Syntax Analysis

top-down parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
• No office hours tomorrow, Thursday 02/28

• Second homework solutions available on sakai under Resources tab

• Fourth homework will be posted by Friday, March 1

• First project due dates: CODE Monday March 4; REPORT Wednesday March 6; 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

\[ S \Rightarrow_{lm}^* x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow_{lm}^* x y \]
**LL(1), recursive descent**

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

Rule: $A \rightarrow \delta$

Derivation:

$$S \Rightarrow^{*_{lm}} x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow^{*_{lm}} x y$$

Diagram:

```
  S
 /   \
A     \
   \   \δ
    \ β
     \ x
      \ y
```
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 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 (top-down)

Basic idea

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

**FIRST** sets

For some rhs $\alpha \in G$, define **FIRST**($\alpha$) as the set of tokens that appear as the first (terminal) symbol in some string that derives from $\alpha$

That is, $a \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$
The FIRST Set - 1 symbol lookahead

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

To build \( \text{FIRST}(X) \) for all single grammar symbols \( X \):

1. if \( X \) is a terminal (token), \( \text{FIRST}(X) := \{ X \} \)
2. if \( X \rightarrow \varepsilon \), then \( \varepsilon \in \text{FIRST}(X) \)

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

   end iterate

Note: if \( \varepsilon \notin \text{FIRST}(Y_1) \), then \( \text{FIRST}(Y_i) \) 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 \( 1 \leq i \leq n \)
Basic idea

*Given* \( A \to \alpha \mid \beta \), the parser should be able to choose between \( \alpha \) & \( \beta \)

**FIRST** sets

For some *rhs* \( \alpha \in G \), define **FIRST**(\( \alpha \)) as the set of tokens that appear as the first symbol in some string that derives from \( \alpha \)

That is, \( \alpha \in \text{FIRST}(\alpha) \) iff \( \alpha \Rightarrow^* \gamma \), for some \( \gamma \)

The LL(1) Property

If \( A \to \alpha \) and \( A \to \beta \) 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, but not quite*
For a non-terminal A, define $\text{FOLLOW}(A)$ as

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

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set.

$$\text{FOLLOW}(A) = \{ a \in (T \cup \{\text{eof}\}) \mid S \text{ eof } \Rightarrow^* \alpha A a \gamma \}$$
The FOLLOW Set

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 \{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 LL(1) Property

A grammar is 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 LL(1) grammar such that \( \varepsilon \in \text{FIRST}(\alpha) \) and \( \varepsilon \in \text{FIRST}(\beta) \)?
Given a grammar that has the \( LL(1) \) property

- Problem: NT \( A \) needs to be replaced in next derivation step
- Assume \( A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \), with
  \[
  \text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \emptyset, \quad \text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \emptyset, \quad \text{and} \quad \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \emptyset \quad \text{(pair-wise disjoint sets)}
  \]

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

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

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

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

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

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

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

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

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

LL(1)? 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
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→ 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)

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, 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, next input+pop ) ⇒ ( [eof], eof, 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

<table>
<thead>
<tr>
<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>
</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;
    }
}``
Syntax Analysis (top-down)

Read EaC: Chapter 3.3

Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4