Lecture 10:

- Parsing (cont)

Announcements:

- HW-2 out
- Quiz 3 on Friday: Grammars, LL(1)
Watch out for left recursion!

R1: \( e \rightarrow n \)

R2: \( e \rightarrow e + n \)

Q: how far do we need to look ahead for “5 + 4 + 3”?

– we have to go to the end of the expression ...
– even though we’re doing a left-most derivation!

1. Looking at 5 (1 lookahead), we don’t know whether to apply R1 or R2
2. To decide R2, need to know if the string ends in "+ n"
3. This means we have to read the entire string to know which rule to apply
4. If the string is longer than our fixed size \( k \), then we are stuck!

One solution

\[ e \rightarrow n + e \mid n \]

Q: How many look aheads needed? ... 2 (see “left factoring”)

Can rewrite left recursion to be in \( LL(k) \) ...

\[ e \rightarrow n e' \]

\[ e' \rightarrow + n e' \mid \epsilon \]

Q: now how far do we need to look ahead for “5 + 4 + 3”?
The above example involved immediate (direct) left recursion

A grammar can also have indirect left recursion

\[
\begin{align*}
  s &\rightarrow t \ a \mid a \\
  t &\rightarrow s \ b \mid b
\end{align*}
\]

- allows derivations: \( s \Rightarrow t \ a \Rightarrow s \ b \ a \)
- having strings of the form: \( a, ba, aba, baba, ababa, \ldots \)

Example rewriting for this grammar

- By replacing RHS of \( t \) in \( s \), we get:

\[
\begin{align*}
  s &\rightarrow s \ b \ a \mid b \ a \mid a
\end{align*}
\]

Now we can rewrite the above

\[
\begin{align*}
  s &\rightarrow a \ s' \mid ba \ s' \\
  s' &\rightarrow ba \ s' \mid \epsilon
\end{align*}
\]
Sometimes we need to **left factor** ...

\[ e \rightarrow \text{if } b \text{ then } s \mid \text{if } b \text{ then } s \text{ else } s \]

- here the first and second choice have a **common prefix**
- this generally means more look-ahead tokens than needed
- in this example, unless \( b \) and \( s \) are of fixed sized, there’s no fixed \( k \)

After left factoring ...

\[ e \rightarrow \text{if } b \text{ then } s \ r \]
\[ r \rightarrow \text{else } s \mid \epsilon \]

- Note that this is now **LL(1)**
What out for **ambiguous grammars**!

\[ e \rightarrow id \mid p \]

\[ p \rightarrow [ id ] \mid id \]

- here there are multiple (left-most) ways to generate an id
  
  \[ e \Rightarrow id \Rightarrow x \]
  
  \[ e \Rightarrow p \Rightarrow id \rightarrow x \]

- the problem is that these produce different parse trees
- and thus, may have different language interpretations (more later)
Check In: Can you spot any of the "LL(k)" problems in our example?

```
<stmt_list> ::= <stmt> | <stmt> ';' <stmt_list>
<stmt> ::= <var> '=' <expr>
<var> ::= 'A' | 'B' | 'C'
<expr> ::= <var> | <var> '+' <var> | <var> '-' <var>
```

Q: Is it left-recursive? No
Q: Can it be left factored? Yes
Q: Is it ambiguous? No
Q: How many look ahead tokens needed? 6 for <stmt_list> (A=B+C;...)

Check In: How would you rewrite the grammar?

A left-factored version with token types, and simplified (removed) <stmt>:

```
<stmt_list> ::= VAR ASSIGN <expr> <stmt_list_tail>
<stmt_list_tail> ::= SEMICOLON <stmt_list> | ε
<expr> ::= VAR <expr_tail>
<expr_tail> ::= PLUS VAR | MINUS VAR | ε
```
The MyPL Syntax Rules

<program> ::= ( <struct_def> | <fun_def> )* 

<struct_def> ::= STRUCT ID LBRACE <fields> RBRACE 
<fields> ::= ( <data_type> ID SEMICOLON )* 

<fun_def> ::= ( <data_type> | VOID_TYPE ) ID LPAREN <params> RPAREN LBRACE ( <stmt> )* RBRACE 
<params> ::= <data_type> ID ( COMMA <data_type> ID )* | ε 

<data_type> ::= <base_type> | ID | ARRAY ( <base_type> | ID ) 
<base_type> ::= INT_TYPE | DOUBLE_TYPE | BOOL_TYPE | STRING_TYPE 

<stmt> ::= <while_stmt> | <if_stmt> | <for_stmt> | <return_stmt> SEMICOLON | <vdecl_stmt> SEMICOLON | <assign_stmt> SEMICOLON | <call_expr> SEMICOLON 
<vdecl_stmt> ::= <data_type> ID ( ASSIGN <expr> | ε ) 
<assign_stmt> ::= <lvalue> ASSIGN <expr> 
<lvalue> ::= ID ( LBRACKET <expr> RBRACKET | ε ) ( DOT ID ( LBRACKET <expr> RBRACKET | ε ) )* 
<if_stmt> ::= IF LPAREN <expr> RPAREN LBRACE ( <stmt> )* RBRACE <if_stmt_t> 
<if_stmt_t> ::= ELSEIF LPAREN <expr> RPAREN LBRACE ( <stmt> )* RBRACE <if_stmt_t> | ELSE LBRACE ( <stmt> )* RBRACE | ε 

<while_stmt> ::= WHILE LPAREN <expr> RPAREN LBRACE ( <stmt> )* RBRACE 
<for_stmt> ::= FOR LPAREN <vdecl_stmt> SEMICOLON <expr> SEMICOLON <assign_stmt> RPAREN LBRACE ( <stmt> )* RBRACE 
<call_expr> ::= ID LPAREN ( <expr> ( COMMA <expr> )* | ε ) RPAREN 
$return_stmt$ ::= RETURN <expr> 
<expr> ::= ( <rvalue> | NOT <expr> | LPAREN <expr> RPAREN ) ( <bin_op> <expr> | ε ) 
<bin_op> ::= PLUS | MINUS | TIMES | DIVIDE | AND | OR | EQUAL | LESS | GREATER | LESS_EQ | GREATER_EQ | NOT_EQUAL 
<rvalue> ::= <base_rvalue> | NULL_VAL | <new_rvalue> | <var_rvalue> | <call_expr> 
<new_rvalue> ::= NEW ID LPAREN ( <expr> ( COMMA <expr> )* | ε ) RPAREN | 
NEW ( ID | <base_type> ) LBRACKET <expr> RBRACKET 
<base_rvalue> ::= INT_VAL | DOUBLE_VAL | BOOL_VAL | STRING_VAL 
<var_rvalue> ::= ID ( LBRACKET <expr> RBRACKET | ε ) ( DOT ID ( LBRACKET <expr> RBRACKET | ε ) )*
Summary – Things to Know

1. How to fix left-recursion.

2. How to left factor (common prefixes).

3. What an ambiguous grammar is.

4. In general, how to detect if a language is $LL(k)$ and how to determine $k$. 