
\[ \text{First}( Y ) = \{ *, \varepsilon \} \]
Computing Follow Sets

Definition \( \text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b \omega \} \)

1. Compute the First sets for all non-terminals first
2. Add \$\ to \text{Follow}(S) (if \( S \) is the start non-terminal)

3. For all productions \( Y \rightarrow \ldots X A_1 \ldots A_n \)
   - Add First\( (A_1) - \{\varepsilon\} \) to \text{Follow}(X). Stop if \( \varepsilon \not\in \text{First}(A_1) \)
   - Add First\( (A_2) - \{\varepsilon\} \) to \text{Follow}(X). Stop if \( \varepsilon \not\in \text{First}(A_2) \)
   - \ldots
   - Add First\( (A_n) - \{\varepsilon\} \) to \text{Follow}(X). Stop if \( \varepsilon \not\in \text{First}(A_n) \)
   - Add \text{Follow}(Y) to \text{Follow}(X)
Follow Sets Example

- Recall the grammar
  \[ E \rightarrow TX \]
  \[ T \rightarrow (E) | \text{int} \ Y \]
  \[ X \rightarrow +E | \varepsilon \]
  \[ Y \rightarrow \ast T | \varepsilon \]
- Follow sets
  \[
  \text{Follow(} + \text{)} = \{ \text{int, (} \} \\
  \text{Follow(} ( \text{)} = \{ \text{int, (} \} \\
  \text{Follow(} \times \text{)} = \{ \$, ) \} \\
  \text{Follow(} \text{)} = \{ +, ) , \$ \} \\
  \text{Follow(} \text{int}) = \{ \ast, + , ) , \$ \}
  \]
Constructing a Parsing Table

- In the first row, list all the terminals
- In the first column, list all the non-terminals
- Compute First Sets and Follow Sets for all the non-terminals (and thus terminals)
- For each production rule, if the right side is not $\epsilon$, if the terminal listed in the first row is in the First Sets of the nonterminal listed in the first column, put the production rules in the tabular
- For each production rule, if the right side is $\epsilon$, if the terminal listed in the first row is in the Follow Sets of the nonterminal listed in the first column, put the production rules in the tabular
Constructing a Parsing Table

- Construct a parsing table $T$ for CFG $G$

- For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\alpha \Rightarrow^* \epsilon$, for each $b \in \text{Follow}(A)$ do
    - $T[A, b] = \alpha$
Constructing a Parsing Table

• Recall the grammar
  
  $E \rightarrow T \times$
  $T \rightarrow (E) \mid \text{int} \; Y$
  $X \rightarrow + \; E \mid \varepsilon$
  $Y \rightarrow * \; T \mid \varepsilon$

  • Where in the line of $Y$ we put $Y \rightarrow * \; T$?
    - In the lines of First($*T$) = { * }

  • Where in the line of $Y$ we put $Y \rightarrow \varepsilon$?
    - In the lines of Follow($Y$) = { $, +, )$ }
LL(1) Languages

- If any entry is multiply defined then $G$ is not LL(1)
  - If $G$ is ambiguous
  - If $G$ is left recursive
  - If $G$ is not left-factored
  - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables
• For some grammars there is a simple parsing strategy
  - Predictive parsing (LL(1))
  - Once you build the LL(1) table, you can write the parser by hand

- A predictive parser runs in linear time.
- Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn.
- Parsers that use recursive descent with backtracking may require exponential time.
Review: terms so far

- LL(k) parser
- LL(1) grammar
- LL(1) language
- Parsing table
- Top down parsing
- Recursive decedent parser
- Predictive parser