8 Parsing

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)

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
  - E → E + Num
  - E → Num
  - E → Num + E
- 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

Parsing complexity
- 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
    - LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
    - LR(n) : Left to right, Right derivation, look ahead at most n symbols.
- 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
  - LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
  - LR(n) : Left to right, Right derivation, look ahead at most n symbols.
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy
  - LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
  - LR(n) : Left to right, Right derivation, look ahead at most n symbols.
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy
  - LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
  - LR(n) : Left to right, Right derivation, look ahead at most n symbols.
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy
  - LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
  - LR(n) : Left to right, Right derivation, look ahead at most n symbols.
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:

\[\text{<term> } \rightarrow \text{<factor>} \{(*|/)<factor>\}^*\]

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: \(E \rightarrow E + T\)
  – 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!

Direct Left-Recursive Grammars

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

Elimination of Direct Left-Recursion

• Consider left-recursive grammar
  \(S \rightarrow S \alpha\)
  \(S \rightarrow \beta\)
• S generates strings
  \(\beta\)
  \(\beta \alpha\)
  \(\beta \alpha \alpha\) ...
• Rewrite using right-recursion
  \(S \rightarrow \beta S'\)
  \(S' \rightarrow \alpha S' | \epsilon\)
• Concretely
  \(T \rightarrow T + \text{id}\)
  \(T \rightarrow \text{id}\)
• \(T\) generates strings
  \(\text{id}\)
  \(\text{id} + \text{id}\)
  \(\text{id} + \text{id} + \text{id} \ldots\)
• Rewrite using right-recursion
  \(T \rightarrow \text{id}\)
  \(T \rightarrow \text{id} T\)
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 (not covered)

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 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 upcoming terminal symbols to decide which grammar rule to use next
• And 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 LL(k) parser, does a Left-to-right parse, a Leftmost-derivation, and k-symbol lookahead
• Grammars where one can decide which rule to use by examining only the next token are LL(1)
• 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 LL(1) grammar

Predictive Parser

Example: consider the grammar

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S \rightarrow \text{begin } S L \\
S \rightarrow \text{print } E \\
L \rightarrow \text{end} \\
L \rightarrow ; S L \\
E \rightarrow \text{num = num}
\]

An S expression starts either with an IF, BEGIN, or PRINT token, and an L expression start with an END or a SEMICOLON token, and an E expression has only one production.

Remember…

• Given a grammar and a string in the language defined by the grammar …
• There may be more than one way to derive the string leading to the 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 valid
• To simplify the problem and the algorithms, we often focus on one of two simple derivation strategies
  – A leftmost derivation
  – A rightmost derivation
**LL(k) and LR(k) parsers**

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

**Predictive Parsing and Left Factoring**

- Consider the grammar
  - \( E \rightarrow T + E \)
  - \( E \rightarrow T \)
  - \( T \rightarrow \text{int} \)
  - \( T \rightarrow \text{int} * T \)
  - \( T \rightarrow ( E ) \)
- 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-factor 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.

**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**

- Consider the \([E, \text{int}]\) entry
  - “When current non-terminal is \( E \) & next input \text{int}, use production \( E \rightarrow T \) \( X \)”
  - 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 \( \text{Y} \rightarrow \varepsilon \)
- Consider the \([E, \ast]\) entry
  - Blank entries indicate error situations
  - “There is no way to derive a string starting with * from non-terminal \( E \)”

The LL(1) parsing table:

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>int</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>
</tr>
<tr>
<td>( X )</td>
<td>( + E )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>( T )</td>
<td>( \text{int} \ Y )</td>
<td>( ( E ) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Y )</td>
<td>( \ast )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
</tr>
</tbody>
</table>
**LL(1) Parsing Algorithm**

initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y
      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

**Computing First Sets**

Definition: First(X) = {t | X →* α} ∪ {ε | X →* ε}

Algorithm sketch (see book for details):
1. for all terminals t do First(t)  { t }
2. for each production X → α do First(X)  { ε }
3. if X → A₁ ... Aᵣ α and ε ∈ First(Aᵢ), 1 ≤ i ≤ n do add First(α) to First(X)
4. for each X → A₁ ... Aᵣ s.t. ε ∈ First(Aᵢ), 1 ≤ i ≤ n do add ε to 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 $S \in \text{Follow}(S)$
  – If $X \rightarrow A B$ then First(B) ⊆ Follow(A) and Follow(X) ⊆ Follow(B)
  – Also if $B \rightarrow* \epsilon$ then Follow(X) ⊆ Follow(A)

---

**Constructing Parsing Tables**

- No table entry can be multiply defined
- If $A \rightarrow \alpha$, where in the line of $A$ do we place $\alpha$?
- In column $t$ where $t$ can start a string derived from $\alpha$
  - $\alpha \rightarrow* t \beta$
  - We say that $t \in \text{First}(\alpha)$
- In the column $t$ 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

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

First sets

- First( ) = { ( ) }
- First( T ) = { int, ( ) }
- First( E ) = { int, ( ) }
- First( int ) = { int }
- First( X ) = { +, ε }
- First( Y ) = { *, ε }
- First( ) = { * }

**Computing 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>pop() push(T X)</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>pop() push(int Y)</td>
</tr>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>pop() halt</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>pop() push(* T)</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>pop() push(int Y)</td>
</tr>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>pop() push(int Y)</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>pop() push(* T)</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>pop() push(int Y)</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>pop() push(* T)</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>pop()</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT!</td>
</tr>
</tbody>
</table>
### Computing Follow Sets

**Algorithm sketch:**

1. Follow(S) $\leftarrow \{ \$ \}$
2. For each production $A \rightarrow \alpha X \beta$
   * add First($\beta$) - $\{ \varepsilon \}$ to Follow($X$)
3. For each $A \rightarrow \alpha X \beta$ where $\varepsilon \in$ First($\beta$)
   * add Follow($A$) to Follow($X$)
   * repeat step(s) ___ until no Follow set grows

### Follow Sets. Example

- Recall the grammar
  
  $E \rightarrow TX$  
  $X \rightarrow +E | \varepsilon$  
  $T \rightarrow (E) | int Y$  
  $Y \rightarrow *T | \varepsilon$

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

### 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 $t \in$ First($\alpha$) do
    * $T[A, t] = \alpha$
  - If $\varepsilon \in$ First($\alpha$), for each $t \in$ Follow($A$) do
    * $T[A, t] = \alpha$
  - If $\varepsilon \in$ First($\alpha$) and $\$ \in$ Follow($A$) do
    * $T[A, \$] = \alpha$

### 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