CS415 Compilers
Syntax Analysis
Top-down Parsing

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

• Midterm on Thursday, March 13
  • closed book, closed notes
  • covers everything up to (and including) top-down parsing
  • Q&A session Tuesday lecture (3/11) - come prepared!

• Report part of first project due today, March 4.
  Sakai submission website is now available. Multiple submissions are possible, but only the last one counts!

• Homework sample solution available in Sakai by Wednesday, March 5.
Review: 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. \(x \in \text{FIRST}(\alpha)\) if \(x \in \text{FIRST}(X_i)\) 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 \(1 \leq i \leq n\)
Review: The FIRST Set

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

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

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

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

end iterate

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

\[
S ::= a \ S \ b \mid \varepsilon
\]

The first set for all symbols (terminals & non-terminals):

First(a)  = \{\}
First(b)  = \{\}
First(S)  = \{\}

The first set for all RHS (right hand side):

First(aSb) = \{\}
First(\varepsilon) = \{\}
The First Set Example

\[ S ::= a \ S \ b \mid \varepsilon \]

The first set for all symbols (terminals & non-terminals):
- \( \text{First}(S) = \{a, \varepsilon\} \)
- \( \text{First}(a) = \{a\} \)
- \( \text{First}(b) = \{b\} \)

The first set for all RHS (right hand side):
- \( \text{First}(aSb) = \{a\} \)
- \( \text{First}(\varepsilon) = \{\varepsilon\} \)
For a non-terminal A, define 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.
Review: The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Initialization: place eof in FOLLOW(<goal>)

2. Iterate until no more terminals or ε can be added to any FOLLOW(X):
   If $A \rightarrow \alpha B \beta$ then
   put \{FIRST(\beta) - ε\} in FOLLOW(B)
   
   If $A \rightarrow \alpha B$ then
   put FOLLOW(A) in FOLLOW(B)
   
   If $A \rightarrow \alpha B \beta$ and ε ∈ FIRST(\beta) then
   put FOLLOW(A) in FOLLOW(B)
The Follow Set Example

\[ S ::= a \ S \ b \mid \varepsilon \]

\[ \text{Follow}(S) = \{ \} \]
The Follow Set Example

\[ S ::= a \, S \, b \mid \varepsilon \]

\[ \text{Follow}(S) = \{ \text{eof, b} \} \]
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 $\text{FIRST}^+(\beta)$ for rule $A \rightarrow \beta$ as

- $(\text{FIRST}(\beta) - \{\varepsilon\}) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\beta)$
- $\text{FIRST}(\beta)$, 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 \( \varepsilon \in \text{FIRST}(\beta) \)?
Is the following grammar LL(1)?

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

\[ S ::= aSb \mid \varepsilon \]

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)?

\[ S ::= a \ S \ b \mid \varepsilon \]

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

/* find rule for \( A \) */
if (current token \( \in \) FIRST\( ^+ \)(\( \beta_1 \)))
  select \( A \rightarrow \beta_1 \)
else if (current token \( \in \) FIRST\( ^+ \)(\( \beta_2 \)))
  select \( A \rightarrow \beta_2 \)
else if (current token \( \in \) FIRST\( ^+ \)(\( \beta_3 \)))
  select \( A \rightarrow \beta_3 \)
else
  report an error and return false

Grammars with the \( LL(1) \) 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 \( LL(1) \) 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}.
**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>

S ::= a S b ∣ ε

First⁺(aSb) = { a }
First⁺(ε) = { eof, b }
Table-driven LL(1) parser

- **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 $X ::= \beta$, 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  
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 this order  
  else report error  

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

format: (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>

Description Format:
(PDA stack content, remaining input, next action)

\[
\begin{align*}
([\text{eof}, S], \text{aaabbb}, aSb) \Rightarrow \\
([\text{eof}, b, S, a], \text{aaabbb}, \text{next input+pop}) \Rightarrow \\
(\ ) \Rightarrow \\
(\ ) \Rightarrow \\
(\ ) \Rightarrow \\
(\ ) \Rightarrow \\
(\ ) \Rightarrow \\
(\Rightarrow ( \ ) \Rightarrow \\
(\Rightarrow ( \ ) \\
\end{align*}
\]
### 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>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
([\text{eof}, S], \text{aabbb}, aSb) & \Rightarrow \\
([\text{eof}, b, S, a], \text{aabbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, S], \text{aabbb}, aSb) & \Rightarrow \\
([\text{eof}, b, b, S, a], \text{aabbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, b, S], \text{abbb}, aSb) & \Rightarrow \\
([\text{eof}, b, b, b, S, a], \text{abbb}, \text{next input+pop}) & \Rightarrow \\
([\text{eof}, b, b, b, S], \text{bbb}, \epsilon ) & \Rightarrow \\
([\text{eof}, b, b, b], \text{bbb}, \text{next input+pop } ) & \Rightarrow ( [\text{eof}, b, b], \text{bb}, \text{next input+pop } ) \Rightarrow \\
([\text{eof}, b], \text{b}, \text{next input+pop } ) & \Rightarrow ( [\text{eof}], \text{eof}, \text{accept} )
\end{align*}
\]
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

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

```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, case eof: return true; break;
    default: return false;
    }
}
```

```c
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>

```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, case 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?
Recursive Descent (homework)

Program ::= Stmtlist .
Stmtlist ::= Stmt NextStmt
NextStmt ::= ; Stmtlist | epsilon
Stmt ::= Assign | Print
Assign ::= ID = Expr
Print ::= ! ID
Expr ::= + Expr Expr |
      - Expr Expr |
      * Expr Expr |
      ID |
      ICONST

Write a recursive descent, single pass ILOC code generator.
- only single char IDs or ICONST; no blanks; single line program
- use call to “next_register()” to get a fresh, virtual register
- use call to offset(ID) to get memory location relative to base addr 1024
- use call to value(ICONST) to get integer value of constant
- hint: recursive procedures return virtual register in which computation result will be stored at runtime, or “null”.

Example:

a=2;b=3;c=+3*ab;!c.
Recursive Descent (Summary)

- 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

∀ \( A \in NT \),
find the longest prefix \( \alpha \) that occurs in two
or more right-hand sides of \( A \)

if \( \alpha \neq \epsilon \) 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
A graphical explanation for the same idea

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

becomes ...

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

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

After left factoring, it becomes:

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

First, let's compute the FIRST sets:

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

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

Factor → Identifier

Identifier → [ ExprList ]

Identifier → ( ExprList )

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? (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.
Language that cannot be LL(k)

**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

\[
G \rightarrow aAb \\
| aBbb
A \rightarrow aAb \\
| 0
B \rightarrow aBbb \\
| 1
\]

**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