Answer to Question from Last Lecture

Q: What is an example of an unambiguous, non-LR grammar?
A: There are many, but consider

```
A ::= /* empty */
    | 'x' A 'x'
    | 'y' A 'y'
    ;
```

- This is the language $\{ww^R \mid w \in \{\mathrm{x}, \mathrm{y}\}\}$, where w^R is the reverse of w.
- It is unambiguous, since there is only one derivation for any string in the language.
- But it is not LR(k) for any k. (How can you see this?)
- In fact, there is no alternative grammar for this language that is LR(k)!

Lecture 13: Project 1 Related

CUP/JFlex interface

- Lexer communicates syntactic categories of tokens as integers.
- These may be defined in the CUP file as symbolic constants (in `terminal` declarations).
- They are converted to Java constants in the generated class

  `chocopy.pa1.ChocoPyTokens`

  which the lexer can then use.
- The lexer bundles syntactic values, semantic values, and source locations into objects of type `java_cup.runtime.Symbol`, which it returns to the parser.
- The `terminal` and `non terminal` declarations in the CUP file tell what types of semantic value the declared symbols have: both from lexical actions (for terminals) and parser actions (nonterminals).

Lexer Features

- In lexical actions, `yytext()` is a Java string containing the matched token, and `yylength()` is its length.
- Actions that execute **return** cause the lexer to deliver a token (a `Symbol`).
- Actions that don't return indicate tokens that are skipped.
- It's always the action of the longest match that gets chosen (or the first in case of ties). As a result,

```
"for"                    { return symbol(ChocoPyTokens.FOR); }
[A-Za-z][A-Za-z0-9]*  { return symbol(ChocoPyTokens.IDENTIFIER,
                                             yytext()); }
```

  will return `FOR` for the input "`for`" and `IDENTIFIER` for the input "`forage`," just as is usually intended.
- And in the case of "`forage`," the lexer will also include additional semantic information: the text of the identifier itself.

Lexer Features: Macros

- You can define abbreviations ("macros") above the first %% in the lexer file for use in patterns, as in

```
ALPHA = [a-zA-Z_]
ALNUM = [a-zA-Z_0-9]
```

which allows you to write

```
{ALPHA}{ALNUM}* { rule for ID; }
```

- Use this to simplify and clarify your actions.

Lexer Features: Using Java Directly

- The converted JFlex program is a Java program. The actions are general Java statements. Use this for "special effects", such as keeping track of indentation levels.
- The Chocopy lexical structure has been considerably simplified from Python's, so that you don't have to worry about continuation lines.
- However, if you did want to follow full Python's rules, you'd need to keep track of when you are in the midst of a bracketed construct (`'(...'`, `'[...]'`, `'{...}'`), because in those cases, newlines behave like spaces.
- Expedient solution: keep a bracket count in a variable and test in the lexical action for `"\n"` to decide whether to return a NEWLINE token.
- For indentation, you'll presumably need some sort of stack to keep track of valid levels of indentation and deal with them at the beginnings of lines.

Lexer Start States

- The lexer is essentially a DFSA that starts over in some initial state whenever the lexer's `next_token` method is called. You can define alternative starting states in this DFSA with `%state` declarations above the first %%, as in

```
%state SPECIAL
```

- This says that patterns or groups of patterns that start with `<SPECIAL>` match only when the lexer starts the machine in state `SPECIAL`, and in that state, other patterns do not match.

- In actions, one can change the start state for subsequent calls of the lexer with the call

```
yybegin(SPECIAL);
```

  to make `SPECIAL` the start state. Initially, the starting state is `YYINITIAL`.

Example

One way to handle C-style comments might be this:

```
%state COMMENT

%%

<YYINITIAL> {
   rules to use when not in a comment
   "/*"     { yybegin(COMMENT); /* But don't return yet. */ }
}
<COMMENT> {
   "*/"     { yybegin(YYINITIAL); /* Don't return yet. */ }
   [^]      { /* Matches any character.  We still don't return. */ }
}
```

Indentation and Matching Nothing

- The start-state feature can be useful when implementing `INDENT` and `DEDENT`, but we leave it to you to figure out how.
- You are likely to face one particular problem in addition: If you have a pattern intended to match indentation, it might have to match *empty* indentation (say, at the beginning of the program).
- Unfortunately, JFlex patterns won't match empty strings.
- Fortunately, there is a ~~kludge~~ useful feature: you can contrive for a pattern to match *too much* text and then return excess text to the lexer to be reprocessed.
- In lexical actions, the call `yypuahback(`N`)` will return the last N matched characters from `yytext()` to the lexer.
- We leave it to you to see where this might be helpful.

Parser Points

- Keep semantic actions simple. For the most part, you don't need much other than, e.g.,

```
statement: RETURN:r expr:e  {: RESULT = new ReturnStmt(rxleft, exright, e); :}
```

- Here, `rxleft` is "the start of the symbol labeled 'r' " and `exright` is "the end of the symbol labeled 'e' ".
- Feel free to introduce new supporting functions in the `parser code` and `action code` sections.

General Advice

- *Read the Project Documentation:* there actually is useful information there!
- *Read the Skeleton:* it gives some clues and contains work you need not do.
- *Read the Tool Documentation:* The manuals for JFlex and CUP are online.
- *Write Test Cases:* Yes, there are already some there, but it would be good to think about how to write such a test suite (and don't forget that we are holding back some tests until the deadline).
- *Use GIT:* Commit often (I have 130 commits just to change the previous year's solution to this year's). Learn how to coordinate with your partners.
- *Meet Regularly With Your Team.* Have a clear idea of what everyone's job is.