CS415 Compilers
Syntax Analysis

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

• Homework #2 sample solution has been posted
• Homework #1 grades have been posted
• Upcoming deadlines:
  • Project 1 code: March 4
  • Project 1 report: March 4
  • Homework #4: March 5, at NOON!

• 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!

• Possible review session on Tuesday, March 6.
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

\[
S \Rightarrow_{lm}^* x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow_{lm}^* x y
\]

Diagram:

```
  S
 / \  \
 A  β
 / x
\   \y
```

input: read left-to-right
**LL(1), recursive descent**

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

**Rule:** $A \rightarrow \delta$

$$S \Rightarrow^{*_{lm}} x \Rightarrow^{*_{lm}} y$$

```
input: read left-to-right
```

```
construct leftmost derivation (forwards)
```

```
1 input symbol lookahead
```

```
LL(1), recursive descent
```
Predictive Parsing

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 (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

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

To build FIRST(X) for all grammar symbols X:
1. if X is a terminal (token), FIRST(X) := \{ X \}
2. if X \Rightarrow \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) \text{ for all } 1 \leq j < i
   
   \epsilon \in \text{FIRST}(X) \ if \ \epsilon \in \text{FIRST}(Y_i) \text{ for all } 1 \leq i \leq k

   end iterate

Note: if \ \epsilon \not\in \text{FIRST}(Y_1), \text{ then FIRST}(Y_i) \text{ 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 \( \varepsilon \in \text{FIRST}(X_j) \) for all \( 1 \leq j < i \)

2. \( \varepsilon \in \text{FIRST}(\alpha) \) if \( \varepsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq n \)
Predictive Parsing

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 **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!

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

$$\text{FOLLOW}_1(A) := \text{the set of terminals that can appear immediately to the right of } A \text{ in some sentential form.}$$

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

$$\text{FOLLOW}_1(A) = \{ a \in (T \cup \{\text{eof}\}) | \ S \text{ eof } \Rightarrow^* \alpha \ A \ a \ \gamma \}$$
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 \{FIRST($\beta$) - $\epsilon$\} in FOLLOW($B$)
3. If $A \rightarrow \alpha B$ then
   put FOLLOW($A$) in FOLLOW($B$)
4. If $A \rightarrow \alpha B \beta$ and $\epsilon \in$ 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

1. $(\text{FIRST}(\delta) - \{ \varepsilon \}) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\delta)$
2. $\text{FIRST}(\delta)$, 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
  
  $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \emptyset$, $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \emptyset$, 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)?

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

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

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

LL(1)?
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 $T$
- Need an algorithm to build the table

Filling in $TABLE[X,y]$, $X \in NT$, $y \in T$

- 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
LL(1) Skeleton Parser

code:

```
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 → B₁B₂...Bₖ then
      pop Stack  // get rid of A
      push Bₖ, Bₖ₋₁, ..., B₁  // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

exit on success
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], aaabbb, aSb) ⇒
( [eof, b, b, S, a], aaabbb, next input+pop) ⇒
( [eof, b, b, S], aabbb, aSB) ⇒
( [eof, b, b, b, S, a], aabbb, next input+pop) ⇒
( [eof, b, b, b, S], abbb, ε ) ⇒
( [eof, b, b, b], abbb, 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></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”;
}
```

```python
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>

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

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

```cpp
main ( ) {
    token = next_token();
    if (S () and token = eof )
        print “accept”
    else
        print “error”;  
}
```
More Syntax Analysis (top-down and bottom-up)

Read EaC: Chapter 3.4