Today

- Quiz 2
- Recursive Descent Parsing

Assignments

- HW2 due
- HW3 out
REVIEW: Parsing

- A context free grammar is a “generator”
- Whereas a parser is a “recognizer”
  - given a token stream
  - determine if the stream is a derivation of the grammar
- Typically a parser also builds an Abstract Syntax Tree (AST)

We’ll look at $LL(k)$ parsers

- read from left-to-right, performing a left-most derivation
- parses top down (parse tree from the root down)
- at most $k$ look ahead symbols (more later)

Consider this (modified) rule:

\[
stmt \rightarrow ( \text{`A'} | \text{`B'} | \text{`C'} ) \text{`='} expr
\]

Assuming the parser knows this rule is to be applied ...

1. it calls lexer’s next_token
2. it checks if it is a literal "A", "B", or "C"
3. it calls lexer’s next_token
4. it checks that it is an ASSIGN token
5. and so on until it finishes the $stmt$ rule

- the parser produces an error if it finds a token it isn’t expecting
Tips for $LL(k)$

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 ($n \Rightarrow 5$ and $e \Rightarrow 5$)
2. But to know if R2 should be applied, we 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

$e \rightarrow n + e | n$

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

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

$e \rightarrow n \ e'$
$e' \rightarrow + \ n \ e' | \epsilon$

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

\[
s \rightarrow t \ a \mid a
\]

\[
t \rightarrow s \ b \mid b
\]

- 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:
  
  \[
s \rightarrow s \ b \ a \mid b \ a \mid a
  \]

Now we can rewrite the above

\[
s \rightarrow a \ s' \mid ba \ s'
\]

\[
s' \rightarrow ba \ s' \mid \epsilon
\]
Sometimes we need to left factor ...

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

- here the first and second choice have a common prefix
- this means more look-ahead tokens than needed

After left factoring ...

\[ e \to \text{if } b \text{ then } s \ r \]
\[ r \to \text{else } s \mid \epsilon \]
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)
Q: Can you spot any of these problems in our example?

\[ stmt \text{ list} \rightarrow stmt \mid stmt \ ';$' \ stmt \text{ list} \]
\[ stmt \rightarrow var \ '=$' \ expr \]
\[ var \rightarrow 'A' \mid 'B' \mid 'C' \]
\[ expr \rightarrow var \mid var \ '+' \ var \mid var \ '-' \ var \]

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 \(\text{stmt \text{ list}}\) (A=B+C; ...)

Q: How would you rewrite the grammar?
Parsing: An example grammar

Simple list of assignment statements

\[
\begin{align*}
\text{stmt} & \rightarrow \text{stmt} \mid \text{stmt} \ ';$' \ \text{stmt} \ \text{list} \\
\text{stmt} & \rightarrow \text{var} \ '='$ \ \text{expr} \\
\text{var} & \rightarrow \ 'A' \mid 'B' \mid 'C' \\
\text{expr} & \rightarrow \text{var} \mid \text{var} \ '+' \ \text{var} \mid \text{var} \ '-' \ \text{var}
\end{align*}
\]

- quotes used here to help distinguish terminals from non-terminals
- Note: many possible grammars for our language!

Recall from last time:

- This grammar is not left-recursive (which is a good thing!)
- But, it is $LL(6)$ (e.g., because of $A = B + C; B = A$)
- We can reduce $k$ by left factoring

Give it a try ...
A more standard way to write the grammar (with problems fixed)

\[
\text{stmt} \text{list} \rightarrow \text{var} \; '=' \; \text{expr} \; \text{stmt} \text{list} \\
\text{stmt} \text{list} \text{tail} \rightarrow \text{';'} \; \text{stmt} \text{list} \mid \epsilon \\
\text{var} \rightarrow \text{'A'} \mid \text{'B'} \mid \text{'C'} \\
\text{expr} \rightarrow \text{var} \; \text{expr} \text{tail} \\
\text{expr} \text{tail} \rightarrow \text{'+'} \; \text{var} \mid \text{'-' \; var} \mid \epsilon
\]

Note that for parsing, it is convenient to rewrite using token types ...

\[
\langle \text{stmt} \text{list} \rangle \ ::= \text{VAR \ ASSIGN} \; \langle \text{expr} \rangle \; \langle \text{stmt} \text{list} \text{tail} \rangle \\
\langle \text{stmt} \text{list} \text{tail} \rangle \ ::= \text{SEMICOLON} \; \langle \text{stmt} \text{list} \rangle \mid \epsilon \\
\langle \text{expr} \rangle \ ::= \text{VAR} \; \langle \text{expr} \text{tail} \rangle \\
\langle \text{expr} \text{tail} \rangle \ ::= \text{PLUS \ VAR} \mid \text{MINUS \ VAR} \mid \epsilon
\]

Where \ ::= \ is used in place of \( \rightarrow \) (read as “becomes”)
Recursive Descent Parsing

A simple approach for ad hoc parsing

• divide parse into separate methods ... roughly one for each non terminal
  – corresponding grammar rule(s) encoded in each non-terminal’s method

• “descend” the parse tree using method calls (possibly recursion)

We’ll use a Parser class with basic methods and member variables:

```java
public class Parser {
  private Lexer lexer;
  private Token currToken;

  public Parser(Lexer lexer) {...}
  // start recursive descent
  public void parse() throws MyPLEXception {...}

  // move forward in token stream
  private void advance() throws MyPLEXception {...}
  // check currToken’s type and advance
  private void eat(TokenType t, String errmsg) ...
  // generate a standard error exception
  private void error(err_msg) ...

  /* recursive descent functions */
  private void stmtList() ...
  private void stmtListTail() ...
  private void expr() ...
  private void exprTail() ...
}
```
The Helper Functions

// get and store the next token
private void advance() throws MyPLEException {
  currToken = lexer.nextToken();
}

// ensure current token in stream is of given type, and advance
private void eat(TokenType t, String errmsg) throws Exception {
  if (currToken.type() == t)
    advance();
  else
    error(errmsg);
}

// handle errors
private void error(String errmsg) throws Exception {
  String s = errmsg + " found '\" + currToken.lexeme() + '\"";
  int row = currToken.row();
  int col = currToken.column();
  throw new MyPLEException("parser", errmsg, row, col);
}

Note that throwing an exception stops parser at first error

• as opposed to printing error, and then continuing
The Recursive Descent Functions

The example:

\[
\langle \text{stmt\_list} \rangle ::= \text{VAR \ ASSIGN} \ \langle \text{expr} \ \langle \text{stmt\_list\_tail} \rangle \\
\langle \text{expr} \rangle ::= \text{VAR} \ \langle \text{expr\_tail} \rangle \\
\langle \text{stmt\_list\_tail} \rangle ::= \text{SEMICOLON} \ \langle \text{stmt\_list} \rangle | \epsilon
\]

How it works:

```java
public parse() throws Exception {
    advance(); // set current token
    stmtList(); // check stmt list rule
    eat(TokenType.EOS, "expecting end of file");
}
```

```java
private void stmtList() throws Exception {
    eat(TokenType.VAR, "expecting variable"); // check if var, advance
    eat(TokenType.ASSIGN, "expecting '='"); // check if =, advance
    expr(); // check expr rule
    stmtListTail(); // check tail rule
}
```

```java
private void expr() throws Exception {
    eat(TokenType.VAR, "expecting variable");
    exprTail();
}
```

Watch out for aligning calls to advance in each recursive descent function

- either assume the current token is already advanced
- or else the function starts by advancing the current token

**Exercise**: Finish the recursive descent functions for this language