CS415 Compilers
Syntax Analysis

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

- Homework #2 and #3 sample solutions have been posted; homework #4 solutions will be posted tonight.

- **Midterm**: Wednesday, March 7, class time slot, in Tillett hall (two rooms); closed book, closed notes, 80 minutes
  - Tillett 264: Section 01
  - Tillett 257: Section 02
  You must attend the exam for the section you are REGISTERED in!

- Review session on Tuesday, March 6, 2:00 – 3:00pm, in CoRE 301 (large conference room)
Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 3.3
Is the following grammar LL(1)?

\[ S \rightarrow a \ S \ b \mid \varepsilon \]
Is the following grammar LL(1)?

\[
S \rightarrow aSb \mid \varepsilon
\]

First(aSb) = \{ a \}
First(\varepsilon) = \{ \varepsilon \}

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

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

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

\[
\text{First}(aSb) = \{ a \} \\
\text{First}(\varepsilon) = \{ \varepsilon \}
\]

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

LL(1)? \hspace{1em} \text{YES, since} \hspace{1em} \{ a \} \cap \{ \text{eof}, b \} = \emptyset
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 $T$
- Need an algorithm to build the table

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

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

This is the $LL(1)$ table construction algorithm
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 $A \rightarrow B_1 B_2 \ldots B_k$ then
      pop Stack // get rid of $A$
push $B_k$, $B_{k-1}$, ..., $B_1$ // 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>ε</td>
<td>ε</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
**LL(1) Parser Example**

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>

( [eof, S], aaabbb, aSb) ⇒
( [eof, b, S, a], aaabbb, next input+pop) ⇒
( [eof, b, S], aabbb, aSb) ⇒
( [eof, b, b, S, a], aabbb, next input+pop) ⇒
( [eof, b, b, S], abbb, aSB) ⇒
( [eof, b, b, b, S, a], abbb, next input+pop) ⇒
( [eof, b, b, b, S], bbb, ε ) ⇒
( [eof, b, b, b], bbb, next input+pop ) ⇒ ( [eof, b, b], bb, next input+pop ) ⇒
( [eof, b], b, next input+pop ) ⇒ ( [eof], eof, 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 \( A \in NT \), 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

|| a | b | eof | other |
---|---|---|---|---|
S | aSb | ε | ε | error |

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

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

How to parse input a a a b b b?

```cpp
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 () {
    token = next_token();
    if (S () and token == eof)
        print "accept"
    else
        print "error";
} 
```
• Build FIRST (and FOLLOW) sets
• Massage grammar to have $LL(1)$ condition
  • Remove left recursion
  • Left factor it (will talk about this within the next few slides)
• 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?
What if my grammar does not have the LL(1) property?
⇒ Sometimes, we can transform the grammar

The Algorithm

∀ A ∈ NT,
    find the longest prefix α that occurs in two or more right-hand sides of A
    if α ≠ ε then replace all of the A productions,
    A → αβ₁ | αβ₂ | ... | αβₙ | γ,
    with
    A → α Z | γ
    Z → β₁ | β₂ | ... | βₙ
    where Z is a new element of NT

Repeat until no common prefixes remain
A graphical explanation for the same idea

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

becomes ...

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

\[
\text{Factor} \rightarrow \text{Identifier} \\
| \text{Identifier} [ \text{ExprList} ] \\
| \text{Identifier} ( \text{ExprList} )
\]

After left factoring, it becomes

\[
\text{Factor} \rightarrow \text{Identifier Arguments} \\
\text{Arguments} \rightarrow [ \text{ExprList} ] \\
| ( \text{ExprList} ) \\
| \varepsilon
\]

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

\[
\text{FIRST} (\text{rhs}_1) = \{ \text{Identifier} \} \\
\text{FIRST} (\text{rhs}_2) = \{ \text{Identifier} \} \\
\text{FIRST} (\text{rhs}_3) = \{ \text{Identifier} \} \\
\text{FIRST} (\text{rhs}_4) = \text{FOLLOW} (\text{Factor})
\]

\[\Rightarrow\text{It has the LL(1) property}\]
No basis for choice

Graphically

Factor → Identifier

Identifier → [ ExprList ]

Identifier → ( ExprList )

becomes ...

Factor → Identifier

Identifier → [ ExprList ]

Identifier → ( ExprList )

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\} \text{ has no } LL(k) \text{ grammar}

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

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

Read EaC: Chapter 3.4
More Syntax Analysis (top-down and bottom-up)

Read EaC: Chapter 3.4