Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:
  \[ t_1, t_2, t_3, t_4, t_5 \]

- The parse tree is constructed:
  - From the top
  - From left to right

Recursive Descent Parsing

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

- Token stream is: \text{int} \ast \text{int}

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

- Try the rules for \( E \) in order

Recursive Descent Parsing. Example (Cont.)

- Try \( E \rightarrow T + E \)

- Then try a rule for \( T \rightarrow (E) \)
  - But ( does not match input token int

- Try \( T \rightarrow \text{int} \ast T \)
  - This will match but + after T will be unmatched

- Have exhausted the choices for \( T \)
  - Backtrack to choice for \( E \)

Recursive Descent Parsing. Example (Cont.)

- Try \( E \rightarrow T \)

- Follow same steps as before for \( T \)
  - And succeed with \( T \rightarrow \text{int} \ast T \) and \( T \rightarrow \text{int} \)
  - With the following parse tree

Recursive Descent Parsing. Notes.

- Easy to implement by hand
  - An example implementation is provided as a supplement “Recursive Descent Parsing”
Recursive-Descent Parsing

- Parsing: given a string of tokens $t_1 t_2 ... t_n$, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1 t_2 ... t_k A ...
  - Try all the productions for $A$: if $A \rightarrow BC$ is a production, the new fringe is $t_1 t_2 ... t_k B C ...
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- But does not always work...
  - Consider a production $S \rightarrow S \alpha$:
    - In the process of parsing $S$ we try the above rule
    - What goes wrong?
  - A left-recursive grammar has a non-terminal $S$ $S \rightarrow^* S \alpha$ for some $\alpha$
  - Recursive descent does not work in such cases
    - It goes into a dead loop

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
  $$S \rightarrow \beta S'$$
  $$S' \rightarrow \alpha S' \mid \varepsilon$$

Elimination of Left-Recursion. Example

- Consider the grammar
  $$S \rightarrow 1 \mid S \ 0 \quad (\beta = 1 \text{ and } \alpha = 0)$$
  can be rewritten as
  $$S \rightarrow 1 S'$$
  $$S' \rightarrow 0 S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general
  $$S \rightarrow S \alpha_1 \mid ... \mid S \alpha_n \mid \beta_1 \mid ... \mid \beta_m$$
  - All strings derived from $S$ start with one of $\beta_1,...,\beta_m$ and continue with several instances of $\alpha_1,...,\alpha_n$
- Rewrite as
  $$S \rightarrow \beta_1 S' \mid ... \mid \beta_m S'$$
  $$S' \rightarrow \alpha_1 S' \mid ... \mid \alpha_n S' \mid \varepsilon$$

General Left Recursion

- The grammar
  $$S \rightarrow A \alpha \mid \delta$$
  $$A \rightarrow S \beta$$
  is also left-recursive because
  $$S \rightarrow^* S \beta \alpha$$
- This left-recursion can also be eliminated
- See book, Section 4.3 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 that could lead to success
• Can be specified as a 2D table
  – 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
  \[
  \begin{align*}
  E & \rightarrow T + E | T \\
  T & \rightarrow \text{int} | \text{int} \ast T | (E)
  \end{align*}
  \]
• Impossible 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 Rule

• If a production
  \[
  A \rightarrow \alpha \beta_1 \beta_2 ... \beta_n | \gamma
  \]
Then
\[
  A \rightarrow \alpha A' | \gamma
  
  A' \rightarrow \beta_1 \beta_2 ... \beta_n
  \]

Left-Factoring Example

• Recall the grammar
  \[
  \begin{align*}
  E & \rightarrow T + E | T \\
  T & \rightarrow \text{int} | \text{int} \ast T | (E)
  \end{align*}
  \]
• Factor out common prefixes of productions
  \[
  \begin{align*}
  E & \rightarrow TX \\
  X & \rightarrow +E | \epsilon \\
  T & \rightarrow (E) | \text{int} Y \\
  Y & \rightarrow \ast T | \epsilon
  \end{align*}
  \]
### LL(1) Parsing Table Example

- **Left-factored grammar**
  
  \[
  \begin{align*}
  E &\rightarrow TX \\
  T &\rightarrow (E) \mid \text{int} Y \\
  X &\rightarrow + E \mid \varepsilon \\
  Y &\rightarrow * T \mid \varepsilon
  \end{align*}
  \]

- **The LL(1) parsing table:**

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

### LL(1) Parsing Table Example (Cont.)

- 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 can generate an \(\text{int}\) in the first place

- Consider the \([Y, +]\) entry
  
  - "When current non-terminal is \(Y\) and current token is \(+\), get rid of \(Y\"
  
  - We'll see later why this is so

### LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  
  - Consider the \([E, \ast]\) entry
  
  - "There is no way to derive a string starting with \(\ast\) from non-terminal \(E\"

### Using Parsing Tables

- Method similar to recursive descent, except
  
  - For each non-terminal \(S\)
  
  - We look at the next token \(a\)
  
  - And choose 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 Algorithm

Initialize stack = \(<S \>$ and next (pointer to tokens)

Repeat

Case stack of

- \(<X, \text{rest}>\) if \(T[X, \ast \text{next}] = Y_1...Y_n\)
  
  Then stack \(\leftarrow <Y_1...Y_n, \text{rest}>\); else error ()

- \(<t, \text{rest}>\) if \(t = \ast \text{next} ++\)
  
  Then stack \(\leftarrow <\text{rest}>\); 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>$</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</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

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

  ![Tree Diagram]

  \[\text{int} \ast \text{int} + \text{int}\]

- The leaves at any point form a string \( \beta \bar{A} \gamma \)
  - \( \beta \) contains only terminals
  - The input string is \( \beta \bar{A} \gamma \)
  - The prefix \( \beta \) matches
  - The next token is \( \beta \)

Predictive Parsing. Review

- A predictive parser is described by a table
  - For each non-terminal \( A \) and for each token \( b \) we specify a production \( A \rightarrow \alpha \)
  - When trying to expand \( A \) we use \( A \rightarrow \alpha \) if \( b \) is the next token and \( A \) the Non-terminal to be expanded.

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary
Constructing Predictive Parsing Tables

• Consider the state $S \rightarrow \gamma \beta A\gamma$
  – With $\beta$ the next token
  – Trying to match $\beta b\delta$

There are two possibilities:

1. $b$ belongs to an expansion of $A$
   • Any $A \rightarrow \alpha$ can be used if $b$ can start a string derived from $\alpha$
     In this case we say that $b \in \text{First}(\alpha)$

Or...

Computing First Sets

Definition $\text{First}(X) = \{ b \mid X \rightarrow^* b \alpha \} \cup \{ e \mid X \rightarrow^* e \}$

1. $\text{First}(b) = \{ b \}$

2. For all productions $X \rightarrow A_1 \ldots A_n$
   • Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{First}(X)$, stop if $e \notin \text{First}(A_i)$
   • Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{First}(X)$, stop if $e \notin \text{First}(A_i)$
   • ... Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{First}(X)$, stop if $e \notin \text{First}(A_i)$
   • Add $\epsilon$ to $\text{First}(X)$

Constructing Predictive Parsing Tables (Cont.)

2. $b$ does not belong to an expansion of $A$
   – The expansion of $A$ is empty and $b$ belongs to an expansion of $\gamma$
   – Means that $b$ can appear after $A$ in a derivation of the form $S \rightarrow \gamma \beta A\gamma$
   – We say that $b \in \text{Follow}(A)$ in this case

• What productions can we use in this case?
  • Any $A \rightarrow \alpha$ can be used if $\alpha$ can expand to $e$
  • We say that $\epsilon \in \text{First}(A)$ in this case

First Sets. Example

• Recall the grammar

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

• First sets

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

Computing Follow Sets

Definition $\text{Follow}(X) = \{ b \mid S \rightarrow \gamma \beta X \beta \delta \}$

1. Compute the First sets for all non-terminals first
2. Add $\epsilon$ to $\text{Follow}(S)$ if $S$ is the start non-terminal
3. For all productions $Y \rightarrow \ldots X A_1 \ldots A_n$
   • Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{Follow}(X)$, stop if $e \notin \text{First}(A_i)$
   • Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{Follow}(X)$, stop if $e \notin \text{First}(A_i)$
   • ... Add $\text{First}(A_i) - \{ \epsilon \}$ to $\text{Follow}(X)$, stop if $e \notin \text{First}(A_i)$
   • Add $\text{Follow}(Y)$ to $\text{Follow}(X)$

Follow Sets. Example

• Recall the grammar

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

• Follow sets

  \[
  \begin{align*}
  \text{Follow}(\ ) &= \{ \text{int }, \} \\
  \text{Follow}(\ ) &= \{ \text{int }, \} \\
  \text{Follow}(T) &= \{ \epsilon, \} \\
  \text{Follow}(\text{int}) &= \{ \epsilon, \} \\
  \text{Follow}(\text{int}) &= \{ \epsilon, \} \\
  \text{Follow}(X) &= \{ \text{int }, \} \\
  \text{Follow}(Y) &= \{ \text{int }, \}
  \end{align*}
  \]
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 \( b \in \text{First}(\alpha) \) do
    - \( T[A, b] = \alpha \)
  - If \( \alpha \rightarrow \epsilon \) and \( b \in \text{Follow}(A) \) do
    - \( T[A, b] = \alpha \)

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
  - And in other cases as well
- 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: a more powerful parsing strategy

Exercise

- Consider the following grammar:
  - \( A \rightarrow BCD \)
  - \( B \rightarrow h B \mid \epsilon \)
  - \( C \rightarrow C g \mid g \mid C h \mid i \)
  - \( D \rightarrow AB \mid \epsilon \)
- 1. compute first and follow sets
- 2. give LL(1) parsing table
- 3. show the action of the LL(1) parser on input string "hhighh"