Today

- Language Implementation (overview)
- Lexical Analysis

Assignments

- HW-1, R-1 (due Thurs)
- HW-2, R-2 (out, due next Thurs)

Reminders

- Quiz on Thursday
Compilation Steps

- Example of “separation of concerns”
  - each stage does a specific task
  - makes compiler implementation easier to manage
- Our plan is to discuss the “front end” steps ... plus write an “interpreter”
  - lexer, parser, semantic analysis (e.g., type checking)
Lexical Analysis: Tokens

Tokens are the smallest meaningful units of a program

Some examples:

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

- Operators
  
  +, =, ==, <=, and so on

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

- Constant values ("Literal" values)
  
  ints, floats, Booleans, string values, and so on

White space and comments are (usually) not tokens

- some exceptions such as Python

Tokens include a type and a "lexeme"

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

- e.g., in the statement: \( x = 42; \)
  
  – the token types might be ID, ASSIGN, INT, SEMICOLON
  
  – and the corresponding lexemes "x", "="", "42", and ";"

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

  – e.g., the variable name of the identifier ("x") and the integer value "42"
Lexical Analysis Basics

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

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

  - Source code is converted to a sequence (a stream) of tokens
  - Removes non-tokens (white space, comments)
  - Tokens stored with with their corresponding line and column numbers
Exercise: With a partner, brainstorm the token types needed for the following code snippets.

Snippet 1:

    println("Hello World!");

Snippet 2:

    while x != y do
        if x == y then
            println(x);
        elseif x < y then
            y = y - x;
        else
            x = x - y;
        end
    end
Exercise: With a partner, give the token sequence (token types and lexemes) for the following code snippets.

Snippet 1:

```plaintext
println("Hello World!");
```

Snippet 2:

```plaintext
while x != y do
    if x == y then
        println(x);
    elseif x < y then
        y = y - x;
    else
        x = x - y;
    end
end
```
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 detects errors in token (only)
For example:
• unknown symbols
• poorly formed constant values (strings, numbers, etc)
• poorly formed identifiers

Dealing with errors
• lexer might return an “error” token
• lexer could throw an exception
• compilers either stop/abort (e.g., Python) or keep going (e.g., C++)