

# Introduction to Programming II

## Compilers and Design

Chris Brooks

Department of Computer Science  
University of San Francisco

Department of Computer Science — University of San Francisco — p.1/71

### 6-2: Stages of Compilation

- What are the stages of the compilation of a program?

Department of Computer Science — University of San Francisco — p.2/71

### 6-3: Stages of Compilation

- What are the stages of the compilation of a program?
- Lexing - separate source code into tokens.
- Parsing - evaluate statements and generate assembly code
- Assembly - transform assembly code into binary code.
  - In Java, these are .class files.
  - Java stops here and uses an interpreter to control runtime execution.
- Other languages produce a single executable. This requires a fourth stage:
  - Linking - Multiple object files are merged into a single executable.

Department of Computer Science — University of San Francisco — p.3/71

### 6-4: Lexer

- The program can initially be seen as a really long string.
  - Sequence of characters.
- The first step is to break this string into a set of program 'building blocks'
  - Numbers, variables, keywords, operators, etc
  - These building blocks are called *tokens*.

Department of Computer Science — University of San Francisco — p.4/71

### 6-5: Tokens

- Tokens are the basic building blocks of a program.
  - x1, 12345, =, for, while
- Tokens have a value and a type.
  - ('123', Token.INTEGER), ('+', Token.PLUSSIGN), ('x1', Token.IDENTIFIER)
- Tokens can be combined to produce statements.

Department of Computer Science — University of San Francisco — p.5/71

### 6-6: Lexing

- The lexer reads an input stream from left to right and pulls off tokens as it identifies them.
- Unrecognized tokens get the type 'Token.UNKNOWN'
- Some tokens have one character and can be recognized immediately.
  - +, -, =
- Other tokens have multiple characters.
- This means that the lexer must be able to recognize the end of a token.
  - variables, numbers, keywords

Department of Computer Science — University of San Francisco — p.6/71

## 6-7: Parsing

- The job of the lexer is just to break the input stream into tokens.
  - Doesn't figure out what a statement *means*.
  - Doesn't try to determine whether an expression is legal.
- Determining whether a sequence of tokens "makes sense" is the job of the *parser*.
- This is what you'll build in Project 2.

Department of Computer Science — University of San Francisco — p.7/71

## 6-8: Parsing

- the parser does this by using a set of rules that describe legal statements.
  - expression: term
  - expression: term (plus | minus) (term | expression)
  - term: (number | variable)
- These rules are referred to as a *grammar*
- The parser finds a set of rules that match a sequence of input tokens.
  - This is called a *parse*
  - This tells how to interpret the *meaning* of a statement.

Department of Computer Science — University of San Francisco — p.8/71

## 6-9: Assembly

- In assembly, tokens or sets of tokens are replaced with bytecodes.
- This is the language that is used by the Java interpreter.
- Lower-level language - closer to the machine level
- High level languages are easier for people to work with

Department of Computer Science — University of San Francisco — p.9/71

## 6-10: Interpreter

- Java uses a separate program known as an *interpreter*
- The interpreter is responsible for:
  - Executing statements
  - Keeping track of variables and their values
  - Managing memory
- In project 2, we'll worry about this.

Department of Computer Science — University of San Francisco — p.10/71

## 6-11: Designing a Program

- Knowing the syntax of how to build a class is only the beginning.
- The bigger challenge is figuring out how to fit the pieces together.
- The software development process consists of the following steps:
  - Establishing requirements
  - Creating a design
  - Implementing the design
  - Testing
- Usually, this is an iterative, repeated process.

Department of Computer Science — University of San Francisco — p.11/71

## 6-12: Requirements

- Requirements indicate what a program is supposed to do.
- Expressed as a *functional specification*.
  - What is the input like?
  - What must the output look like?
  - Are there other programs it must interact with?
  - How quickly must the program run?
- Often, this comes from a client.
- Usually not as precise as you'd like.

Department of Computer Science — University of San Francisco — p.12/71

### 6-13: Design

- This is the 'how' part of the program.
- Specifies the classes that are needed and how they interact.
- What methods are called, what data is returned.
- Skipping this step can lead to serious, unpleasant bugs.
- A good design should make implementation straightforward.

### 6-14: Structure Charts

- A structure chart is a helpful tool for doing design.
- Helps to *divide and conquer*
- Divide the problem into smaller pieces until you reach pieces that can be tackled directly.
- This is called top-down design.

### 6-15: Example: student database

- Let's say we've been hired by USF to build an application for tracking students.
- It should be able to do the following:
  - Add new students to the database
  - Delete a student from the database
  - Enter a student's test scores
  - Print out the average of a student's scores
- These are our *requirements*

### 6-16: Top-down structure chart

- main is at the top of the chart
- Nodes below this represent portions of the program that are called.
- Arrows indicate input and output of data.
- Nodes at the next level are then decomposed in the same way.
- Eventually, we get to methods we know how to implement.

### 6-17: Top-down structure chart

- The structure chart indicates the calling sequence of the program.
- Serves as a template for the actual program.
- Annotate parameters as in, out, or in-out
- Objects called with methods on them are parameters too!

### 6-18: Choosing Objects

- Each object should have one well-defined responsibility.
- A common mistake is to "cram too much" into a single object or method.
- Methods are typically 1-2 screens of code.
- Before making a new class, does what you need already exist?

#### 6-19: Bottom-up-design

- To implement the program in our structure chart, start at the leaves and work upward.
- Bottom-level methods can be implemented directly.
- Methods at the next level up are implemented in terms of those methods.
- Eventually, you make it back up to main.

#### 6-20: Unit testing

- As you code each method, you should test it in isolation.
- Write a small program that calls this method with all expected inputs.
- Testing each method seems tedious, but it's very important.
  - A simple bug missed at a lower level can be very difficult to find later on.
  - Good programming practice can help limit the amount of time you spend debugging.