Parsing III
(Top-down parsing: recursive descent & LL(1))
We set out to study parsing

- Specifying syntax
  - Context-free grammars
  - Ambiguity

- Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal

- Predictive top-down parsing
  - The LL(1) condition
  - Simple recursive descent parsers
  - Table-driven LL(1) parsers
Picking the “Right” Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?
- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami (CYK) algorithm or Earley’s algorithm

Fortunately,
- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are $LL(1)$ and $LR(1)$ grammars
Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, 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 $\gamma$

We will defer the problem of how to compute FIRST sets until we look at the $LR(1)$ table construction algorithm
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
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!

This is almost correct
See the next slide
Predictive Parsing

What about \( \varepsilon \)-productions?

⇒ They complicate the definition of LL(1)

If \( A \to \alpha \) and \( A \to \beta \) and \( \varepsilon \in \text{FIRST}(\alpha) \), then we need to ensure that \( \text{FIRST}(\beta) \) is disjoint from \( \text{FOLLOW}(\alpha) \), too.

Define \( \text{FIRST}^+(\alpha) \) as

- \( \text{FIRST}(\alpha) \cup \text{FOLLOW}(\alpha) \), if \( \varepsilon \in \text{FIRST}(\alpha) \)
- \( \text{FIRST}(\alpha) \), otherwise

Then, a grammar is LL(1) iff \( A \to \alpha \) and \( A \to \beta \) implies

\[
\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset
\]

\( \text{FOLLOW}(\alpha) \) is the set of all words in the grammar that can legally appear immediately after an \( \alpha \).
**FIRST and FOLLOW Sets**

**FIRST(α)**
For some $\alpha \in T \cup NT$, 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 $\gamma$.

**FOLLOW(α)**
For some $\alpha \in NT$, define $\text{FOLLOW}(\alpha)$ as the set of symbols that can occur immediately after $\alpha$ in a valid sentence.

$\text{FOLLOW}(S) = \{\text{EOF}\}$, where $S$ is the start symbol.

To build $\text{FOLLOW}$ sets, we need $\text{FIRST}$ sets ...
Computing FOLLOW Sets

\[
\text{FOLLOW}(S) \leftarrow \{\text{EOF}\}
\]

for each \( A \in NT \), \( \text{FOLLOW}(A) \leftarrow \emptyset \)

while (FOLLOW sets are still changing)

for each \( p \in P \), of the form \( A \rightarrow \beta_1 \beta_2 \ldots \beta_k \)

\[
\text{FOLLOW}(\beta_k) \leftarrow \text{FOLLOW}(\beta_k) \cup \text{FOLLOW}(A)
\]

\[
\text{TRAILER} \leftarrow \text{FOLLOW}(A)
\]

for \( i \leftarrow k \) down to 2

if \( \varepsilon \in \text{FIRST}(\beta_i) \) then

\[
\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup \{ \text{FIRST}(\beta_i) - \{ \varepsilon \} \} 
\]

\[
\cup \text{TRAILER}
\]

else

\[
\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup \text{FIRST}(\beta_i)
\]

\[
\text{TRAILER} \leftarrow \emptyset
\]
Computing $\text{FIRST}$ Sets

for each $\alpha \in (T \cup \varepsilon)$
\[ \text{FIRST}(\alpha) \leftarrow \alpha \]

for each $A \in \text{NT}$
\[ \text{FIRST}(A) \leftarrow \emptyset \]

while (FIRST sets are still changing)

for each $p \in P$, where $p$ has the form $A \rightarrow \beta$
if $\beta$ is $\beta_1 \beta_2 \ldots \beta_k$, where $\beta_i \in T \cup \text{NT}$, then
\[ \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup (\text{FIRST}(\beta_1) - \{\varepsilon\}) \]

$i \leftarrow 1$
while ($\varepsilon \in \text{FIRST}(\beta_i)$ and $i < k$)
\[ \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup (\text{FIRST}(\beta_{i+1}) - \{\varepsilon\}) \]

$i \leftarrow i+1$
if $i = k$ and $\varepsilon \in \text{FIRST}(\beta_k)$, then
\[ \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{\varepsilon\} \]
Predictive Parsing

Given a grammar that has the $LL(1)$ property

- Can write a simple routine to recognize each $lhs$
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \emptyset$

/* find an $A$ */
if (current_word $\in$ FIRST($\beta_1$))
    find a $\beta_1$ and return true
else if (current_word $\in$ FIRST($\beta_2$))
    find a $\beta_2$ and return true
else if (current_word $\in$ FIRST($\beta_3$))
    find a $\beta_3$ and return true
else
    report an error and return false

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

Parsers that capitalize on the $LL(1)$ property are called **predictive parsers**.

One kind of predictive parser is the **recursive descent** parser.

Of course, there is more detail to “find a $\beta_i$” (§ 3.3.4 in EAC)
Recall the expression grammar, after transformation

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>$\rightarrow$</td>
<td>Expr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>$\rightarrow$</td>
<td>Term Expr'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Expr'</td>
<td>$\rightarrow$</td>
<td>+ Term Expr'</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$</td>
<td>$</td>
<td>- Term Expr'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
<td>$\rightarrow$</td>
<td>Factor Term'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Term'</td>
<td>$\rightarrow$</td>
<td>* Factor Term'</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$</td>
<td>$</td>
<td>/ Factor Term'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>6</td>
<td>Factor</td>
<td>$\rightarrow$</td>
<td>number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$</td>
<td>$</td>
<td>id</td>
<td></td>
</tr>
</tbody>
</table>

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

- Goal
- Expr
- Expr'
- Term
- Term'
- Factor

Each recognizes one NT or T

The term *descent* refers to the direction in which the parse tree is built.
Recursive Descent Parsing  (Procedural)

A couple of routines from the expression parser

Goal( )
\[
\text{token} \leftarrow \text{next\_token( )}; \\
\text{if } (\text{Expr( )} = \text{true} \& \text{token} = \text{EOF}) \\
\text{then next compilation step;} \\
\text{else} \\
\quad \text{report syntax error;} \\
\quad \text{return false;}
\]

Expr( )
\[
\text{if } (\text{Term( )} = \text{false}) \\
\text{then return false;} \\
\text{else return Eprime( );}
\]

Factor( )
\[
\text{if } (\text{token} = \text{Number}) \text{ then} \\
\text{token} \leftarrow \text{next\_token( )}; \\
\text{return true;} \\
\text{else if } (\text{token} = \text{Identifier}) \text{ then} \\
\text{token} \leftarrow \text{next\_token( )}; \\
\text{return true;} \\
\text{else} \\
\quad \text{report syntax error;} \\
\quad \text{return false;}
\]

EPrime, Term, & TPrime follow the same basic lines (Figure 3.7, EAC)

looking for Number or Identifier, found token instead
Recursive Descent Parsing

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

```
Expr()
result ← true;
if (Term() = false)
    then return false;
else if (EPrime() = false)
    then result ← false;
else
    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
Left Factoring

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

The Algorithm

\[ \forall A \in \text{NT}, \]
find the longest prefix \( \alpha \) that occurs in two or more right-hand sides of \( A \)

if \( \alpha \neq \varepsilon \) then replace all of the \( A \) productions,
\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma, \]

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

A graphical explanation for the same idea

\[ A \rightarrow \alpha \beta_1 \]
\[ | \alpha \beta_2 \]
\[ | \alpha \beta_3 \]

becomes ...

\[ A \rightarrow \alpha Z \]
\[ Z \rightarrow \beta_1 \]
\[ | \beta_2 \]
\[ | \beta_n \]
**Left Factoring**  
(An example)

Consider the following fragment of the expression grammar

<table>
<thead>
<tr>
<th></th>
<th>Factor</th>
<th>FIRST(rhs&lt;sub&gt;1&lt;/sub&gt;) = { Identifier }</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identifier</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Identifier [ ExprList ]</td>
<td>FIRST(rhs&lt;sub&gt;2&lt;/sub&gt;) = { Identifier }</td>
</tr>
<tr>
<td>3</td>
<td>Identifier ( ExprList )</td>
<td>FIRST(rhs&lt;sub&gt;3&lt;/sub&gt;) = { Identifier }</td>
</tr>
</tbody>
</table>

After left factoring, it becomes

<table>
<thead>
<tr>
<th></th>
<th>Factor</th>
<th>FIRST(rhs&lt;sub&gt;1&lt;/sub&gt;) = { Identifier }</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identifier Arguments</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Arguments [ ExprList ]</td>
<td>FIRST(rhs&lt;sub&gt;2&lt;/sub&gt;) = { }</td>
</tr>
<tr>
<td>3</td>
<td>( ExprList )</td>
<td>FIRST(rhs&lt;sub&gt;3&lt;/sub&gt;) = { }</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
<td>FOLLOW(Factor)</td>
</tr>
</tbody>
</table>

\[ \Rightarrow \text{It has the } LL(1) \text{ property} \]

This form has the same syntax, with the \textit{LL(1)} property.
Graphically becomes ...

No basis for choice

Word determines correct choice
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

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

\begin{align*}
G & \to aAb \\
& \quad \mid aBbb \\
A & \to aAb \\
& \quad \mid 0 \\
B & \to aBbb \\
& \quad \mid 1 \\
\end{align*}

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.
Recursive Descent (Summary)

1. Build FIRST (and FOLLOW) sets
2. Massage grammar to have $LL(1)$ condition
   a. Remove left recursion
   b. Left factor it
3. Define a procedure for each non-terminal
   a. Implement a case for each right-hand side
   b. Call procedures as needed for non-terminals
4. Add extra code, as needed
   a. Perform context-sensitive checking
   b. Build an IR to record the code

Can we automate this process?
Building Top-down Parsers

Given an $LL(1)$ grammar, and its FIRST & FOLLOW sets ...

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs’s
  - Each returns true on success and throws an error on false
  - Simple, working $(, \textit{perhaps} \textit{ugly},)$ code

- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better

- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning
Building Top-down Parsers

Strategy

• Encode knowledge in a table
• Use a standard “skeleton” parser to interpret the table

Example

• The non-terminal \textit{Factor} has three expansions
  \[ \rightarrow (Expr) \text{ or } \textit{Identifier} \text{ or } \textit{Number} \]
• Table might look like:

\begin{tabular}{c|c|c|c|c|c|c}
  & + & - & * & / & Id. & Num. & EOF \\
\hline
\textit{Factor} & - & - & - & - & 10 & 11 & - \\
\end{tabular}

Terminal Symbols

Non-terminal Symbols

Error on `+`

Reduce by rule 10 on `+`
Building Top Down Parsers

Building the complete table
• Need a row for every $NT$ & a column for every $T$
• Need a table-driven interpreter for the table
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         // recognized TOS
      token ← next_token()
    else report error looking for TOS
  else
    // TOS is a non-terminal
    if TABLE[TOS,token] is A → B₁B₂...Bₖ then
      pop Stack         // get rid of A
      push Bₖ, Bₖ₋₁, ..., B₁ // in that order
    else report error expanding TOS

TOS ← top of Stack
Building Top Down Parsers

Building the complete table

• Need a row for every $NT$ & a column for every $T$
• Need an algorithm to build the table

Filling in $\text{TABLE}[X,y]$, $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$, if $y \in \text{FIRST}(\beta)$
2. entry is the rule $X \rightarrow \varepsilon$ if $y \in \text{FOLLOW}(X)$ and $X \rightarrow \varepsilon \in G$
3. entry is error if neither 1 nor 2 define it

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

This is the $LL(1)$ table construction algorithm
Extra Slides Start Here
Recursive Descent in Object-Oriented Languages

• Shortcomings of Recursive Descent
  → Too procedural
  → No convenient way to build parse tree

• Solution
  → Associate a class with each non-terminal symbol
    ▪ Allocated object contains pointer to the parse tree

Class NonTerminal {

  public:
  NonTerminal(Scanner & scnr) { s = &scnr; tree = NULL; }
  virtual ~NonTerminal() { }
  virtual bool isPresent() = 0;
  TreeNode * abSynTree() { return tree; }

  protected:
  Scanner * s;
  TreeNode * tree;
}
Non-terminal Classes

Class Expr : public NonTerminal {
public:
    Expr(Scanner & scnr) : NonTerminal(scnr) { }
    virtual bool isPresent();
}

Class EPrime : public NonTerminal {
public:
    EPrime(Scanner & scnr, TreeNode * p) :
        NonTerminal(scnr) { exprSofar = p; }
    virtual bool isPresent();
protected:
    TreeNode * exprSofar;
}

... // definitions for Term and TPrime

Class Factor : public NonTerminal {
public:
    Factor(Scanner & scnr) : NonTerminal(scnr) { }
    virtual bool isPresent();
}
bool Expr::isPresent() {
    Term * operand1 = new Term(*s);
    if (!operand1->isPresent()) return FALSE;

    Eprime * operand2 = new EPrime(*s, NULL);
    if (!operand2->isPresent()) // do nothing;

    return TRUE;
}
Implementation of `isPresent`

```cpp
bool EPrime::isPresent() {
    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {
        s->advance();

        Term * operand2 = new Term(*s);
        if (!operand2->isPresent()) throw SyntaxError(*s);

        EPrime * operand3 = new EPrime(*s, NULL);
        if (operand3->isPresent()); //do nothing

        return TRUE;
    }
    else return FALSE;
}
```
bool Expr::isPresent() { // with semantic processing

    Term * operand1 = new Term(*s);
    if (!operand1->isPresent()) return FALSE;
    tree = operand1->abSynTree();

    EPrime * operand2 = new EPrime(*s, tree);
    if (operand2->isPresent())
        tree = operand2->absSynTree();

    // here tree is either the tree for the Term
    // or the tree for Term followed by EPrime
    return TRUE;
}
bool EPrime::isPresent() { // with semantic processing
    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {
        s->advance();

        Term * operand2 = new Term(*s);
        if (!operand2->isPresent()) throw SyntaxError(*s);

        TreeNode * t2 = operand2->absSynTree();
        tree = new TreeNode(op, exprSofar, t2);

        Eprime * operand3 = new Eprime(*s, tree);
        if (operand3->isPresent())
            tree = operand3->absSynTree();
        return TRUE;
    }
    else return FALSE;
}
bool Factor::isPresent() { // with semantic processing
    token_type op = s->nextToken();

    if (op == IDENTIFIER | op == NUMBER) {
        tree = new TreeNode(op, s->tokenValue());
        s->advance();
        return TRUE;
    } else if (op == LPAREN) {
        s->advance();
        Expr * operand = new Expr(*s);
        if (!operand->isPresent()) throw SyntaxError(*s);
        if (s->nextToken() != RPAREN) throw SyntaxError(*s);
        s->advance();
        tree = operand->absSynTree();
        return TRUE;
    } else return FALSE;
}