4 (c) parsing

\[
2 \ast (3 + 4) + 5
\]
Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL.
- A recogniser simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar.
- A parser construct a derivation or parse tree for a sentence (if possible).
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A recursive descent parser is a way to implement a top-down parser that is particularly simple.
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) of the parse tree and move down towards the leaves. Easy to implement by hand, but work with restricted grammars. examples:
  - Predictive parsers (e.g., LL(k))

• **Bottom-up parsers:** build the nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handle a larger class of grammars. examples:
  – shift-reduce parser (or LR(k) parsers)

• Both are general techniques that can be made to work for all languages (but not all grammars!).

\[ A = 1 + 3 \times 4 / 5 \]
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

\[
\begin{align*}
E & \rightarrow E + \text{Num} \\
E & \rightarrow \text{Num} \\
E & \rightarrow \text{Num + E} \\
E & \rightarrow \text{Num}
\end{align*}
\]

• The first one, with its 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?
• Parsing an arbitrary Context Free Grammar is \( O(n^3) \), e.g., it can take time proportional the cube of the number of symbols in the input. This is bad! (why?)
• If we constrain the grammar somewhat, we can always parse in linear time. This is good!
• 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

• 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

• Grammar rules which are left-recursive lead to non-termination!
Recursive Decent Parsing: Example

For the grammar:

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

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!
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 Left Recursion

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

- \( S \) generates strings
  
  \( \beta \)
  
  \( \beta \alpha \)
  
  \( \beta \alpha \alpha \)
  
  \( \ldots \)

- Rewrite using right-recursion
  
  \[ S \rightarrow \beta S' \]
  
  \[ S' \rightarrow \alpha S' \mid \epsilon \]

- Concretely
  
  \[ T \rightarrow T + id \]
  
  \[ T \rightarrow id \]

- \( T \) generates strings
  
  \( id \)
  
  \( id+id \)
  
  \( id+id+id \)
  
  \( \ldots \)

- Rewrite using right-recursion
  
  \[ T \rightarrow id T' \]
  
  \[ T' \rightarrow id T' \]
  
  \[ T' \rightarrow \epsilon \]
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
  \[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]
  \[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon \]
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
  – … but that can be done automatically
• Unpopular because of backtracking
  – Thought to be too inefficient
• In practice, backtracking is eliminated by restricting the grammar, allowing us to successfully predict which rule to use
Predictive Parser

• A **predictive parser** uses information from the *first terminal symbol* of each expression to decide which production to use.

• A predictive parser is also known as an **LL(\(k\))** parser because it does a *Left-to-right parse*, a *Leftmost-derivation*, and *k-symbol lookahead*.

• A grammar in which it is possible to decide which production to use examining only the first token (as in the previous example) are called **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

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

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 just 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
  – A leftmost derivation
  – A rightmost derivation
**LL(k) and LR(k) parsers**

- Two important classes of parsers are called LL(k) parsers and LR(k) parsers.
- The name LL(k) means:
  - L - *Left-to-right* scanning of the input
  - L - Constructing *leftmost derivation*
  - k – max number of input symbols needed to select parser action
- The name LR(k) means:
  - L - *Left-to-right* scanning of the input
  - R - Constructing *rightmost derivation* in reverse
  - k – max number 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 applies next
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

• A grammar must be left-factored before use for predictive parsing

• Left-factoring involves rewriting the rules so that, if a non-terminal has more than one rule, each begins with a terminal
Left-Factoring Example

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

```
E → T + E
E → T
T → int
T → int * T
T → ( E )

E → T X
X → + E
X → ε
T → ( E )
T → int Y
Y → * T
Y → ε
```
Left Factoring

• Consider a rule of the form
  \[ A \rightarrow a \ B1 \mid a \ B2 \mid a \ B3 \mid \ldots \ a \ Bn \]
• 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:
  \[ A \rightarrow a \ A1 \]
  \[ A1 \rightarrow B1 \mid B2 \mid B3 \ldots \ Bn \]
Using Parsing Tables

• 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
• Method similar to recursive descent, except
  – For each non-terminal S
  – We look at the next token $a$
  – And chose 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 Table Example

#### Left-factored grammar

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

#### 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>T X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+ E</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td>\varepsilon</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>
**LL(1) Parsing Table Example**

- Consider the [E, int] entry
  - “When current non-terminal is E and next input is int, use production E → T X”
  - Only production that can generate an 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 → ε
- 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>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>ε</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td>ε</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

Production rules:

- E → T X
- X → + E | ε
- T → ( E ) | int Y
- Y → * T | ε
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 == < >

where:
(1) next points to the next input token
(2) X matches some non-terminal
(3) t matches some terminal
## 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>pop();push(T X)</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>pop();push(int Y)</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>pop();next++</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();next++</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>pop();push(int Y)</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>pop();next++;</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>pop()</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>pop()</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT!</td>
</tr>
</tbody>
</table>

### Grammar

<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>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+E</td>
<td></td>
<td>ε</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>*T</td>
<td></td>
<td>ε</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

### Production Rules

- E → TX
- X → +E
- X → ε
- T → (E)
- T → int Y
- Y → *T
- Y → ε
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 line of \( A \) we place \( \alpha \) ?
• 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 \( \epsilon \) 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 First set can be grown
First Sets. Example

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

- First sets

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

• Definition:
  \[ \text{Follow}(X) = \{ t \mid S \xrightarrow{*} \beta X t \delta \} \]

• 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

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

• Follow sets

\[
\begin{align*}
\text{Follow}(+)& = \{ \text{int}, ( ) \} \\
\text{Follow}(\ast)& = \{ \text{int}, ( ) \} \\
\text{Follow}(())& = \{ \text{int}, ( ) \} \\
\text{Follow}(E)& = \{ (), $ \} \\
\text{Follow}(X)& = \{ $(, ) \} \\
\text{Follow}(T)& = \{ (+, ) , $ \} \\
\text{Follow}( )& = \{ (+, ) , $ \} \\
\text{Follow}(\text{int})& = \{ (*, +, ) , $ \}
\end{align*}
\]
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
  – If G is left recursive
  – If G is not left-factored
• Most programming language grammars are not 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. They are namely 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 $A \rightarrow \beta$)

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$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+$</td>
<td>num2*num3$</td>
<td>shift</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$</td>
<td>shift</td>
</tr>
<tr>
<td>E+T*</td>
<td>num3$</td>
<td>shift</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>

### Grammar Rules:

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