4 (c) parsing

- A grammar describes syntactically legal strings in a language
- A recogniser simply accepts or rejects strings
- A generator produces strings
- A parser constructs a parse tree for a string
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A recursive descent parser easily implements a top-down parser for simple grammars

Top down vs. bottom up parsing

- The parsing problem is to connect the root node $S$ with the tree leaves, the input
- **Top-down parsers**: starts constructing the parse tree at the top (root) and move down towards the leaves. Easy to implement by hand, but requires restricted grammars. E.g.:
  - Predictive parsers (e.g., LL(k))
- **Bottom-up parsers**: build nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handles larger class of grammars. E.g.:
  - shift-reduce parser (or LR(k) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).

Top down vs. bottom up parsing

- Both are general techniques that can be made to work for all languages (but not all grammars!)
- Recall that a given language can be described by several grammars
- Both of these grammars describe the same language

$$A = 1 + 3 \ast 4 / 5$$

- The first one, with it’s left recursion, causes problems for top down parsers
- For a given parsing technique, we may have to transform the grammar to work with it

$$E \rightarrow E + Num$$
$$E \rightarrow Num + E$$
$$E \rightarrow Num$$

- $$E \rightarrow E + Num$$
- $$E \rightarrow Num + E$$
- $$E \rightarrow Num$$
• How hard is the parsing task? How to we measure that?
• Parsing an arbitrary CFG is $O(n^3)$ -- it can take time proportional the cube of the number of input symbols
  • This is bad! (why?)
• If we constrain the grammar somewhat, we can always parse in linear time. This is good! (why?)
• Linear-time parsing
  – LL parsers
    • Recognize LL grammar
    • Use a top-down strategy
  – LR parsers
    • Recognize LR grammar
    • Use a bottom-up strategy

Top Down Parsing Methods
• Simplest method is a full-backup, recursive descent parser
• Often used for parsing simple languages
• Write recursive recognizers (subroutines) for each grammar rule
  – If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  – If rule fails, return failure. Caller may try another choice or fail
  – On failure it “backs up”

Top Down Parsing Methods: Problems
• When going forward, the parser consumes tokens from the input, so what happens if we have to back up? -- suggestions?
• Algorithms that use backup tend to be, in general, inefficient
  – There might be a large number of possibilities to try before finding the right one or giving up
• Grammar rules which are left-recursive lead to non-termination!

Recursive Decent Parsing: Example
For the grammar:

```
<term> -> <factor> {(*|/)<factor>}*
```

We could use the following recursive descent parsing subprogram (this one is written in C)

```c
void term() {
  factor();    /* parse first factor*/
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();  /* get next token */
    factor();   /* parse next factor */
  }
}
```
Problems

• Some grammars cause problems for top down parsers
• Top down parsers do not work with left-recursive grammars
  – E.g., one with a rule like: E -> E + T
  – We can transform a left-recursive grammar into one which is not
• A top down grammar can limit backtracking if it only has one rule per non-terminal
  – The technique of rule factoring can be used to eliminate multiple rules for a non-terminal

Left-recursive grammars

• A grammar is left recursive if it has rules like
  \[ X \rightarrow X \beta \]
• Or if it has indirect left recursion, as in
  \[ X \rightarrow A \beta \\
  A \rightarrow X \]
• Q: Why is this a problem?
  – A: it can lead to non-terminating recursion!

Left-recursive grammars

• Consider
  \[ E \rightarrow E + Num \\
  E \rightarrow Num \]
• We can manually or automatically rewrite a grammar removing left-recursion, making it ok for a top-down parser.

Elimination of Left Recursion

• Consider left-recursive grammar
  \[ S \rightarrow S \alpha \\
  S \rightarrow \beta \]
• S generates strings
  \[ \beta \alpha \alpha \alpha \ldots \]
• Rewrite using right-recursion
  \[ S \rightarrow \beta S' \\
  S' \rightarrow \alpha S' | \epsilon \]
• Concretely
  \[ T \rightarrow T + id \\
  T \rightarrow id \]
• T generates strings
  \[ id \]
  \[ id + id \]
  \[ id + id + id \ldots \]
• Rewrite using right-recursion
  \[ T \rightarrow id \\
  T \rightarrow id T \]
More Elimination of Left-Recursion

• In general
  \[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]
• 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
  \[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]
  \[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \epsilon \]

General Left Recursion

• The grammar
  \[ S \rightarrow A \alpha | \delta \]
  \[ A \rightarrow S \beta \]
  is also left-recursive because
  \[ S \rightarrow^* S \beta \alpha \]
  where \( \rightarrow^* \) means “can be rewritten in one or more steps”
• This indirect left-recursion can also be automatically eliminated

Summary of Recursive Descent

• Simple and general parsing strategy
  – Left-recursion must be eliminated first
  – \( \ldots \) but that can be done automatically
• Unpopular because of backtracking
  – Thought to be too inefficient
• In practice, backtracking is eliminated by further restricting the grammar to allow us to successfully predict which rule to use

Predictive Parsers

• That there can be many rules for a non-terminal makes parsing hard
• A predictive parser processes the input stream typically from left to right
  – Is there any other way to do it? Yes for programming languages!
• It uses information from peeking ahead at the \textit{upcoming terminal symbols} to decide which grammar rule to use next
• And \textit{always} makes the right choice of which rule to use
• How much it can peek ahead is an issue
Predictive Parsers

• An important class of predictive parser only peek ahead one token into the stream
• An an \textit{LL}(k) parser, does a \textit{Left-to-right} parse, a \textit{Leftmost}-derivation, and \textit{k-symbol} lookahead
• Grammars where one can decide which rule to use by examining only the \textit{next} token are \textit{LL}(1)
• \textit{LL}(1) grammars are widely used in practice
  – The syntax of a PL can usually be adjusted to enable it to be described with an \textit{LL}(1) grammar

Remember…

• Given a grammar and a string in the language defined by the grammar …
• There may be more than one way to \textit{derive} the string leading to the \textit{same parse tree}
  – It depends on the order in which you apply the rules
  – And what parts of the string you choose to rewrite next
• All of the derivations are \textit{valid}
• To simplify the problem and the algorithms, we often focus on one of two simple derivation strategies
  – A \textit{leftmost} derivation
  – A \textit{rightmost} derivation

\textbf{LL(k) and LR(k) parsers}

• Two important parser classes are \textit{LL}(k) and \textit{LR}(k)
• The name \textit{LL}(k) means:
  – \textit{L}: \textit{Left-to-right} scanning of the input
  – \textit{L}: Constructing \textit{leftmost} derivation
  – \textit{k}: max # of input symbols needed to predict parser action
• The name \textit{LR}(k) means:
  – \textit{L}: \textit{Left-to-right} scanning of the input
  – \textit{R}: Constructing \textit{rightmost} derivation in reverse
  – \textit{k}: max # of input symbols needed to select parser action
• A \textit{LR}(1) or \textit{LL}(1) parser never need to “\textit{look ahead}” more than \textit{one} input token to know what parser production rule applies

\textbf{Predictive Parser}

Example: consider the grammar

\begin{align*}
S & \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S & \rightarrow \text{begin } S \text{ L} \\
S & \rightarrow \text{print } E \\
L & \rightarrow \text{end} \\
L & \rightarrow ; S \text{ L} \\
E & \rightarrow \text{num } = \text{ num}
\end{align*}

An \textit{S} expression starts either with an \textit{IF}, \textit{BEGIN}, or \textit{PRINT} token, and an \textit{L} expression start with an \textit{END} or a \textit{SEMICOLON} token, and an \textit{E} expression has only one production.
Predictive Parsing and Left Factoring

- Consider the grammar:
  \[ E \rightarrow T + E \]
  \[ E \rightarrow T \]
  \[ T \rightarrow \text{int} \]
  \[ T \rightarrow \text{int} \times T \]
  \[ T \rightarrow (E) \]

- Hard to predict because:
  - For \( T \), two productions start with \( \text{int} \)
  - For \( E \), it is not clear how to predict which rule to use

- Must **left-factored** grammar before use for predictive parsing

- Left-factoring involves rewriting rules so that, if a non-terminal has > 1 rule, each begins with a **terminal**

Left Factoring

- Consider a rule of the form
  \[ A \rightarrow a B_1 | a B_2 | a B_3 | \ldots | a B_n \]
- A top-down parser generated from this grammar is not efficient as it requires backtracking.
- To avoid this problem we left factor the grammar.
  - Collect all productions with the same left-hand side and begin with the same symbols on the right-hand side
  - Combine common strings into a single production and append a new non-terminal to end of this new production
  - Create new productions using this new non-terminal for each of the suffixes to the common production.
- After left factoring the above grammar is transformed into:
  \[ A \rightarrow a A_1 \]
  \[ A_1 \rightarrow B_1 | B_2 | B_3 | \ldots | B_n \]

Left-Factoring Example

Add new non-terminals X and Y to factor out common prefixes of rules:

- For each non-terminal the revised grammar, there is either only one rule or every rule begins with a terminal or \( \varepsilon \)

Using Parsing Tables

- LL(1) means that for each non-terminal and token there is only **one** production
- Can be represented as a simple table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one rule’s action or empty if error
- Method similar to recursive descent, except
  - For each non-terminal \( S \)
  - We look at the next token \( a \)
  - And chose the production shown at table cell \([S, a]\)
- Use a stack to keep track of pending non-terminals
- Reject when we encounter an error state, accept when we encounter end-of-input
**LL(1) Parsing Table Example**

Left-factored grammar

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

The LL(1) parsing table

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

End of input symbol

**LL(1) Parsing Algorithm**

initialize stack = <S $> and next

repeat

case stack of
  <X, rest> : if T[X,*next] = Y_1...Y_n
              then stack ← <Y_1... Y_n rest>;
              else error();
  <t, rest> : if t == *next ++
              then stack ← <rest>;
              else error();

until stack == < >

where:
(1) next points to the next input token
(2) X matches some non-terminal
(3) t matches some terminal

**LL(1) Parsing Table Example**

Consider the [E, \text{int}] entry
- “When current non-terminal is E & next input \text{int}, use production \text{E} \rightarrow \text{T X}”
- It’s the only production that can generate an \text{int} in next 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 where Y \rightarrow \epsilon

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

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

**LL(1) Parsing Example**

Stack | Input | Action
------|-------|-------
E $ | int * int $ | pop();push(T X)
T X $ | int * int $ | pop();push(int Y)
int Y X $ | int * int $ | pop();next++
Y X $ | * int $ | pop();push(* T)
* T X $ | * int $ | pop();push(Y)
T X $ | int $ | pop();push(int Y)
int Y X $ | int $ | pop();next++
Y X $ | $ | pop()
X $ | $ | pop()
$ | $ | ACCEPT!

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>$</th>
<th></th>
</tr>
</thead>
<tbody>
<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>\epsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td>\epsilon</td>
<td></td>
<td>\epsilon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constructing Parsing Tables

- No table entry can be multiply defined
- If $A \rightarrow \alpha$, where in the line of $A$ we place $\alpha$?
- In column $t$ where $t$ can start a string derived from $\alpha$
  - $\alpha \rightarrow^* \beta$
  - We say that $t \in \text{First}(\alpha)$
- In the column $t$ if $\alpha$ is $\epsilon$ and $t$ can follow an $A$
  - $S \rightarrow^* \beta A t \delta$
  - We say $t \in \text{Follow}(A)$

First Sets. Example

Recall the grammar

\[
\begin{align*}
E &\rightarrow T X \\
T &\rightarrow ( E ) | \text{int} Y \\
X &\rightarrow + E | \epsilon \\
Y &\rightarrow * T | \epsilon
\end{align*}
\]

First sets

\[
\begin{align*}
\text{First}( ( ) ) &\rightarrow \{ ( ) \} \\
\text{First}( ) &\rightarrow \{ \} \\
\text{First}( \text{int} ) &\rightarrow \{ \text{int} \} \\
\text{First}( + ) &\rightarrow \{ + \} \\
\text{First}( * ) &\rightarrow \{ * \}
\end{align*}
\]

Computing First Sets

Definition: $\text{First}(X) = \{ t | X \rightarrow^* \alpha \} \cup \{ \epsilon | X \rightarrow^* \epsilon \}$

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

Computing Follow Sets

- Definition:
  \[
  \text{Follow}(X) = \{ t | S \rightarrow^* \beta X t \delta \}
  \]

- 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 $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \rightarrow^* \epsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
Computing Follow Sets

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

• Follow sets
  Follow( + ) = { int, ( }  
  Follow( * ) = { int, ( }  
  Follow( ( ) = { int, ( }  
  Follow( E ) = { }, $}  
  Follow( X ) = { $, ) }  
  Follow( T ) = { +, ) , $}  
  Follow( ) = { +, ) , $}  
  Follow( int) = { *, +, ) , $}  

Constructing LL(1) Parsing Tables

• Construct a parsing table T for CFG G  
• For each production A → α in G do:  
  – For each terminal t ∈ First(α) do 
    • T[A, t] = α  
  – If ε ∈ First(α), for each t ∈ Follow(A) do 
    • T[A, t] = α  
  – If ε ∈ First(α) and $ ∈ Follow(A) do 
    • T[A, $] = α

Notes on LL(1) Parsing Tables

• If any entry is multiply defined then G is not LL(1)  
• Reasons why a grammar is not LL(1) include  
  – G is ambiguous  
  – G is left recursive  
  – G is not left-factored  
• Most programming language grammars are not strictly LL(1)  
• There are tools that build LL(1) tables
Bottom-up Parsing

- YACC uses bottom-up parsing. There are two important operations that bottom-up parsers use: **shift** and **reduce**
  - In abstract terms, we do a simulation of a **Push Down Automata** as a finite state automata
- Input: given string to be parsed and the set of productions.
- Goal: Trace a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol

**Algorithm**

1. Start with an empty stack and a full input buffer. (The string to be parsed is in the input buffer.)
2. Repeat until the input buffer is empty and the stack contains the start symbol:
   a. **Shift** zero or more input symbols onto the stack from input buffer until a handle (beta) is found on top of the stack. If no handle is found report syntax error and exit.
   b. **Reduce** handle to the nonterminal A. (There is a production A -> beta)
3. **Accept** input string and return some representation of the derivation sequence found (e.g., parse tree)

- The four key operations in bottom-up parsing are **shift**, **reduce**, **accept** and **error**.
- Bottom-up parsing is also referred to as shift-reduce parsing.
- Important thing to note is to know when to shift and when to reduce and to which reduce.

**Example of Bottom-up Parsing**

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT BUFFER</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>num1+num2*num3$</td>
<td>shift</td>
</tr>
<tr>
<td>$num1</td>
<td>+num2*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$F</td>
<td>+num2*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$T</td>
<td>+num2*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$E</td>
<td>+num2*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$E+</td>
<td>num2*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$E+num2</td>
<td>*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$E+F</td>
<td>*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>$E+T</td>
<td>*num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>E+T</td>
<td>num3$</td>
<td>reduc</td>
</tr>
<tr>
<td>E+T+num3</td>
<td>$</td>
<td>reduc</td>
</tr>
<tr>
<td>E+T+F</td>
<td>$</td>
<td>reduc</td>
</tr>
<tr>
<td>E+T</td>
<td>$</td>
<td>reduc</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Example of Bottom-up Parsing

```
E -> E+T | T       | E-T
T -> T*F | F       | T/F
F -> (E)  | id      | -E
num
```