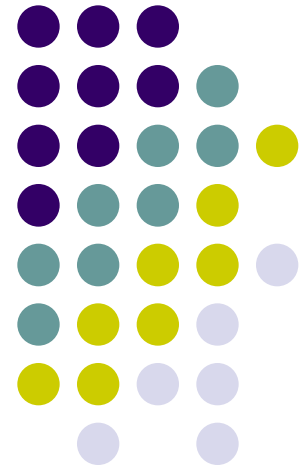


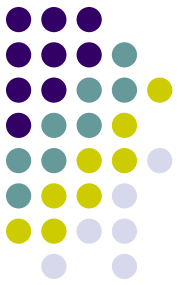
CSC402 Programming Language Implementation

Dr. Lutz Hamel
Tyler Hall Rm 251
lutzhamel@uri.edu



Welcome!

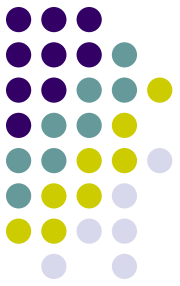




Course Objectives

- Provide a solid foundation with respect to programming language implementation including
 - grammar construction
 - parsing techniques,
 - intermediate representations (tree construction, pattern matching and tree walking techniques)
 - symbol table construction
 - code generation
- We will study a number of different programming language implementation techniques including compilers, interpreters, and virtual machines.
- You can add domain specific and general programming language implementations to your tool chest.

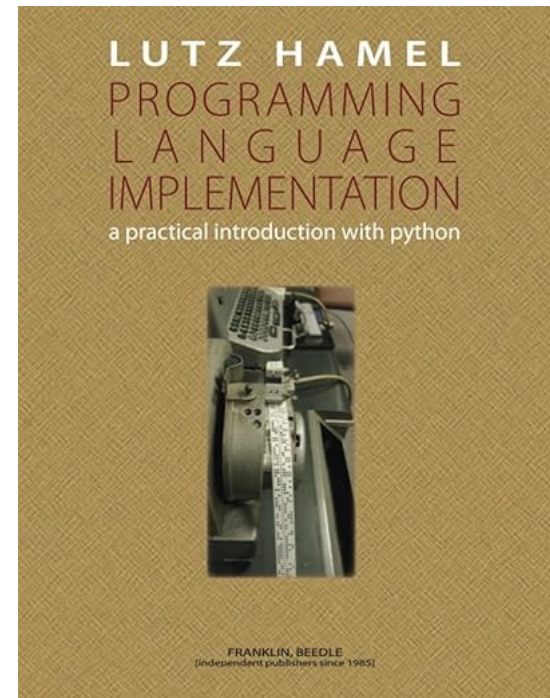
Required Textbook

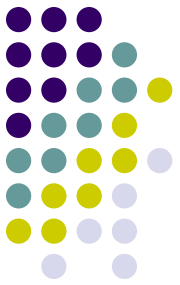


- Required textbook:

Programming Language Implementation: A Practical Introduction with Python. Lutz Hamel, Franklin & Beedle, 2024.

- Available in URI bookstore and on Amazon.





Some Definitions

- Domain Specific Language (DSL)
 - In software development a DSL is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.[‡]
 - Examples: Html, MSDOS/Linux shell scripts, game engine scripting languages

[‡] Wikipedia



Some Definitions

- General (Purpose) Programming Language[‡]
 - A general purpose programming language is a programming language designed to be used for writing software in a wide variety of application domains.
 - In many ways a general purpose language only has this status because it does not include language constructs designed to be used within a specific application domain (e.g., a page description language contains constructs intended to make it easier to write programs that control the layout of text and graphics on a page).

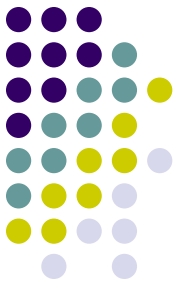
[‡] Wikipedia



Some Definitions

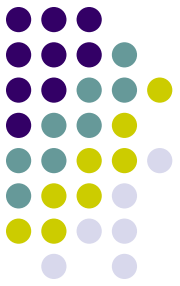
- High-Level Programming Language
 - A language that supports data abstraction and “structured programming”
 - e.g. class definitions and while-loops, if-then-else statements
- Low-Level Programming Language
 - A language that does NOT support data abstraction and “structured programming”
 - Most assembly languages and bytecodes fall into this category

The Structure of Programming Languages



- A programming language is a formal system of symbols that are combined to make up larger structures according to certain rules – **the Syntax of a Programming Language**
- The combination of symbols and the larger structures carry information which language processors need to decode.
- We will see that the architecture of language processors is geared towards extracting this information by accessing the hierarchy of symbols and structures embedded in programming languages – **Syntax Analysis**

The Structure of Programming Languages



The hierarchy (low to high):

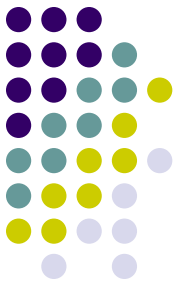
- symbol (character)
- word (token)
- phrase
- sentence

Symbols are combined to form words, words are combined to form phrases, and phrases are combined to form sentences.

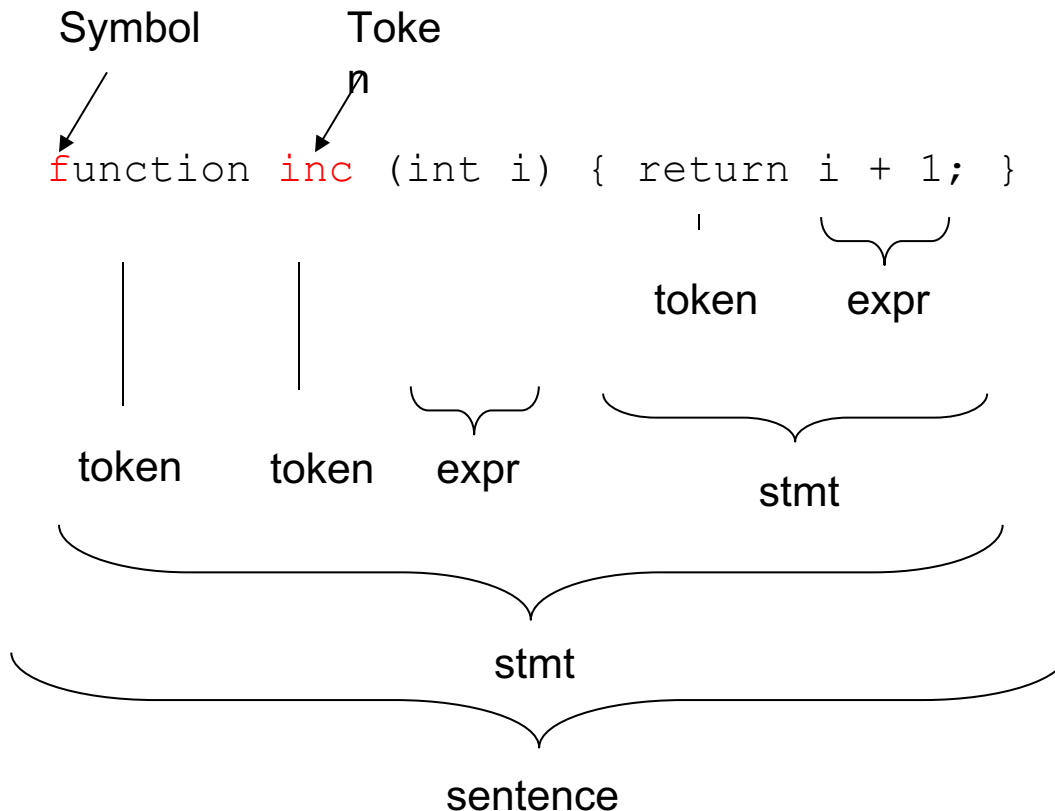
A programming language is a collection of valid sentences; a sentence is valid if the symbols, words, and phrases are combined according to the rules of the language.

These rules are usually specified using a grammar (more on that later)

The Structure of Programming Languages



An Example: Function Definition



- a function definition is a sentence, this sentence is a stmt
- the stmt is composed of two tokens (function, inc), an expr, and a stmt
- the expr is composed of four tokens: (,),int,i
- the stmt is composed of a token (return) and an expr
- the expr is composed of three tokens: i, +, 1

👉 Language processors are built to extract this kind of hierarchy and process it.

Note: the structure of a language is also called the syntax.

The Structure of Programming Languages



- Programming text page vs. Symbol Stream

- We usually represent programs as 2D text

```
i=0
while i < 10 do
    print i
    i=i+1
enddo
```

- However, to the language processor this appears to be just a stream of symbols:

```
i=0<cr>while<sp>i<sp><<sp>10<sp>do<cr><tab>print<sp>i<cr>...
```

- Here, <cr>, <sp>, and <tab> are special symbols

The Behavior of Programming Languages

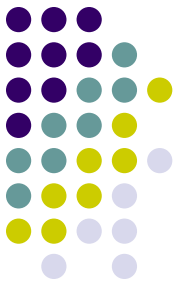


- In addition to specifying the syntax of a programming language we also need to specify its behavior – **the Semantics of the Language**
- Every programmer instinctively knows what the following program fragment does:

```
i=0
while i < 10 do
    print i
    i=i+1
enddo
```

- But we need to tell the language processor what this program means; how it should behave.

The Behavior of Programming Languages



Example of a specification:

Syntax:

WhileStatement:

while *Expression* **do** *Statement* **enddo**

Semantics:

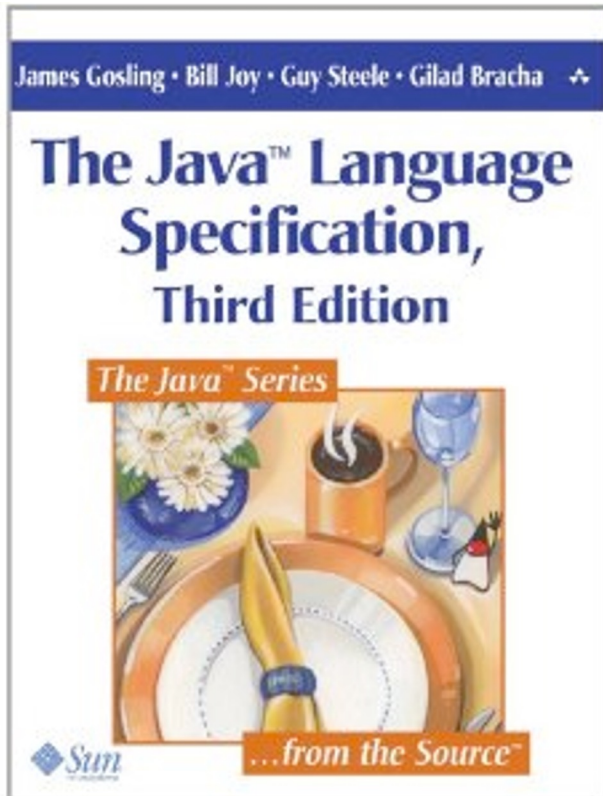
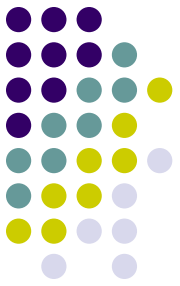
The while statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is false.

The *Expression* must have type Boolean, or an error occurs.

A while statement is executed by first evaluating the *Expression*:

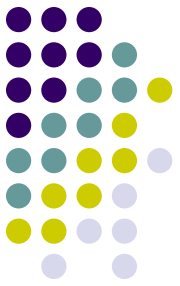
1. If the value is *true*, then the contained *Statement* is executed. If execution of the *Statement* completes normally, then the entire while statement is executed again, beginning by re-evaluating the *Expression*.
2. If the value is *false*, no further action is taken and the while statement terminates.

The Behavior of Programming Languages



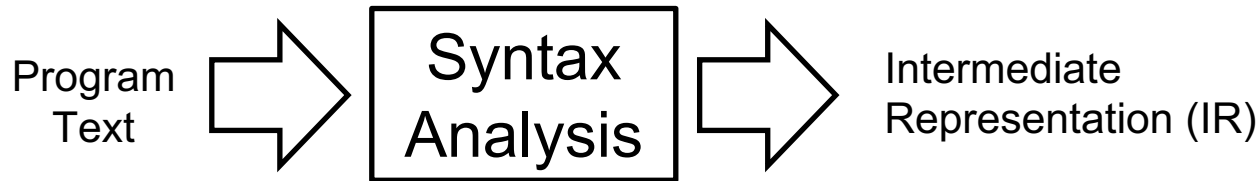
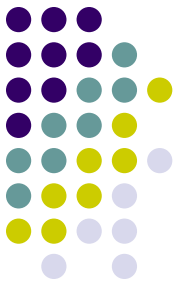
- The specification of general purpose programming languages can be very complex.
- In the case of Java this is a 700 page book!
- Domain specific programming languages tend to be less complex and therefore much easier and faster to implement.

Building Blocks of Language Processors



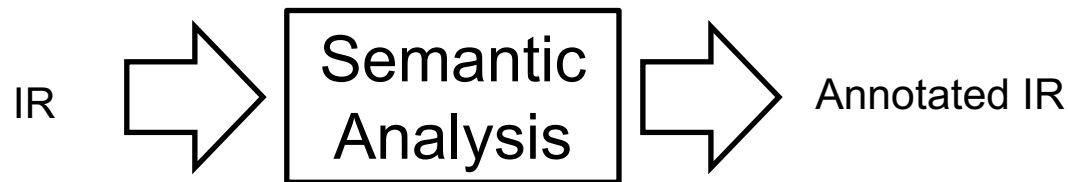
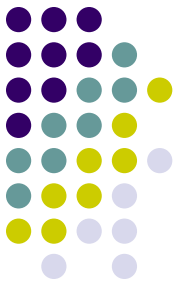
- Most programming language processors are made up of one or more three main building blocks:
 - Syntax Analysis – program text/structure analysis
 - Semantic Analysis – program behavior analysis
 - Code Generation

Syntax Analysis

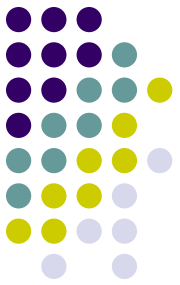


- The syntax analysis reads the program text and produces an intermediate representation (IR)
- The IR is an **abstract representation** of the program text

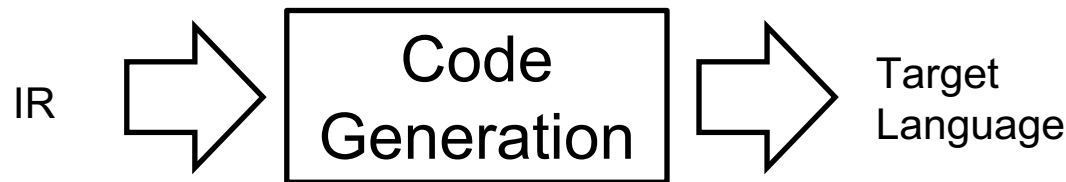
Semantic Analysis



- The semantic analysis reads the IR and analyzes the encoded behavior
- The semantics analysis typically outputs an annotated version of the IR
- These annotations insure the correct behavior of the program, for example, memory space for a declared variable.



Code Generation

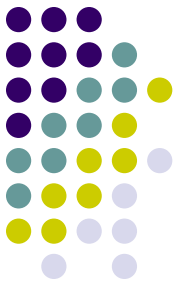


- The semantic analysis reads the IR and translates it into the target language
- The target language could be a high level language, assembly code, or byte code.
- The target code can also be a spreadsheet that summarizes data described with the IR, etc.

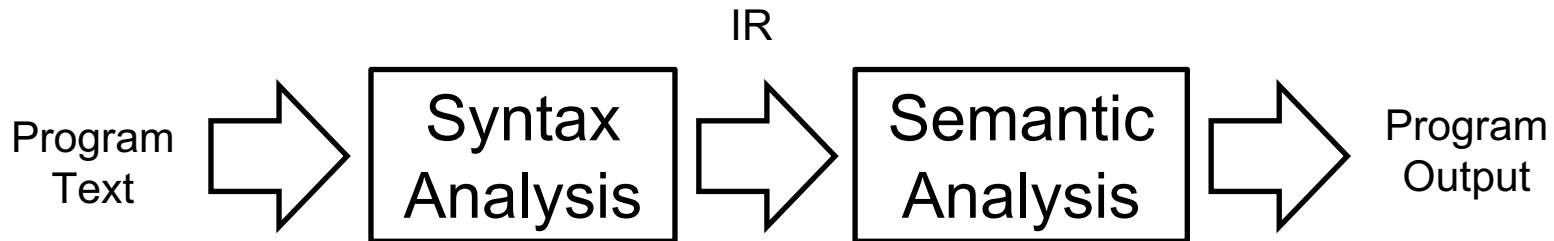
The Structure of Language Processors



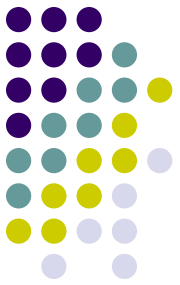
- We can now plug these building blocks together in different configuration in order to obtain a variety of language processors.
- In particular, we can configure these building blocks as:
 - Interpreter
 - Translator/Compiler
 - Simple Translator



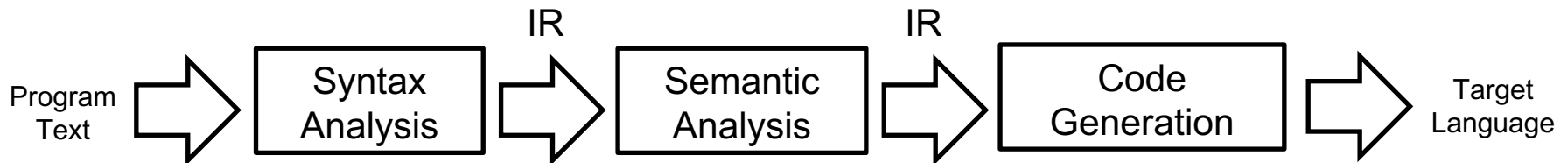
The Interpreter



- An interpreter is made up of a syntactic and a semantic analysis block.
- An interpreter reads, decodes, and executes code.
- For interpreters the semantic analysis block is slightly modified – it analyzes and **executes** the IR producing the program output.
- Examples include simple programmable calculators as well as languages such as Ruby and Python.



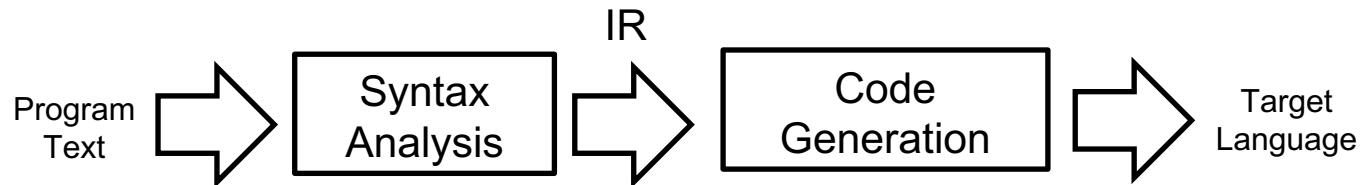
The Translator/Compiler



- A translator consists of all three of our building blocks.
- A translator reads text in one language and emits output conforming to another language.
- We often fit an additional optimization phase between the semantic analysis and the code generation phases.
- Examples include log file generators, assemblers and of course compilers.
- Note: A compiler is a translator that translates a high-level language to a low-level language.



The Simple Translator



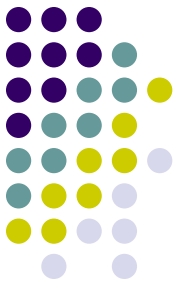
- A simple translator consists of a syntax analysis block and a code generation block
- It does not perform any semantic analysis
- Think of it as the Reader followed by the Generator.
- Examples include pretty printers and other formatters.

Example: Processing the Java Language



- A processing pipeline for a language can consist of multiple language processors.
- The language processing pipeline for Java consists mainly of
 - A compiler from Java to bytecode
 - A bytecode interpreter

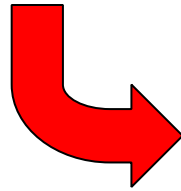
Example: Processing the Java Language



Java:

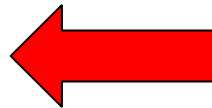
```
class Funny {  
  
    public int i = 0;  
  
    public Funny(int x) {  
        i = x;  
    }  
  
    public static void main(String[] args) {  
        Funny a[] = new Funny[10];  
  
        for (int j = 0; j < 10; j++) {  
            a[j] = new Funny(j);  
        }  
    }  
}
```

compile



Program
Output

interpret

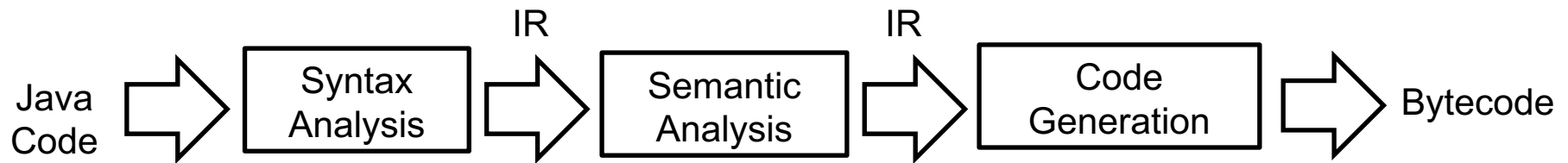
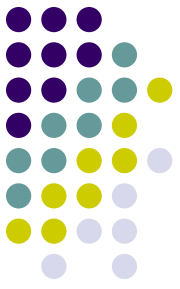


Bytecode:

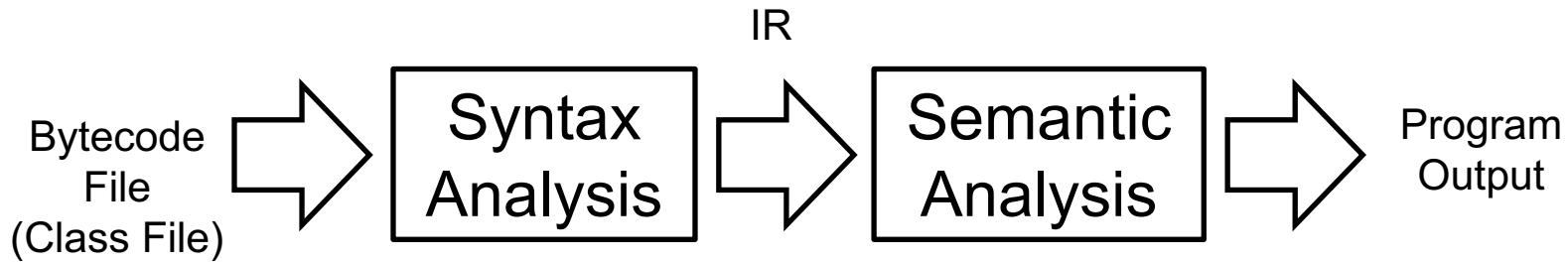
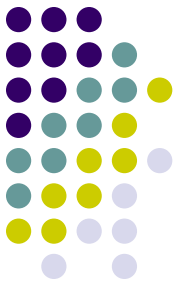
```
class Funny extends java.lang.Object{  
public int i;  
public Funny(int);  
    Code:  
    0:   aload_0  
    1:   invokespecial   #1; //Method java/lang/Object."<init>":()V  
    4:   aload_0  
    5:   iconst_0  
    6:   putfield         #2; //Field i:I  
    9:   aload_0  
    10:  iload_1  
    11:  putfield         #2; //Field i:I  
    14:  return  
public static void main(java.lang.String[]);  
    Code:  
    0:   bipush   10  
    2:   anewarray   #3; //class Funny  
    5:   astore_1  
    6:   iconst_0  
    7:   istore_2  
    8:   iload_2  
    9:   bipush   10  
   11:   if_icmpge   31  
   14:   aload_1  
   15:   iload_2  
   16:   new         #3; //class Funny  
   19:   dup  
   20:   iload_2  
   21:   invokespecial   #4; //Method "<init>":(I)V  
   24:   aastore  
   25:   iinc       2, 1  
   28:   goto       8  
   31:   return  
}
```

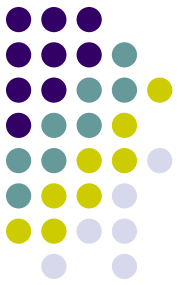
Note: javap -c <classname> will show bytecode.

Example: Processing the Java Language - Compiler



Example: Processing the Java Language – Bytecode Interpreter





Assignments & Readings

- Read Chapter 1
- Assignment #0:
 - Download & Read Syllabus
 - upload a copy into BrightSpace