Top-Down Parsing

Outline

- Recursive Descent Parsing
- Left Recursion
- LL(1) Parsing
  - LL(1) Parsing Tables
  - LP(1) Parsing Algorithm
- Constructing LL(1) Parsing Tables
  - First, Follow

In One Slide

- An LL(1) parser reads tokens from left to right and constructs a top-down leftmost derivation. LL(1) parsing is a special case of recursive descent parsing in which you can predict which single production to use from one token of lookahead. LL(1) parsing is fast and easy, but it does not work if the grammar is ambiguous, left-recursive, or not left-factored (i.e., it does not work for most programming languages).
Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:
  \[ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \]
  The parse tree is constructed
  - From the top
  - From left to right

Recursive Descent Parsing

- We’ll try recursive descent parsing first
  - “Try all productions exhaustively, backtrack”
- Consider the grammar
  \[
  E \rightarrow T + E \mid T \\
  T \rightarrow ( E ) \mid \text{int} \mid \text{int } T
  \]
- Token stream is: \text{int } \text{int}
- Start with top-level non-terminal \text{E}
- Try the rules for \text{E} in order

Recursive Descent Example

- Try \( E_0 \rightarrow T_1 + E_2 \)
- Then try a rule for \( T_1 \rightarrow ( E_3 ) \)
  - But ( does not match input token \text{int} \\
- Try \( T_1 \rightarrow \text{int} \). Token matches.
  - But + after \( T_1 \) does not match input token * \\
- Try \( T_1 \rightarrow \text{int } T_2 \)
  - This will match but + after \( T_1 \) will be unmatched
- Have exhausted the choices for \( T_1 \)
  - Backtrack to choice for \( E_0 \)
Recursive Descent Example (2)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for $T_1$
  - And succeed with $T_1 \rightarrow \text{int} \ast T_2$ and $T_2 \rightarrow \text{int}$
  - With the following parse tree

```
E_0
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
T_1
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
int
```

Recursive Descent Parsing

- Parsing: given a string of tokens $t_1 t_2 \ldots t_n$, find its parse tree
- **Recursive descent parsing**: Try all the productions exhaustively
  - At a given moment the **fringe** of the parse tree is: $t_1 t_2 \ldots t_k A \ldots$
  - Try all the productions for $A$: if $A \rightarrow BC$ is a production, the new fringe is $t_1 t_2 \ldots t_k B C \ldots$
  - **Backtrack** when the fringe doesn’t match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S a$:
  - In the process of parsing $S$ we try the above rule
  - What goes wrong?

- A **left-recursive grammar** has
  
  $S \rightarrow \ast S \alpha$

  for some $\alpha$

  Recursive descent does not work in such cases
  - It goes into an $\infty$ loop
Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha \mid \beta \]

• \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \).

• Can rewrite using \textit{right-recursion}
  \[ S \rightarrow \beta \ T \\
  T \rightarrow \alpha \ T \mid \varepsilon \]

Example of Eliminating Left Recursion

• Consider the grammar
  \[ S \rightarrow 1 \mid S \ 0 \quad (\beta = 1 \text{ and } \alpha = 0) \]

  It can be rewritten as
  \[ S \rightarrow 1 \ T \\
  T \rightarrow 0 \ T \mid \varepsilon \]

More Left Recursion Elimination

• In general
  \[ S \rightarrow S \alpha_1 \mid \ldots \mid S \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \]

  • All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \) and continue with several instances of \( \alpha_1, \ldots, \alpha_n \).

  • Rewrite as
    \[ S \rightarrow \beta_1 \ T \mid \ldots \mid \beta_m \ T \\
    T \rightarrow \alpha_1 \ T \mid \ldots \mid \alpha_n \ T \mid \varepsilon \]
General Left Recursion

• The grammar
  \[ S \to A \alpha \mid \delta \]
  \[ A \to S \beta \]
  is also left-recursive because
  \[ S \to^* S \beta \alpha \]
• This left-recursion can also be eliminated
• See book, Section 2.3
• Detecting and eliminating left recursion are popular test questions

Summary of Recursive Descent

• Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - ... but that can be done automatically
• Unpopular because of backtracking
  - Thought to be too inefficient
• Often, we can avoid backtracking ...

Predictive Parsers

• 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
  - First L means “left-to-right” scan of input
  - Second L means “leftmost derivation”
  - The k means “predict based on k tokens of lookahead”
• In practice, LL(1) is used
Sometimes Things Are Perfect

- The “.ml-lex” format you emit in PA2
- Will be the input for PA3
  - actually the reference “.ml-lex” will be used
- It can be “parsed” with no lookahead
  - You always know just what to do next
- Ditto with the “.ml-ast” output of PA3
- Just write a few mutually-recursive functions
- They read in the input, one line at a time

LL(1)

- In recursive descent, for each non-terminal and input token there may be a choice of which production to use
- 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
  - Each table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar
  \[
  E \rightarrow T + E | T \\
  T \rightarrow \text{int} | \text{int} \ast T | (E)
  \]
- Impossible to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- A grammar must be 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
  \[ E \rightarrow TX \]
  \[ X \rightarrow +E \mid \varepsilon \]
  \[ T \rightarrow (E) \mid \text{int} Y \]
  \[ Y \rightarrow \ast T \mid \varepsilon \]

LL(1) Parsing Table Example

• Left-factored grammar
  \[ E \rightarrow TX \]
  \[ X \rightarrow +E \mid \varepsilon \]
  \[ T \rightarrow (E) \mid \text{int} Y \]
  \[ Y \rightarrow \ast T \mid \varepsilon \]

  • The LL(1) parsing table ($ is a special end marker):

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

LL(1) Parsing Table Example Analysis

• Consider the \([E, \text{int}]\) entry
  - “When current non-terminal is \(E\) and next input is \(\text{int}\), use production \(E \rightarrow T X\)”
  - This production can generate an \(\text{int}\) in the first position

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>*</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td>(E)</td>
<td>$</td>
</tr>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td>TX</td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>+E</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>Y</td>
<td>\ast T</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>
• 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

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

LL(1) Parsing Tables: Errors

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

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

Using Parsing Tables

• 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
LL(1) Parsing Algorithm

initialize stack = <S $>
next = (pointer to tokens)
repeat
match stack with
| <X, rest>: if T[X,*next] = Y₁...Yₙ
    then stack ← <Y₁...Yₙ rest>
    else error ()
| <t, rest>: if t == *next ++
    then stack ← <rest>
    else error ()
until stack == < >

<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 Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

LL(1) Languages

- **LL(1) languages** can be LL(1) parsed
  - A language Q is LL(1) if there exists an LL(1) table such the LL(1) parsing algorithm using that table accepts exactly the strings in Q
- No table entry can be **multiply defined**
- Once we have the table
  - The parsing algorithm is **simple and fast**
  - No backtracking is necessary
- Want to generate parsing tables from CFG!
Q: Movies (263 / 842)
• This 1982 Star Trek film features Spock nerve-pinching McCoy, Kirstie Alley "losing" the Kobayashi Maru, and Chekov being mind-controlled by a slug-like alien. Ricardo Montalban is "is intelligent, but not experienced. His pattern indicates two-dimensional thinking."

Q: Music (238 / 842)
• For two of the following four lines from the 1976 Eagles song Hotel California, give enough words to complete the rhyme.
  - So I called up the captain / "please bring me my wine"
  - Mirrors on the ceiling / pink champagne on ice
  - And in the master's chambers /

Q: Books (727 / 842)
• Name 5 of the 9 major characters in A. A. Milne's 1926 books about a "bear of very little brain" who composes poetry and eats honey.
Top-Down Parsing. Review
• Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

\[ E \rightarrow T \cdot E \]

\[ int \cdot int \cdot int \]

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

\[ E \rightarrow T \cdot E \]

\[ int \cdot int \cdot int \]
Top-Down Parsing. Review

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

  ```plaintext
  E
  /   \
T     E
/     /  \
int   int   int
```

- The leaves at any point form a string \( \beta \gamma \)
  - \( \beta \) contains only terminals
  - The input string is \( \beta \delta \)
  - 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 \( \beta \delta \)

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. \( b \) does not belong to an expansion of \( A \)
   - The expansion of \( A \) is empty and \( b \) belongs to an expansion of \( \gamma \) (e.g., \( b\omega \))
   - Means that \( b \) can appear after \( A \) in a derivation of the form \( S \rightarrow^* \beta A b \omega \)
   - We say that \( b \in \text{Follow}(A) \) in this case

   - What productions can we use in this case?
     - Any \( A \rightarrow \alpha \) can be used if \( \alpha \) can expand to \( \varepsilon \)
     - We say that \( \varepsilon \in \text{First}(A) \) in this case
Computing First Sets

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

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

2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \notin \text{First}(A_1) \)
   - Add \( \text{First}(A_2) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \notin \text{First}(A_2) \)
   - …
   - Add \( \text{First}(A_n) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \notin \text{First}(A_n) \)
   - Add \( \epsilon \) to \( \text{First}(X) \)
     (ignore \( A_i \) if it is \( X \))

Example First Set Computation

- Recall the grammar
  - \( E \rightarrow TX \)
  - \( T \rightarrow (E) \mid \text{int} Y \)
  - \( X \rightarrow +E \mid \epsilon \)
  - \( Y \rightarrow *T \mid \epsilon \)
- First sets
  - \( \text{First( \() = \{ \) \} \)
  - \( \text{First( )} = \{ \) \}
  - \( \text{First( int)} = \{ \text{int} \} \)
  - \( \text{First( )} = \{ \) \}
  - \( \text{First( new)} = \{ \) \}
  - \( \text{First( T)} = \{ \text{int, new} \} \)
  - \( \text{First( E)} = \{ \text{int, new} \} \)
  - \( \text{First( X)} = \{ +, \epsilon \} \)
  - \( \text{First( Y)} = \{ *, \epsilon \} \)
  - \( \text{First( *)} = \{ *, \epsilon \} \)

Computing Follow Sets

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

1. Compute the First sets for all non-terminals first
2. Add \( S \) 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 \( \text{First}(A_1) - \{ \epsilon \} \) to \( \text{Follow}(X) \). Stop if \( \epsilon \notin \text{First}(A_1) \)
   - Add \( \text{First}(A_2) - \{ \epsilon \} \) to \( \text{Follow}(X) \). Stop if \( \epsilon \notin \text{First}(A_2) \)
   - …
   - Add \( \text{First}(A_n) - \{ \epsilon \} \) to \( \text{Follow}(X) \). Stop if \( \epsilon \notin \text{First}(A_n) \)
   - Add \( \text{Follow}(Y) \) to \( \text{Follow}(X) \)
Example Follow Set Computation

- Recall the grammar
  \[
  \begin{align*}
  E &\rightarrow TX & X &\rightarrow +E &\epsilon \\
  T &\rightarrow (E) &\text{int} &Y &\rightarrow +T &\epsilon
  \end{align*}
  \]
- Follow sets
  \[
  \begin{align*}
  \text{Follow}(+) = \{\text{int}, (\} \\
  \text{Follow}(* ) = \{\text{int}, (\} \\
  \text{Follow}(\text{int}) = \{\text{int}, (\} \\
  \text{Follow}(E) = \{\} \cup \{\$\} \\
  \text{Follow}(T) = \{+, (\} \cup \{\$\} \\
  \text{Follow}(X) = \{\} \\
  \text{Follow}(Y) = \{+, (\} \cup \{\$\}
  \end{align*}
  \]

Constructing LL(1) Parsing Tables

- Here is how to construct a parsing table \( T \) for context-free grammar \( G \)
  - For each production \( A \rightarrow \alpha \) in \( G \) do:
    - For each terminal \( b \in \text{First}(\alpha) \) do:
      - \( T[A, b] = \alpha \)
    - If \( \alpha \rightarrow \star \epsilon \), for each \( b \in \text{Follow}(A) \) do:
      - \( T[A, b] = \alpha \)

LL(1) Table Construction Example

- Recall the grammar
  \[
  \begin{align*}
  E &\rightarrow TX & X &\rightarrow +E &\epsilon \\
  T &\rightarrow (E) &\text{int} &Y &\rightarrow +T &\epsilon
  \end{align*}
  \]
- Where in the row of \( Y \) do we put \( Y \rightarrow \star T \)?
  - In the columns of First( \( \star \) ) = \{\( \star \}\}
LL(1) Table Construction Example

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

- Where in the row of \( Y \) we put \( Y \rightarrow \varepsilon \)?
  
  - In the columns of \( \text{Follow}(Y) = \{ \$, +, ) \} \)

<table>
<thead>
<tr>
<th>( \text{int} )</th>
<th>( * )</th>
<th>( + )</th>
<th>(</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int} Y )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E )</td>
<td>( TX )</td>
<td>( TX )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X )</td>
<td>( * E )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td></td>
</tr>
<tr>
<td>( Y )</td>
<td>( * T )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td></td>
</tr>
</tbody>
</table>

Avoid Multiple Definitions!

Notes on LL(1) Parsing Tables

- 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) (e.g., Java, Ruby, C++)
- There are tools that build LL(1) tables
Simple Parsing Strategies

- **Recursive Descent Parsing**
  - But backtracking is too annoying, etc.

- **Predictive Parsing, aka. LL(k)**
  - Predict production from k tokens of lookahead
  - Build LL(1) table
  - Parsing using the table is fast and easy
  - But many grammars are not LL(1) (or even LL(k))

- **Next: a more powerful parsing strategy for grammars that are not LL(1)**

---

Homework

- **Today: WA1 (written homework) due**
  - Turn in to drop-box by 1pm.

- **Friday: PA2 (Lexer) due**
  - You may work in pairs.

- **Next Tuesday: Chapters 2.3.3**
  - Optional Wikipedia article