4 (c) parsing

**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)

**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.

**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 succeed 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:

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

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!

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
  \( \beta \alpha \beta \alpha \alpha \ldots \)
- Rewrite using right-recursion
  \( S \rightarrow \beta S' \)
  \( S' \rightarrow \alpha S' \mid \epsilon \)
- Concretely
  \( T \rightarrow T + \text{id} \)
  \( T \rightarrow \text{id} \)
- \( T \) generates strings
  \( \text{id+id} \)
  \( \text{id+id+id} \ldots \)
- 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 \textbf{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

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

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 → T + E
  - E → T
  - T → int
  - T → int * T
  - T → ( E )
- Hard to predict because
  - For T, two productions start with 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.

**Left-Factoring Example**

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

\[
\begin{align*}
E & \rightarrow T \cdot E \\
E & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow \text{int} \cdot T \\
T & \rightarrow ( E )
\end{align*}
\]

For each non-terminal the revised grammar, there is either only one rule or every rule begins with a terminal or $E$.

**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
  - 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**

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

The LL(1) parsing table

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+ E</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td>(E)</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

**LL(1) Parsing Table Example**

- Consider the $[E, \text{int}]$ entry
  - “When current non-terminal is E & next input int, use production $E \rightarrow T \cdot X$”
  - It’s the 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 $Y \rightarrow e$
- Consider the $[E, \ast]$ entry
  - Blank entries indicate error situations
  - “There is no way to derive a string starting with * from non-terminal E”

\[
\begin{align*}
\text{int} & \rightarrow \ast \rightarrow \ast E \rightarrow \ast e \rightarrow \ast e \rightarrow \ast e \\
E & \rightarrow TX \rightarrow TX \rightarrow ( ) \rightarrow S \\
X & \rightarrow + E \rightarrow e \rightarrow e \rightarrow e \\
T & \rightarrow \text{int Y} \rightarrow \ast T \rightarrow \ast e \rightarrow \ast e \rightarrow \ast e \\
Y & \rightarrow \ast T \rightarrow \ast e \rightarrow \ast e \rightarrow \ast e |
\end{align*}
\]
**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. (1) next points to the next input token
2. (2) X matches some non-terminal
3. (3) t matches some terminal

**Constructing Parsing Tables**

- No table entry can be multiply defined
- If A → α, where in the line of A do we place α ?
- In column t where t can start a string derived from α
  * α → * t β
  * We say that t ∈ First(α)
- In the column t if α is ε and t can follow an A
  * S → * β A t δ
  * We say t ∈ Follow(A)

**First Sets. Example**

Recall the grammar

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T X</td>
</tr>
<tr>
<td>T</td>
<td>( E )</td>
</tr>
</tbody>
</table>

First sets

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First( X ) = { t 1 }</td>
<td>First( T ) = { int, ( }</td>
<td></td>
</tr>
<tr>
<td>First( ) = { ( }</td>
<td>First( E ) = { int, ( }</td>
<td></td>
</tr>
<tr>
<td>First( int ) = { int }</td>
<td>First( X ) = { +, ε }</td>
<td></td>
</tr>
<tr>
<td>First( + ) = { + }</td>
<td>First( Y ) = { +, ε }</td>
<td></td>
</tr>
<tr>
<td>First( * ) = { * }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Computing First Sets**

Definition: First(X) = { t | X → *α0 } ∪ { ε | X → * ε }

Algorithm sketch (see book for details):

1. for all terminals t do First(t) ≔ { t }
2. for each production X → ε do First(X) ≔ { ε }
3. if X → A1 ... An α and ε ∈ First(Ai), 1 ≤ i ≤ n do add First(α) to First(X)
4. for each X → A1 ... An 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:

  Follow(X) = { t | S → * β X t δ }

- Intuition

  - If S is the start symbol then $S$ ∈ Follow(S)
  - If X → A B then First(B) ⊆ Follow(A) and Follow(X) ⊆ Follow(B)
  - Also if B → ε then Follow(X) ⊆ Follow(A)
Computing Follow Sets

Algorithm sketch:

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

Follow Sets. Example

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

- Follow sets
  
  \[
  \begin{align*}
  \text{Follow}(+) &= \{ \text{int}, ( ) \} \\
  \text{Follow}(\ast) &= \{ \text{int}, ( ) \} \\
  \text{Follow}(E) &= \{ [], \$ \} \\
  \text{Follow}(X) &= \{ [], \$ \} \\
  \text{Follow}(T) &= \{ +, \$ \} \\
  \text{Follow}(Y) &= \{ +, \$ \} \\
  \text{Follow}(\text{int}) &= \{ +, \$ \}
  \end{align*}
  \]

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 \( \epsilon \in \text{First}(\alpha) \), for each \( t \in \text{Follow}(A) \) do
    - \( T[A, t] = \alpha \)
  - If \( \epsilon \in \text{First}(\alpha) \) 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)
- 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