Syntax Analysis

top-down parsing

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

• 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

\[ S \Rightarrow_{lm}^* x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow_{lm}^* x y \]
Parsing Techniques: Top-down parsers

LL(1), recursive descent

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

rule $A \rightarrow \delta$

$$S \Rightarrow_{lm}^* x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow_{lm}^* x y$$
If it picks the wrong production, a top-down parser may backtrack. Alternatively, one can look ahead in the input and 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.
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 (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 FIRST(X) for all single grammar symbols X:
1. if X is a terminal (token), FIRST(X) := \{ X \}
2. if X \rightarrow \epsilon, then \epsilon \in FIRST(X)
3. iterate until no more terminals or \epsilon 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
   
   \epsilon \in \text{FIRST}(Y_j) \ for \ all \ 1 \leq j < i
   
   \epsilon \in \text{FIRST}(X) \ if \ \epsilon \in \text{FIRST}(Y_i) \ for \ all \ 1 \leq i \leq k
   
   end iterate

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

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

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 $A \rightarrow \alpha \mid \beta$, 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, $\alpha \in \text{FIRST}(\alpha)$ 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!

This is almost correct, but not quite
For a non-terminal $A$, define $\text{FOLLOW}(A)$ as

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

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

$$\text{FOLLOW}(A) = \{ a \in (T \cup \{\text{eof}\}) \mid S \ eof \Rightarrow^* \alpha \ A \ a \ \gamma \}$$
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 \{FIRST(\( B \)) - \( \varepsilon \)\} 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 FIRST(\( B \)) \) 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 \( \epsilon \in \text{FIRST}(\alpha) \) and \( \epsilon \in \text{FIRST}(\beta) \)?
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
  
  $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \emptyset$, $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \emptyset$, and
  
  $\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \emptyset$ (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\(^+(\varepsilon)\) = ( First(\(\varepsilon\)) - \{ \varepsilon \} ) \cup \text{Follow}(S) = \{ \text{eof, b} \}

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

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

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

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

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

- **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 $T + \text{"eof"}$
• 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()
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 $B_k, B_{k-1}, \ldots, B_1$ // 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>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
Table-driven LL(1) parser

\[
\begin{array}{c|c|c|c|c}
 & a & b & \text{eof} & \text{other} \\
\hline
S & aSb & \varepsilon & \varepsilon & \text{error} \\
\end{array}
\]

\[
\begin{align*}
([\text{eof}, S], \text{aaabbb}, aSb) & \Rightarrow \\
([\text{eof}, b, S, a], \text{aaabbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, S], \text{aabbb}, aSb) & \Rightarrow \\
([\text{eof}, b, b, S, a], \text{aabbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, b, S], \text{abbb}, aSB) & \Rightarrow \\
([\text{eof}, b, b, b, S, a], \text{abbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, b, b, S], \text{bbb}, \varepsilon) & \Rightarrow \\
([\text{eof}, b, b, b], \text{bbb}, \text{next input+pop}) & \Rightarrow ( [\text{eof}, b, b], \text{bb}, \text{next input+pop} ) \Rightarrow \\
([\text{eof}, b], \text{b}, \text{next input+pop}) & \Rightarrow ( [\text{eof}], \text{eof}, \text{accept} )
\end{align*}
\]
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>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>error</td>
</tr>
</tbody>
</table>

([eof, $S$], aaabbb, aSb) $\Rightarrow$
([eof, b, $S$, a], aaabbb, next input+pop) $\Rightarrow$
([eof, b, b, $S$], aabbb, aSb) $\Rightarrow$
([eof, b, b, $S$, a], aabbb, next input+pop) $\Rightarrow$
([eof, b, b, b, $S$], abbb, aSb) $\Rightarrow$
([eof, b, b, b, $S$, a], abbb, next input+pop) $\Rightarrow$
([eof, b, b, b, b, $S$], bbb, $\varepsilon$) $\Rightarrow$
([eof, b, b, b], bbb, next input+pop) $\Rightarrow$ ([eof, b, b], bb, next input+pop) $\Rightarrow$
([eof, b], b, next input+pop) $\Rightarrow$ ([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></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>

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

<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>

```
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”; 
}
Recursive Descent (Summary)

- 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
$\exists$ 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

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

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

We can rewrite this as

\[
Fee \rightarrow \beta \; Fie \\
Fie \rightarrow \alpha \; Fie \\
\mid \; \epsilon
\]

where \( Fie \) is a new non-terminal

\textit{This accepts the same language, but uses only right recursion}
The expression grammar contains two cases of left recursion

\[
\begin{align*}
Expr & \rightarrow Expr + Term \\
& \mid Expr - Term \\
& \mid Term \\
& \mid e
\end{align*}
\]

\[
\begin{align*}
Term & \rightarrow Term * Factor \\
& \mid Term / Factor \\
& \mid Factor
\end{align*}
\]

Applying the transformation yields

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

\[
\begin{align*}
Term & \rightarrow Factor \ Term' \\
Term' & \rightarrow * Factor \ Term' \\
& \mid / Factor \ Term' \\
& \mid \varepsilon
\end{align*}
\]

These fragments use only right recursion
Substituting them back into the grammar yields

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{Goal} \rightarrow Expr</td>
</tr>
<tr>
<td>2</td>
<td>\textbf{Expr} \rightarrow Term Expr'</td>
</tr>
<tr>
<td>3</td>
<td>\textbf{Expr'} \rightarrow + Term Expr'</td>
</tr>
<tr>
<td></td>
<td>\quad \mid - Term Expr'</td>
</tr>
<tr>
<td></td>
<td>\quad \mid \varepsilon</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{Term} \rightarrow Factor Term'</td>
</tr>
<tr>
<td>5</td>
<td>\textbf{Term'} \rightarrow * Factor Term'</td>
</tr>
<tr>
<td></td>
<td>\quad \mid / Factor Term'</td>
</tr>
<tr>
<td></td>
<td>\quad \mid \varepsilon</td>
</tr>
<tr>
<td>6</td>
<td>\quad \mid \varepsilon</td>
</tr>
<tr>
<td>7</td>
<td>\textbf{Factor} \rightarrow \underline{number}</td>
</tr>
<tr>
<td>8</td>
<td>\quad \mid \underline{id}</td>
</tr>
<tr>
<td>9</td>
<td>\quad \mid (\textit{Expr})</td>
</tr>
</tbody>
</table>

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- General left recursion removal algorithm p.103 EAC
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,

\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma, \]

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

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

becomes ...

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

\[ \text{Factor} \rightarrow \text{Identifier} \]

\[ \quad \text{| Identifier [ ExprList ]} \]

\[ \quad \text{| Identifier ( ExprList )} \]

After left factoring, it becomes

\[ \text{Factor} \rightarrow \text{Identifier Arguments} \]

\[ \text{Arguments} \rightarrow [ ExprList ] \]

\[ \quad \text{| ( ExprList )} \]

\[ \quad \text{| } \varepsilon \]

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

Left Factoring (An example)

Factor

Identifier

[ ExprList ]

Identifier

( ExprList )

No basis for choice

becomes ...

Factor

Identifier

[ ExprList ]

Identifier

( ExprList )

ε

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 \(LL(k)\) grammar

\[
G \rightarrow aAb \\
\mid aBbb
\]

\[
A \rightarrow aAb \\
\mid 0
\]

\[
B \rightarrow aBbb \\
\mid 1
\]

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

Read EaC: Chapter 3.4