CS415 Compilers
Syntax Analysis
Part 4

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

• Third homework due on Sunday, March 7.

• First project deadline extension: Thursday (03/04) for code, and Saturday (03/06) for report. If you submit code by Saturday (03/06), 20% penalty.

• Fourth homework to be posted by Monday, March 8.
Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 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
Parsing Techniques: Top-down parsers

**LL(1), recursive descent**

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

\[
S \Rightarrow^*_{lm} xA\beta \Rightarrow_{lm} x\delta\beta \Rightarrow^*_{lm} x\gamma
\]

**Rule:** \( A \rightarrow \delta 

? Means that we don’t know yet this part of the parse tree
Basic idea

\(\text{Given } A \rightarrow \alpha | \beta, \text{ the parser should be able to choose between } \alpha \text{ & } \beta\)

**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, \(a \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma\), 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!

\[\text{This is almost correct, but not quite}\]
The FIRST Set - 1 symbol lookahead

\[ a \in \text{FIRST}_1(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build FIRST(X) for all grammar symbols X:
1. if X is a terminal (token), FIRST(X) := \{ X \}
2. if X \rightarrow \varepsilon, then \varepsilon \in FIRST(X)

3. iterate until no more terminals or \varepsilon can be added to any FIRST(X):
   if X \rightarrow Y_1 Y_2 \ldots Y_k then
   a \in \text{FIRST}(X) if a \in \text{FIRST}(Y_i) and
   \varepsilon \in \text{FIRST}(Y_j) \text{ for all } 1 \leq j < i
   \varepsilon \in \text{FIRST}(X) if \varepsilon \in \text{FIRST}(Y_i) \text{ for all } 1 \leq i \leq k
   end iterate

Note: if \varepsilon \notin \text{FIRST}(Y_1), then FIRST(Y_i) is irrelevant, for 1 < i
The FIRST Set

\[ a \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build \( \text{FIRST}(\alpha) \) for \( \alpha = X_1 X_2 \ldots X_n \):

1. \( a \in \text{FIRST}(\alpha) \) if \( a \in \text{FIRST}(X_i) \) and 
   \[ \varepsilon \in \text{FIRST}(X_j) \text{ for all } 1 \leq j < i \]

2. \( \varepsilon \in \text{FIRST}(\alpha) \) if \( \varepsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq n \)
For a non-terminal $A$, define $\text{FOLLOW}(A)$ as

$$\text{FOLLOW}(A) := \text{the set of terminals that can appear immediately to the right of } A \text{ in some sentential form.}$$

Thus, a non-terminal’s $\text{FOLLOW}$ set specifies the tokens that can legally appear after it; a terminal has no $\text{FOLLOW}$ set

$$\text{FOLLOW}(A) = \{ a \in (T \cup \{\text{eof}\}) \mid S \text{ eof } \Rightarrow^* \alpha \ A \ a \ \gamma \}$$
To build FOLLOW(X) for all non-terminal X:

1. Place eof in FOLLOW(<goal>)

   iterate until no more terminals or eof can be added
to any FOLLOW(X):

2. If \( A \rightarrow \alpha B \beta \) then
   put \( \{ \text{FIRST}(\beta) - \varepsilon \} \) in FOLLOW(B)

3. If \( A \rightarrow \alpha B \) then
   put FOLLOW(A) in FOLLOW(B)

4. If \( A \rightarrow \alpha B \beta \) and \( \varepsilon \in \text{FIRST}(\beta) \) then
   put FOLLOW(A) in FOLLOW(B)
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 \( \text{FOLLOW}(A) \), too.

Define \( \text{FIRST}^+(\delta) \) for rule \( A \rightarrow \delta \) as

- \( (\text{FIRST}(\delta) - \{ \varepsilon \}) \cup \text{FOLLOW}(A) \), if \( \varepsilon \in \text{FIRST}(\delta) \)
- \( \text{FIRST}(\delta) \), otherwise
The \( \text{LL}(1) \) Property

A grammar is \( \text{LL}(1) \) iff \( A \rightarrow \alpha \) and \( A \rightarrow \beta \) implies
\[
\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset
\]

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

**Question:** Can there be two rules \( A \rightarrow \alpha \) and \( A \rightarrow \beta \) in a \( \text{LL}(1) \) grammar such that \( \varepsilon \in \text{FIRST}(\alpha) \) and \( \varepsilon \in \text{FIRST}(\beta) \)?
Given a grammar that has the \textit{LL(1)} property

- Problem: NT \( A \) needs to be replaced in next derivation step
- Assume \( A \to \beta_1 \mid \beta_2 \mid \beta_3 \), with

\[
\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \emptyset, \quad \text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \emptyset, \text{ and} \\
\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \emptyset \quad \text{(pair-wise disjoint sets)}
\]

/* find rule for \( A \) */
if (current token \( \in \text{FIRST}^+(\beta_1) \))
    select \( A \to \beta_1 \)
else if (current token \( \in \text{FIRST}^+(\beta_2) \))
    select \( A \to \beta_2 \)
else if (current token \( \in \text{FIRST}^+(\beta_3) \))
    select \( A \to \beta_3 \)
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. The other is a \textit{table-driven parser}.
Is the following grammar LL(1)?

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

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

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

Follow(S) = \{ \text{eof}, b \}

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

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

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

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

LL(1)? \text{YES, since } \{ 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 $T + "eof"$
- Need an algorithm to build the table

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

- 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()  // scanner call
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 $A \rightarrow B_1B_2...B_k$ 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)
use eof as bottom-of-stack marker

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>

( [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], aabbb, aSb) ⇒
( [eof, 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], 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 $S$. 

---

LL(1) Parser Example
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>

```c
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;
    }
}
```
LL(1) Parser Example

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>e</td>
<td>e</td>
<td>error</td>
</tr>
</tbody>
</table>

```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; break;
    default: return false;
    }
}
```

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

How to parse input `a a a b b b`?
More Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4