Today

- Quiz 1
- Language implementation overview
- Lexical Analysis

Assignments

- HW1 due
- HW2 (out, due Tues)
Compilation (typically) moves in increasingly abstract steps

1. identify language “tokens” in source (more later)
2. ensure syntax is correct (“parse”) 
3. ensure source code is “correct” (nothing used before defined, no type errors)
4. generate an intermediate representation (for optimization)
5. improve performance of code (optimization)
6. generate executable code (machine code)

Example of “separation of concerns” (engineering design strategy)

- too complex to do “all at once” (similarly, single vs multi-pass)
- easier to manage, maintain, update/improve
Abstract Syntax Tree (AST) Interpreters
- execute the program directly from the AST ... what we'll do for MyPL

Bytecode Interpreters
- intermediate representation is bytecode
- usually includes optimization (back end) steps
- interpreter runs bytecode directly ... “write once run anywhere"

Just-in-time Compiler
- intermediate representation is bytecode (with optimization)
- instead of interpreting bytecode, converts to and runs machine code
- still “write once run anywhere” ... Java, JS, C#
Lexical Analysis

Tokens are the smallest meaningful units of a program

Some examples:

- Special words ("reserved" words)
  
  \texttt{int, if, while, new, class, public, and so on}

- Operators and Punctuation
  
  \texttt{+, =, ==, <=, (, ;, and so on}

- Identifiers
  
  variable names, function names, class names, etc.

- Constant values ("Literal" values)
  
  \texttt{42, 3.14, true, "abc", and so on}

White space and comments are (usually) not tokens

- some exceptions such as Python

Tokens include a \textit{type} and a \textit{"lexeme"} (a value)

- the lexeme is just the token’s value in the source file

- e.g., in the statement: \texttt{x = 42;}
  
    - the token types might be \texttt{ID, ASSIGN, INTVAL, SEMICOLON}
    
    - and the corresponding lexemes \texttt{"x", "="}, \texttt{"42"}, and \texttt{";"}

- for some tokens, the lexemes are needed for program execution

  - e.g., the variable name of the identifier ("\texttt{x}") and the int value "\texttt{42}"
Lexical Analysis Basics

- Goal is to simplify syntax analysis (parsing) ... and detect errors early
- The basic idea:

```
Source Code:          Token Sequence (as TYPE(lexeme)):
-------------------------------------------------------------------------------------------------
int fun()             INTTYPE("int"), ID("fun"), LPAREN(""), RPAREN(""),
{                      LBRACKET(""),
  int x = 0;           INTTYPE("int"), ID("x"), ASSIGN("="), INT("0"), SEMICOLON(";"),
  return x;           RETURN("return"), ID("x"), SEMICOLON(";"),
}                      RBRACKET("""
```

- Source code is converted to a sequence (or a stream) of tokens
- Removes non-tokens (white space, comments)
- Tokens also typically stored with corresponding line and column numbers
**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 x: int = 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**
- **LESS_THAN**
- **MODULO**
- **INTTYPE**
- **IF**
- **END**
- **INTVAL**
- **COLON**
- **LPAREN**
- **PLUS**
- **VAR**
- **STRINGVAL**
- **EQUAL**
- **RPAREN**
- **SEMICOLON**
- **WHILE**
- **DO**
- **ELSE**
- **SET**
- **ID**
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++)