top-down parsing
part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Second and third homework solutions available on sakai under Resources tab

• Fourth homework due Sunday, March 10, 11:59pm.

• First project new due dates: CODE Wednesday March 6; REPORT Friday March 8; benchmark programs have been posted.
  ~uli/cs415/projects/proj1/benchmarks

• Midterm: Wednesday, March 13, in class;
  closed book, closed notes, 80 minutes
Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.3
LL(1), recursive descent

1 input symbol lookahead
construct leftmost derivation (forwards)
input: read left-to-right

\[ S \Rightarrow_{lm}^* x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow_{lm}^* x y \]

? Means that we don’t know yet this part of the parse tree
**LL(1), recursive descent**

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

input: read left-to-right

**rule \( A \rightarrow \delta \)**

\[
S \Rightarrow^{*_{lm}} \cdot A \beta \Rightarrow_{lm} \cdot \delta \beta \Rightarrow^{*_{lm}} \cdot y
\]

\[ x \]

\[ y \]

? Means that we don’t know yet this part of the parse tree
**LL(1) Parser Example**

Is the following grammar LL(1)?

\[ S \rightarrow a \ S \ b \ | \ \epsilon \]

\[
\text{First}(aSb) = \{ a \} \\
\text{First}(\epsilon) = \{ \epsilon \} \\
\text{Follow} (S) = \{ \text{eof, b} \}
\]

\[
\text{First}^+(aSb) = \{ a \} \\
\text{First}^+(\epsilon) = (\text{First}(\epsilon) - \{ \epsilon \}) \cup \text{Follow}(S) = \{ \text{eof, b} \}
\]

LL(1)?
Is the following grammar LL(1)?

\[ S \rightarrow a \, S \, b \mid \varepsilon \]

First(aSb) = \{ a \}
First(\varepsilon) = \{ \varepsilon \}
Follow (S) = \{ eof, b \}

First'^+(aSb) = \{ a \}
First'^+(\varepsilon) = (First(\varepsilon) - \{ \varepsilon \}) \cup Follow(S) = \{ eof, b \}

LL(1)? YES, since \{ a \} \cap \{ eof, b \} = \emptyset
### LL(1) Parse Table

Table-driven LL(1) parser

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

**current input symbol**

**rules for non-terminal**

**non-terminal on top of the stack**
Building the complete table
• Need a row for every $NT$ & a column for every $T + "eof"$
• Need an algorithm to build the table

Filling in $TABLE[X,y]$, $X \in NT$, $y \in T \cup \{eof\}$
• entry is the rule $X \rightarrow \beta$, if $y \in FIRST+(\beta)$
• entry is error otherwise

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

This is the $LL(1)$ table construction algorithm
**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
    // TOS is a non-terminal
    if TABLE[TOS, token] is $A \rightarrow B_1 B_2 \ldots B_k$ then
      pop Stack
      push $B_k, B_{k-1}, \ldots, B_1$
      // in that order
    else report error expanding TOS
  TOS ← top of Stack
Table-driven LL(1) parser

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>e</td>
<td>e</td>
<td>error</td>
</tr>
</tbody>
</table>

How to parse input a a a b b b ?

Describe action as sequence of states
(PDA stack content, remaining input, next action)

PDA stack content: [ X, ... Z ], where Z is the TOS
next actions: rule or next input+pop or error or accept
**Table-driven LL(1) parser**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

- \([\text{eof}, S], aaabbb, aSb) \Rightarrow\)
- \([\text{eof}, b, S, a], aaabbb, \text{next input+pop}) \Rightarrow\)
- \([\text{eof}, b, S], aabbb, aSb) \Rightarrow\)
- \([\text{eof}, b, b, S, a], aabbb, \text{next input+pop}) \Rightarrow\)
- \([\text{eof}, b, b, S], abbb, aSB) \Rightarrow\)
- \([\text{eof}, b, b, b, S, a], abbb, \text{next input+pop}) \Rightarrow\)
- \([\text{eof}, b, b, b, S], bbb, \varepsilon) \Rightarrow\)
- \([\text{eof}, b, b, b], bbb, \text{next input+pop}) \Rightarrow ( [\text{eof}, b, b], bb, \text{next input+pop}) \Rightarrow\)
- \([\text{eof}, b, b, b], b, \text{next input+pop}) \Rightarrow ( [\text{eof}, \varepsilon, \text{accept})\)
Recursive descent LL(1) parser

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

1. Every NT is associated with a parsing procedure.

2. The parsing procedure for $A \in \text{NT}$, proc $A$, is responsible to parse and consume any (token) string that can be derived from $A$; it may recursively call other parsing procedures.

3. The parser is invoked by calling proc $S$ for start symbol $S$. 
recursive descent LL(1) parser

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td><strong>aSb</strong></td>
<td>ε</td>
<td>ε</td>
<td><strong>error</strong></td>
</tr>
</tbody>
</table>

main () {
    token = next_token();
    if (S () and token = eof )
        print “accept”
    else
        print “error”;
}

bool S () {  
    switch token {
    case a: token = next_token();
        S();
        if token = b
            {token = next_token(); return true;}  
        else
            return false;
    break;
    case b, eof: return true; break;
    default: return false;
    }
}
**Recursive descent LL(1) parser**

### Syntax Table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

### Code

```c
bool S() {
    switch token {
    case a: token = next_token();
        S();
        if token = b
            {token = next_token(); return true;}
        else
            return false;
    break;
    case b, eof:
        return true;
    break;
    default:
        return false;
    }
}
```

### Main Function

```c
main() {
    token = next_token();
    if (S() and token = eof)
        print "accept"
    else
        print "error";
}
```

---

**How to parse input a a a b b b?**
• Build FIRST (and FOLLOW) sets
• Massage grammar to have $LL(1)$ condition
  • Remove left recursion
  • Left factor it
• Define a procedure for each non-terminal
  • Implement a case for each right-hand side
  • Call procedures as needed for non-terminals
• Add extra code, as needed
  • Perform context-sensitive checking
  • Build an IR (e.g., simple code generation)
  • ...

Can we automate this process?
Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if $\exists \; A \in NT$ such that

$\exists \; \text{a derivation } A \Rightarrow^+ A\alpha$, 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

• For a top-down parser, any recursion must be right recursion

• We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler
To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[ Fee \rightarrow Fee \alpha \]
\[ \quad \mid \beta \]

where neither \( \alpha \) nor \( \beta \) start with \( Fee \)

We can rewrite this as

\[ Fee \rightarrow \beta \text{Fie} \]
\[ \text{Fie} \rightarrow \alpha \text{Fie} \]
\[ \quad \mid \varepsilon \]

where \( \text{Fie} \) is a new non-terminal

This accepts the same language, but uses only right recursion
Eliminating Left Recursion

The expression grammar contains two cases of left recursion

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} + \text{Term} & \text{Term} & \rightarrow \text{Term} \ast \text{Factor} \\
& | \quad \text{Expr} - \text{Term} & & | \quad \text{Term} / \text{Factor} \\
& | \quad \text{Term} & & | \quad \text{Factor}
\end{align*}
\]

Applying the transformation yields

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

\[
\begin{align*}
\text{Term} & \rightarrow \text{Factor} \text{Term'} \\
\text{Term'} & | \quad \ast \text{Factor} \text{Term'} \\
& | \quad / \text{Factor} \text{Term'} \\
& | \quad \varepsilon
\end{align*}
\]

These fragments use only right recursion
Eliminating Left Recursion

Substituting them back into the grammar yields

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong></td>
<td>→  <strong>Expr</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong></td>
<td>→  <strong>Term Expr’</strong></td>
</tr>
<tr>
<td>3</td>
<td><strong>Expr’</strong></td>
<td>→  + <strong>Term Expr’</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Term</strong></td>
<td>→  <strong>Factor Term’</strong></td>
</tr>
<tr>
<td>5</td>
<td><strong>Term’</strong></td>
<td>→  * <strong>Factor</strong></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></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>→  <strong>number</strong></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
- General left recursion removal algorithm, see EAC.
What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

The Algorithm

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

Repeat until no common prefixes remain
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_3

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

\[ A \rightarrow \alpha Z \]
\[ \beta_1 \]
\[ \beta_2 \]
\[ \beta_3 \]
Consider the following fragment of the expression grammar

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

After left factoring, it becomes

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

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

\[
\begin{align*}
\text{FIRST}(\text{rhs}_1) &= \{ \text{Identifier} \} \\
\text{FIRST}(\text{rhs}_2) &= \{ \} \\
\text{FIRST}(\text{rhs}_3) &= \{ \} \\
\text{FIRST}(\text{rhs}_4) &= \text{FOLLOW(Factor)} \\
\Rightarrow \text{It has the LL(1) property}
\end{align*}
\]
Left Factoring  (An example)

Graphically

Factor  ➔  Identifier
Identifier ➔ [  ExprList  ]
Identifier ➔ (  ExprList  )

becomes …

Factor  ➔  Identifier
Identifier ➔ [  ExprList  ]
Identifier ➔ (  ExprList  )

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

\[ \{a^n 0 b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\} \] has no LL(k) grammar

\[
\begin{align*}
G & \rightarrow aAb \\
& \mid aBbB
\end{align*}
\]

\[
\begin{align*}
A & \rightarrow aAb \\
& \mid 0
\end{align*}
\]

\[
\begin{align*}
B & \rightarrow aBbB \\
& \mid 1
\end{align*}
\]

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.
More Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4