Top-Down Parsing Algorithms

Matthew Dwyer
324E Nichols Hall
dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer
Implementing parsers

• Two approaches
  – Top-down
  – Bottom-up
• Today: Top-Down
  – Easier to understand and program manually
• Then: Bottom-Up
  – More powerful and used by most parser generators
Intro to Top-Down Parsing

• The parse tree is constructed
  – From the top
  – From left to right

• Terminals are seen in order of appearance in the token stream:

```
  t2 3  t9
 /     \   \
4      7    \\
 /     \   \\n t5  t6 t8
```

Recursive Descent Parsing

• Consider the grammar

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
\]

• Token stream is: \( \text{int}_5 \ast \text{int}_2 \)

• Start with top-level non-terminal \( E \)

• Try the rules for \( E \) in order
Recursive Descent Parsing. Example (Cont.)

• Try \( E_0 \rightarrow T_1 + E_2 \)

• Then try a rule for \( T_1 \rightarrow ( E_3 ) \)
  – But \( ( \) does not match input token \( \text{int}_5 \)

• 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
Recursive Descent Parsing.

Example (Cont.)

• Has exhausted the choices for $T_1$
  – Backtrack to choice for $E_0$

• Try $E_0 \rightarrow T_1$

• Follow same steps as before for $T_1$
  – and succeed with $T_1 \rightarrow \text{int} \cdot T_2$
  and $T_2 \rightarrow \text{int}$
A Recursive Descent Parser.

Preliminaries

• Let TOKEN be the type of tokens
  – Special tokens INT, OPEN, CLOSE, PLUS, TIMES

• Let the global next point to the next token
A Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - A given token terminal
    
    ```
    bool term(TOKEN tok) { return *next++ == tok; }
    ```
  - A given production of S (the n\textsuperscript{th})
    
    ```
    bool S_n() { ... }
    ```
  - Any production of S:
    
    ```
    bool S() { ... }
    ```

- These functions advance next
A Recursive Descent Parser (3)

• For production $E \rightarrow T$
  
  ```java
  bool E1() { return T(); }
  ```

• For production $E \rightarrow T + E$
  
  ```java
  bool E2() { return T() && term(PLUS) && E(); }
  ```

• For all productions of $E$ (with backtracking)
  
  ```java
  bool E() {
    TOKEN *save = next;
    return (next = save, E1())
           || (next = save, E2());
  }
  ```

CIS 706 Translators I
A Recursive Descent Parser (4)

• Functions for non-terminal $T$

```c
bool T_1() { return term(OPEN) && E() && term(CLOSE); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(INT); }

bool T() {
    TOKEN *save = next;
    return (next = save, T_1())
            || (next = save, T_2())
            || (next = save, T_3()); }
```
Recursive Descent Parsing. Notes.

• To start the parser
  – Initialize next to point to first token
  – Invoke E()

• Notice how this simulates our previous example

• Easy to implement by hand
• But does not always work …
When Recursive Descent Does Not Work

• Consider a production $S \rightarrow S \ a$

```cpp
def s1():
    return s() and term(a)
def s():
    return s1()
```

• $S()$ will get into an infinite loop

• A left-recursive grammar has a non-terminal $S$

$S \rightarrow^+ S\alpha$ for some $\alpha$

• Recursive descent does not work in such cases
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 right-recursion

\[ S \rightarrow \beta S' \]
\[ S' \rightarrow \alpha S' \mid \varepsilon \]
More Elimination of Left-Recursion

• 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
  \[ S \rightarrow \beta_1 \ S' \mid \ldots \mid \beta_m \ S' \]
  \[ S' \rightarrow \alpha_1 \ S' \mid \ldots \mid \alpha_n \ S' \mid \epsilon \]
General Left Recursion

• The grammar
  \[ S \rightarrow A \alpha | \delta \]
  \[ A \rightarrow S \beta \]
• is also left-recursive because
  \[ S \rightarrow^+ S \beta \alpha \]
• This \textit{indirect} left-recursion can also be eliminated
• See book for general algorithm
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

• In practice, backtracking is eliminated by restricting the grammar
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
  – 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, 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
• Can be specified via 2D tables
  – One dimension for current non-terminal to expand
  – One dimension for next token
  – A table entry contains one production
Predictive Parsing and Left Factoring

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

• Hard 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
  \[ E \rightarrow T \ X \]
  \[ X \rightarrow + \ E \mid \varepsilon \]
  \[ T \rightarrow (E) \mid \text{int} \ Y \]
  \[ Y \rightarrow * \ T \mid \varepsilon \]
LL(1) Parsing Table Example

• Left-factored grammar

E → T X  X → + E | ε
T → ( E ) | int Y  Y → * T | ε

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(   )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+ E</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>
• 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 generates an int in the first place

• Consider the \([Y, +]\) entry
  – “When current non-terminal is \(Y\) and current token is +, get rid of \(Y\)”
  – \(Y\) can be followed by + only in a derivation in which \(Y \rightarrow \varepsilon\)
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”
Using Parsing Tables

• Method similar to recursive descent, except
  – For each non-terminal S
  – look at the next token a
  – choose the production shown at [S,a]
• use a stack to keep track of pending non-terminals
• reject when error state
• accept when end-of-input ($)
LL(1) Parsing Algorithm

initialize stack = <S, $> and next
repeat
  case stack of
    <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 == < >
## 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 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>
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

• We want to generate parsing tables from CFG
• If $A \rightarrow \alpha$, where in the row of $A$ does $\alpha$ go?
• In the column of $t$ where $t$ can start a string derived from $\alpha$
  \[ \alpha \rightarrow^* t \beta \]
  – We say that $t \in \text{First}(\alpha)$
• In the column of $t$ if $\alpha$ is $\varepsilon$ and $t$ can follow an $A$
  \[ S \rightarrow^* \beta A t \delta \]
  – We say $t \in \text{Follow}(A)$
Computing First Sets

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

Algorithm sketch (see book for details):
1. for all terminals \( t \) do \( \text{First}(t) \leftarrow \{ t \} \)
2. for each production \( X \rightarrow \varepsilon \) do \( \text{First}(X) \leftarrow \{ \varepsilon \} \)
3. if \( X \rightarrow A_1 \ldots A_n \alpha \) and \( \varepsilon \in \text{First}(A_i), \ 1 \leq i \leq n \) do
   • add \( \text{First}(\alpha) \) to \( \text{First}(X) \)
4. for each \( X \rightarrow A_1 \ldots A_n \) s.t. \( \varepsilon \in \text{First}(A_i), \ 1 \leq i \leq n \) do
   • add \( \varepsilon \) to \( \text{First}(X) \)
5. repeat steps 4 & 5 until no \( \text{First} \) set can be grown
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:

\[ \text{Follow}(X) = \{ t \mid S \rightarrow^* \beta \ X \ t \ \delta \} \]

• Intuition

– If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)
– If \( X \rightarrow A \ B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
– If \( B \rightarrow^* \varepsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
Computing Follow Sets (Cont.)

Algorithm sketch:
1. Follow(S) ← { $ }  
2. For each production A → α X β
   • add First(β) - {ε} to Follow(X)  
3. For each A → α X β where ε ∈ First(β)
   • add Follow(A) to Follow(X)  
   – repeat step(s) until no Follow set grows
Follow Sets. Example

• Recall the grammar
  
  \[\begin{align*}
  E & \rightarrow T X \\
  T & \rightarrow ( E ) \mid \text{int } Y \\
  \end{align*}\]

  \[\begin{align*}
  X & \rightarrow + E \mid \varepsilon \\
  Y & \rightarrow * T \mid \varepsilon \\
  \end{align*}\]

• Follow sets

  \[\begin{align*}
  \text{Follow( + )} & = \{ \text{int, ( } \} \\
  \text{Follow( * )} & = \{ \text{int, ( } \} \\
  \text{Follow( ( )} & = \{ \text{int, ( } \} \\
  \text{Follow( E )} & = \{ \}, \$ \} \\
  \text{Follow( X )} & = \{ \$, ) \} \\
  \text{Follow( T )} & = \{ +, ) , \$ \} \\
  \text{Follow( ) )} & = \{ +, ) , \$ \} \\
  \text{Follow( Y )} & = \{ +, ) , \$ \} \\
  \text{Follow( int) } & = \{ *, +, ) , \$ \} \\
  \end{align*}\]
Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production \( A \rightarrow \alpha \) in G do:
  - For each terminal \( t \in \text{First}(\alpha) \) do
    - \( T[A, t] = \alpha \)
  - If \( \varepsilon \in \text{First}(\alpha) \), for each \( t \in \text{Follow}(A) \) do
    - \( T[A, t] = \alpha \)
  - If \( \varepsilon \in \text{First}(\alpha) \) and \( \$ \in \text{Follow}(A) \) do
    - \( T[A, \$] = \alpha \)
Notes on LL(1) Parsing Tables

• If any entry is multiply defined then G is not LL(1)
  – If G is ambiguous, left-recursive, not left-factored, …
• Most programming language grammars are not LL(1)
• There are tools that build LL(1) tables
Review

• For some grammars there is a simple parsing strategy
  – Predictive parsing

• Next time: a more powerful parsing strategy that is commonly used