Compiler Design

Top-down Parser

Hwansoo Han
Top-down parsers

- **LL** = *Left-to-right input scan, Leftmost derivation*
- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers

- **LR** = *Left-to-right input scan, Rightmost derivation*
- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars
Top-down Parser

- **Problems in Top-down parser**
  - Backtrack → predictive parser
  - Left-recursion → may result in infinite loop

- **Predictive parser**
  - LL(1) property
  - Left factoring transforms some non-LL(1) to LL(1)
**Remember the expression grammar?**

- **Version with precedence derived last lecture**

|   |  
|---|---|
| 1 | \( \text{Goal} \rightarrow \text{Expr} \) |
| 2 | \( \text{Expr} \rightarrow \text{Expr} + \text{Term} \) |
| 3 | \( \quad | \text{Expr} - \text{Term} \) |
| 4 | \( \quad | \text{Term} \) |
| 5 | \( \text{Term} \rightarrow \text{Term} \ast \text{Factor} \) |
| 6 | \( \quad | \text{Term} \div \text{Factor} \) |
| 7 | \( \quad | \text{Factor} \) |
| 8 | \( \text{Factor} \rightarrow \text{number} \) |
| 9 | \( \quad | \text{id} \) |

And the input \( x - 2 \ast y \)
Let's try $x \mp 2 * y$:

- This worked well, except that "−" doesn't match "＋"
- The parser must backtrack to here
Continuing with $x - 2 * y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>4</td>
<td>Term - Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>7</td>
<td>Factor - Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$\downarrow x - 2 * y$</td>
</tr>
<tr>
<td>—</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$\downarrow x - 2 * y$</td>
</tr>
</tbody>
</table>

This time, “–” and “–” matched

⇒ Now, we need to expand Term - the last NT on the fringe
Trying to match the “2” in \(x - 2 \times y\):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>&lt;id, x&gt; - Factor</td>
<td>(x - \uparrow _2 \times \downarrow y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id, x&gt; - &lt;num, 2&gt;</td>
<td>(x - \downarrow _2 \times \uparrow y)</td>
</tr>
<tr>
<td></td>
<td>&lt;id, x&gt; - &lt;num, 2&gt;</td>
<td>(x - _2 \times _y)</td>
</tr>
</tbody>
</table>

Where are we?
- “2” matches “2”
- We have more input, but no \(N\_S\) left to expand
- The expansion terminated too soon

⇒ Need to backtrack
## Top-down Parser - backtrack (4)

- **Trying again with “2” in** $x - 2 \ast y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$&lt;id,x&gt; \rightarrow \text{Term}$</td>
<td>$x - \uparrow 2 \ast y$</td>
</tr>
<tr>
<td>5</td>
<td>$&lt;id,x&gt; \rightarrow \text{Term} \ast \text{Factor}$</td>
<td>$x - \uparrow 2 \ast y$</td>
</tr>
<tr>
<td>7</td>
<td>$&lt;id,x&gt; \rightarrow \text{Factor} \ast \text{Factor}$</td>
<td>$x - \uparrow 2 \ast y$</td>
</tr>
<tr>
<td>8</td>
<td>$&lt;id,x&gt; \rightarrow &lt;\text{num},2&gt; \ast \text{Factor}$</td>
<td>$x - \uparrow 2 \ast y$</td>
</tr>
<tr>
<td>—</td>
<td>$&lt;id,x&gt; \rightarrow &lt;\text{num},2&gt; \ast \text{Factor}$</td>
<td>$x - 2 \uparrow \ast y$</td>
</tr>
<tr>
<td>—</td>
<td>$&lt;id,x&gt; \rightarrow &lt;\text{num},2&gt; \ast &lt;id,y&gt;$</td>
<td>$x - 2 \uparrow \ast y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt; \rightarrow &lt;\text{num},2&gt; \ast &lt;id,y&gt;$</td>
<td>$x - \uparrow 2 \ast y$</td>
</tr>
</tbody>
</table>

- This time, we matched & consumed all the input

$\Rightarrow$ **Success!**
Left Recursion

- Top-down parsers cannot handle left-recursive grammars
  - Formally,
    A grammar is left recursive if \( \exists A \in NT \) such that
    \( \exists \) a derivation \( A \Rightarrow^+ A\alpha \), for some string \( \alpha \in (NT \cup T)^+ \)

- Our expression grammar is left recursive
  - This can lead to non-termination in a top-down parser
  - We would like to convert the left recursion to right recursion
Eliminating Left Recursion

❖ Remove left recursion

- Original grammar

\[ F \rightarrow F \alpha \]
\[ | \beta \]
where neither \( \alpha \) nor \( \beta \) starts with \( F \)

- Rewrite the above as

\[ F \rightarrow \beta P \]
\[ P \rightarrow \alpha P \]
\[ | \varepsilon \]
where \( P \) is a new non-terminal

- Accepts the same language, but uses only right recursion
Eliminating Left Recursion

❖ The expression grammar contains two cases of left recursion

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
& \quad \mid \text{Expr} - \text{Term} \\
& \quad \mid \text{Term}
\end{align*}
\]

\[
\begin{align*}
\text{Term} & \rightarrow \text{Term} \times \text{Factor} \\
& \quad \mid \text{Term} \div \text{Factor} \\
& \quad \mid \text{Factor}
\end{align*}
\]

❖ Applying the transformation yields

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} \text{Expr}' \\
\text{Expr}' & \rightarrow + \text{Term} \text{Expr}' \\
& \quad \mid - \text{Term} \text{Expr}' \\
& \quad \mid \varepsilon
\end{align*}
\]

\[
\begin{align*}
\text{Term} & \rightarrow \text{Factor} \text{Term}' \\
\text{Term}' & \rightarrow \times \text{Factor} \text{Term}' \\
& \quad \mid \div \text{Factor} \text{Term}' \\
& \quad \mid \varepsilon
\end{align*}
\]

- These fragments use only right recursion
- They retain the original left associativity (evaluate left to right)
Predictive Parsing

❖ **Basic idea**

*Given* $A \rightarrow \alpha | \beta$, *the parser should be able to choose between* $\alpha$ & $\beta$

❖ **FIRST sets**

- For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from $\alpha$
- That is, $x \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* x \gamma$, for some $\gamma$

❖ **The LL(1) Property** (first version)

- If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
- This would allow the parser to make a correct choice with a lookahead of exactly one symbol!
What about $\varepsilon$-productions?

- They complicate the definition of LL(1)
  - If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(\alpha)$, too.
  - Define $\text{FIRST}^+(\alpha)$ for $A \rightarrow \alpha$ as:
    - $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\alpha)$
    - $\text{FIRST}(\alpha)$, otherwise.
  - Then, a grammar is $LL(1)$ iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies
    $$\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset$$
The FIRST Set

❖ **Definition**
  - \( x \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* x \gamma, \) for some \( \gamma \)

❖ **Building FIRST(X)**
  - If \( X \) is a terminal (token), \( \text{FIRST}(X) = \{X\} \)
  - If \( X \rightarrow \varepsilon \), then \( \varepsilon \in \text{FIRST}(X) \)
  - **Iterate until** no more terminals or \( \varepsilon \) can be added to any \( \text{FIRST}(X) \)
    - if \( X \rightarrow y_1y_2 \ldots y_k \) then
      - \( a \in \text{FIRST}(X) \) if \( a \in \text{FIRST}(y_i) \) and \( \varepsilon \in \text{FIRST}(y_h) \) for all \( 1 \leq h < i \)
      - \( \varepsilon \in \text{FIRST}(X) \) if \( \varepsilon \in \text{FIRST}(y_i) \) for all \( 1 \leq i \leq k \)

❖ **Note**
  - If \( \varepsilon \not\in \text{FIRST}(y_1) \), then \( \text{FIRST}(y_i) \) is irrelevant, for \( i > 1 \)
The FOLLOW Set

❖ **Definition**
  - FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form

❖ **Building FOLLOW(X) for all non-terminal X**
  - EOF ∈ FOLLOW(S)
  - Iterate until no more terminals can be added to any FOLLOW(X)
    - If A → αB, then put FOLLOW(A) in FOLLOW(B)
    - If A → αBβ, then put \{FIRST(β) - ε\} in FOLLOW(B)
    - If A → αBβ and ε ∈ FIRST(β), then put FOLLOW(A) in FOLLOW(B)
  - End iterate

❖ **Note**
  - FOLLOW is for non-terminals, no FOLLOW for terminals
  - No ε in FOLLOW(X) for any non-terminal X
Left Factoring

- What if my grammar does not have the LL(1) property?
  ⇒ Sometimes, we can transform the grammar

\[
\forall A \in NT, \\
\text{find the longest prefix } \alpha \text{ that occurs in two or more right-hand sides of } A \\
\text{if } \alpha \neq \varepsilon \text{ then replace all of the } A \text{ productions, } \\
A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma, \\
\text{with} \\
A \rightarrow \alpha Z \mid \gamma \\
Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n
\]

where \( Z \) is a new element of \( NT \)

Repeat until no common prefixes remain
Left Factoring (An example)

- Consider the following fragment of the expression grammar

  \[
  \begin{align*}
  \text{Factor} & \rightarrow \text{Identifier} \\
  & \mid \text{Identifier [ ExprList ]} \\
  & \mid \text{Identifier ( ExprList )}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{FIRST}^+ (\text{rhs}_1) &= \{ \text{Identifier} \} \\
  \text{FIRST}^+ (\text{rhs}_2) &= \{ \text{Identifier} \} \\
  \text{FIRST}^+ (\text{rhs}_3) &= \{ \text{Identifier} \}
  \end{align*}
  \]

- After left factoring, it becomes

  \[
  \begin{align*}
  \text{Factor} & \rightarrow \text{Identifier Arguments} \\
  \text{Arguments} & \rightarrow [ \text{ExprList} ] \\
  & \mid ( \text{ExprList} ) \\
  & \mid \varepsilon
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{FIRST}^+ (\text{rhs}_1) &= \{ [ ] \} \\
  \text{FIRST}^+ (\text{rhs}_2) &= \{ ( ) \} \\
  \text{FIRST}^+ (\text{rhs}_3) &= \{ \varepsilon \} \cup \text{FOLLOW( Factor )}
  \end{align*}
  \]

- This form has the same syntax, with the \textit{LL(1)} property.
**Left Recursion & Left Factoring (Generality)**

**Question**

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the $LL(1)$ condition? (and can be parsed predictively with a single token lookahead?)

**Answer**

Given a CFG that doesn’t meet the $LL(1)$ condition, it is undecidable whether or not an equivalent $LL(1)$ grammar exists.

**Example** that has no $LL(1)$ grammar

$$\{a^n 0 b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\}$$
Language that Cannot Be $LL(1)$

**Example**

$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$ has no $LL(1)$ grammar

$$G \rightarrow aAb$$
$$\quad \mid aBbb$$

$$A \rightarrow aAb$$
$$\quad \mid 0$$

$$B \rightarrow aBbb$$
$$\quad \mid 1$$

**Problem:** need an unbounded number of $a$ characters before you can determine whether you are in the $A$ group or the $B$ group.
Automate Predictive Parsing

❖ **Given a grammar that has the \textit{LL(1)} property**
  - Can write a simple routine to recognize each \textit{lhs}
  - Code is both simple & fast

❖ **Consider** \( A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \), with
  - \( \text{FIRST}^+ (\beta_1) \cap \text{FIRST}^+ (\beta_2) = \emptyset \)
  - \( \text{FIRST}^+ (\beta_2) \cap \text{FIRST}^+ (\beta_3) = \emptyset \)
  - \( \text{FIRST}^+ (\beta_1) \cap \text{FIRST}^+ (\beta_3) = \emptyset \)

```c
/* find an A */
if (current_token \in \text{FIRST}^+ (\beta_1))
    find a \beta_1 and return true
else if (current_token \in \text{FIRST}^+ (\beta_2))
    find a \beta_2 and return true
else if (current_token \in \text{FIRST}^+ (\beta_3))
    find a \beta_3 and return true
else
    report an error and return false
```

Grammars with the \textit{LL(1)} property are called \textit{predictive grammars} because the parser can “predict” the correct expansion at each point in the parse.

Parsers that capitalize on the \textit{LL(1)} property are called \textit{predictive parsers}.

One kind of predictive parser is the \textit{recursive descent} parser.
### Predictive Parsing Example

#### Expression grammar, after transformation

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong></td>
<td>→</td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong></td>
<td>→</td>
</tr>
<tr>
<td>3</td>
<td><strong>Expr’</strong></td>
<td>→</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><strong>Term</strong></td>
<td>→</td>
</tr>
<tr>
<td>7</td>
<td><strong>Term’</strong></td>
<td>→</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><strong>Factor</strong></td>
<td>→</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This produces a parser with six *mutually recursive* routines:

- **Goal**
- **Expr**
- **EPrime**
- **Term**
- **TPrime**
- **Factor**

Each recognizes one NT or T

The term *descent* refers to the direction in which the parse tree is built.
Recursive Descent Parser

A couple of routines from the expression parser

Goal()

token ← next_token();
if (Expr() = true & token = EOF)
then next compilation step;
else
report syntax error;
return false;

Expr()
if (Term() = false)
then return false;
else return Eprime();

Factor()
if (token = Number) then

token ← next_token();
return true;
else if (token = Identifier) then

token ← next_token();
return true;
else
report syntax error;
return false;

EPrime, Term, & TPrime follow the same basic lines

looking for EOF, found other token

looking for Number or Identifier, found other token instead
Recursive Descent Parser (cont’d)

EPrime()
// Expr’ → + Term Expr’
// Expr’ → - Term Expr’
if (token = + or token = -) then
  token ← next_token();
  if (Term()) then
    return EPrime();
  else return false; // Fail
// Expr’ → ε
else if (token = EOF) then
  return true;
else return false; // Fail

Term & TPrime follow the same basic lines
Parse Tree - Recursive Descent Parser

❖ To build a parse tree:
  ▪ Augment parsing routines to build nodes
  ▪ Pass nodes between routines using a stack
  ▪ Node for each symbol on rhs
  ▪ Action is to pop rhs nodes, make them children of lhs node, and push this subtree

❖ To build an abstract syntax tree
  ▪ Build fewer nodes
  ▪ Put them together in a different order

```plaintext
Expr → Term Expr'

Expr()
result ← true;
if (Term() = false) then
  return false;
else if (EPrime() = false) then
  result ← false;
else // successfully parsed!
  build an Expr node
  pop EPrime node
  pop Term node
  make EPrime & Term children of Expr
  push Expr node
  return result;
```

Success ⇒ build a piece of the parse tree

This is a preview of Chapter 4
Building Table-driven Parser

❖ **Strategy**
  - Encode knowledge in a table
  - Use a standard “skeleton” parser to interpret the table

❖ **Example**
  - The non-terminal *Factor* has two expansions
    - *Identifier* or *Number*
  - Need a row for every *NT* & a column for every *T*
  - Table might look like:

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>−</th>
<th>*</th>
<th>/</th>
<th>Id.</th>
<th>Num.</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Factor</strong></td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>10</td>
<td>11</td>
<td>−</td>
</tr>
</tbody>
</table>

Error on ‘+’  
Reduce by rule 10 on ‘x’
LL(1) Skeleton Parser

token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack
      token ← next_token()
    else report error looking for TOS
  else
    if TABLE[TOS,token] is $A \rightarrow B_1 B_2 \ldots B_k$ then
      pop Stack
      push $B_k, B_{k-1}, \ldots, B_1$
    else report error expanding TOS
TOS ← top of Stack
Building LL(1) table

- **Building the complete table for LL(1)**
  - Need a row for every $NT$ & a column for every $T$
  - Need an algorithm to build the table

- **Filling in $\text{TABLE}[X,y]$, $X \in NT, y \in T \cup \{EOF\}$**
  1. entry is the rule $X \rightarrow \beta$, if $y \in \text{FIRST}^+ (\beta)$
  2. entry is error, otherwise

- If any entry is defined multiple times, $G$ is not $LL(1)$
Summary

❖ **Top-down parser**
  ▪ Use leftmost derivation
  ▪ Bad pick of rewrite rule results in *backtrack*

❖ **Left recursion removal**
  ▪ Avoid non-terminating top-down parser

❖ **Predictive parsing**
  ▪ LL(1) property ensures only one production rule is chosen by looking ahead one terminal symbol.

❖ **Left factoring**
  ▪ Transform some non-LL(1) to LL(1)

❖ **Automatic top-down parser generation**
  ▪ Recursive decent parser
  ▪ Building LL(1) table: \( f(X,y) \rightarrow P \) (where \( X \in NT, y \in T \))