Today

- Lexical analysis (wrap up)
- Formal grammars (intro)

Assignments

- HW2 out (due Thur)
**Exercise**: With a partner, brainstorm the **token types** needed for the following MyPL code snippets.

Snippet 1:

```plaintext
print("Hello World!")
```

Snippet 2:

```plaintext
var int x := 0
while x < 10 do
  if x % 2 = 0 then
    set x := x + 3
  else
    set x := x + 1
  end
end
```

**Exercise**: With a partner, give the **token sequence** (token types and lexemes) for the code snippets. Assume the following token types:

- ASSIGN
- ID
- EQUAL
- LESS_THAN
- LPAREN
- RPAREN
- MODULO
- PLUS
- SET
- INT_TYPE
- WHILE
- DO
- IF
- THEN
- ELSE
- END
- VAR
- STRING_VAL

---

S. Bowers 2 of 11
A Lexer is implemented using either

- a lexical analyzer tool (Lex, Flex, JFlex, ...)
- or as an ad hoc program (hand written) ... we'll do this!

Lexer usually called one token at a time

- the parser asks the lexer for the next token
- the lexer reads just enough from the source code to create a token
- the token (type, lexeme, line & column number) returned to the parser

Lexer only detects errors in forming tokens ... for example:

- unexpected characters/symbols (like an exclamation mark)
- poorly formed constant values (strings, numbers, etc)
- poorly formed identifiers

Dealing with errors

- lexer returns a special error token
- lexer raises an exception ... what we'll do
- compilers stop (e.g., Python) or keep going (e.g., C++)
Some hints for implementing HW2's `nextToken()` function

Can check if end of file as well as next symbol using `peek`

```java
if (peek() == -1)
    return new Token(TokenType.EOS, "", line, column);

if (((char)peek() == '=')) {
    ...
}
```

Use `read()` to build next token one character symbol at a time

```java
string lexeme = "";  // the current lexeme
char symbol;        // the current character symbol
...
symbol = (char)read();
++column;
```

Check if a character is a digit using `Character.isDigit()`

```java
if (Character.isDigit(symbol)) {
    ... build up lexeme to hold number ...
}
```

Check if a character is a letter using `Character.isLetter()`

```java
if (Character.isLetter(symbol)) {
    ...
}
```

Check if a character is whitespace using `Character.isWhitespace()`

```java
if (Character.isWhitespace(symbol))
    return nextToken();  // recursive call
```
Use the symbol as the lexeme if the lexeme is “unimportant”

```java
if (symbol == ' .')
    return Token(TokenType.DOT, ".", line, column);
```

Throw an exception if token can’t be created

```java
string msg = "unexpected symbol '" + symbol + "'");
throw new MyPLEException(msg, self.line, self.column)
```

You don’t need to modularize your `nextToken()` function ... 

- it is okay in this case to have a relatively long function body
- you can modularize it if you want, but it isn’t required for HW 2
Syntax Analysis: The Plan ...

1. Start with defining a language’s syntax using formal grammars

2. Describe how to check syntax using recursive descent parsing

3. Extend parsing by adding abstract syntax trees (parse trees)
Formal Grammars

Grammars define rules that specify a language’s syntax

- a “language” here means a set of allowable strings

Grammars can be used within

- Lexers (lexical analysis) e.g., numbers, strings, comments
- Parsers (syntax analysis) check if syntax is correct

We’ll look at context-free grammars

- won’t get into all the details

- typically studied in a compiler or theory (formal languages) class
Grammar Rules

Grammar rules define “productions” (“rewritings”)

\[ s \rightarrow a \]

- Here we say \( s \) “produces” (or “yields”) \( a \)
  - \( s \) is a non-terminal symbol (LHS of a rule) ... sometimes as \( \langle s \rangle \) or \( S \)
  - \( a \) is a terminal symbol
  - terminal and non-terminal symbols are disjoint
  - set of terminals is the “alphabet” of the language
  - often there is a distinguished “start” symbol

Concatenation

\[ s \rightarrow ab \]

- \( s \) yields the string \( a \) followed by the string \( b \)

\[ t \rightarrow uv \]
\[ u \rightarrow a \]
\[ v \rightarrow b \]

- \( t \) yields the same exact string as \( s \)
Alternation

\[ s \to a \mid b \]

- \( s \) yields the string \( a \) or \( b \)

- the same as:
  
  \[ s \to a \]

  \[ s \to b \]

The empty string

\[ s \to a \mid \epsilon \]

- \( \epsilon \) denotes the special “empty” terminal

- \( s \) yields either the string \( a \) or \( '' \) (empty string)

Kleene Star (Closure)

\[ s \to a^* \]

- \( s \) yields the strings with zero or more \( a \)’s
Recursion

\[ s \rightarrow (s) \mid () \]

- \( s \) yields the strings of “well balanced” parentheses
- note that this is not possible to express using \( * \) (closure)
- the opposite is true through ...
- \( * \) can be implemented using recursion (w/ the empty string ...)

Q: How can we represent \( s \rightarrow aa^* \) using recursion?

\[ s \rightarrow a \mid as \]

- sometimes denoted as \( s \rightarrow a^+ \)
**MyPL Constants**

**Using grammar rules to define constant values ...**

```
BOOL_VAL  →  'true' | 'false'
INT_VAL   →  ⟨pdigit⟩⟨digit⟩* | '0'
DOUBLE_VAL →  INT_VAL '.' ⟨digit⟩⟨digit⟩*
STRING_VAL →  "" ⟨character⟩* ""
ID        →  ⟨letter⟩(⟨letter⟩|⟨digit⟩|'_' )* 
⟨letter⟩  →  'a' | ... | 'z' | 'A' | ... | 'Z'
⟨pdigit⟩  →  '1' | ... | '9'
⟨digit⟩   →  '0' | ⟨pdigit⟩
```

... where ⟨character⟩ is any symbol (letter, number, etc.) except '"'