CSE 3341 Introduction

Scott, Chapter 1
Objectives

• 3341: Principles of Programming Languages
• Master important concepts for PLs
• Master several different language paradigms
  – Imperative, object-oriented, functional
• Master some implementation issues
  – You will have some idea how to implement compilers and interpreters for PLs
• Other related courses
  – 6341: Foundations of Programming Languages
  – 5343: Compiler Design and Implementation
Programming in Machine Code

• Too labor-intensive and error-prone

• Euclid’s GCD algorithm in MIPS machine code

```
27bdff00 0a000000 00000000 00000000 00000000 00000000 00000000 00000000
```

• Assembly lang
  – Mnemonics
  – Translated by an assembler
Evolution of Programming Languages

- Hardware
- Machine code
- Assembly language
- Macro assembly language
- FORTRAN, 1954: first machine-independent, high-level programming language
  - The IBM Mathematical Formula Translating System
- LISP, 1958 (LISt Processing)
- ALGOL, 1958 (ALGOarithmic Language)
- Many hundreds of languages since then
Incomplete History
Why So Many Programming Languages?

• Evolution of language features and user needs
  – Control flow: goto vs. if-then, switch-case, while-do
  – Procedures (Fortran, C) vs. classes/objects (C++, Java)
  – Weak types (C) vs. strong types (Java)
  – Error conditions: error codes (C) vs. exceptions and exception handling (C++, Java)
  – Memory management: programmer (C, C++) vs. language (Java through garbage collection)
Why So Many Programming Languages?

• Different application domains require different specialized languages
  – Scientific computing (Fortran, C, Matlab)
  – Business applications (Cobol)
  – Artificial intelligence (Lisp)
  – Systems programming (C, C++)
  – Enterprise computing (Java, C#)
  – Web programming (PHP, JavaScript)
  – String processing (AWK, Perl)
Programming Languages Spectrum

• Imperative languages
  – What are the steps the computer should follow in order to achieve the programmer’s goals?
  – “Prescriptive” attitude
  – Traditional imperative; object-oriented

• Declarative languages
  – What are the properties of the desired?
  – “Descriptive” attitude – higher level of abstraction
  – Often, lower performance than imperative languages
  – Functional; logic

• The lines are blurred – e.g., CLOS
Example: Euclid’s GCD Algorithm

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a – b;
        else b = b – a;
    }
    return a;
}  /* C procedure */
```

**C**: First, compare \( a \) and \( b \). If they are equal, stop. Otherwise, ...

assign to \( a \) ... assign to \( b \) ...

**Scheme**: same as a math definition

\[
gcd(a,b) = \begin{cases} 
  a & \text{if } a=b \\
  gcd(b,a-b) & \text{if } a>b \\
  gcd(a,b-a) & \text{otherwise}
\end{cases}
\]

```
(define gcd (a b)
  (cond ( (= a b)  a )
        ( (> a b) (gcd (– a b)  b) )
        ( else (gcd (– b a)  a) )
  )
) ; Scheme function
```
Programming Languages Paradigms

- **Imperative** (Fortran, C, Pascal, Ada)
  - Underlying model: von Neumann machine
  - Primary abstraction: procedure

- **Object-oriented** (Smalltalk, C++, Java, C#, CLOS)
  - Underlying model: object calculus
  - Primary abstraction: class or object

- **Functional** (Lisp, Scheme, ML, Haskell)
  - Underlying model: lambda calculus
  - Primary abstraction: mathematical function

- **Logic** (Prolog)
  - Underlying model: first-order logic
Why Study Programming Languages?

• Choose the right language for the job
  – They all have strengths and weaknesses

• Learn new languages faster
  – This is a course on common principles of PL

• Understand your tools better
  – Compilers, debuggers, assemblers, linkers

• Write your own languages
  – Happens more often than you’d think!

• PLs are important in computing; it is embarrassing if you do not know the basic concepts
Implementation Methods

• Compilation (C, C++, ML)

• Interpretation (Lisp)

• Hybrid systems (Java)
The Entire Compiler Toolchain (1/2)

• Preprocessor: source to source translation
  – E.g., GNU C/C++ macro preprocessor `cpp`
    • Inlines `#include`, evaluates `#ifdef`, expands `#define`
    • Produces valid C or C++ source code

• Compiler: source to assembly code
  – E.g., GNU C/C++/... compiler `gcc`
  – Produces assembly language for the target processor

• Assembler: assembly to object code
  – E.g., GNU assembler `as`
  – Translates mnemonics (e.g., ADD) to opcodes; resolves symbolic names for memory locations
The Entire Compiler Toolchain (2/2)

• Linker: **object code** from several modules (including libraries) to **single executable program**
  – E.g. GNU linker `ld`
  – Resolves inter-module symbol references; relocates the code (recomputes addresses)

• Example: **gcc** from Unix command line is a **driver program** that invokes the entire toolchain
  – `gcc -E test.c`: preprocessor (output: C code)
  – `gcc -S test.c`: preprocessor+compiler (output: assembly)
  – `gcc -c test.c`: preprocessor+compiler+assembler (output: object code for this compilation unit)
  – `gcc test.c`: preprocessor+compiler+assembler+linker
Overview of Compilation

Character stream

Token stream

Parse tree

Abstract syntax tree or other intermediate form

Modified intermediate form

Assembly or machine language, or other target language

Modified target language

Scanner (lexical analysis)

Parser (syntax analysis)

Semantic analysis and intermediate code generation

Machine-independent code improvement (optional)

Target code generation

Machine-specific code improvement (optional)

Symbol table
Source Code for Euclid’s GCD Algorithm

• This is code in Pascal, but you should have no problem reading it

```pascal
program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
    if i > j then i := i - j
    else j := j - i
  writeln(j);
end.
```
Tokens (After Lexical Analysis)

PROGRAM, (IDENT, "gcd"), LPAREN, (IDENT, "input"), COMMA, (IDENT, "output"), SEM, VAR, (IDENT, "i"), COMMA, (IDENT, "j"), COLON, INTEGER, SEM, BEGIN, ...
Parse Tree (After Syntax Analysis)
Abstract Syntax Tree and Symbol Table

```
program

(5) read

(3) read

(6) while

(3) write

(7) writeln

<> if

(6) >

(7) :=

1 INTEGER type
2 TEXTFILE type
3 INPUT 2
4 OUTPUT 2
5 GCD  program
6 I 1
7 J 1
```
Assembly (Target Language)

```
addiu sp, sp, -32  # reserve room for local variables
sw ra, 20(sp)      # save return address
jal getint         # read
nop
sw v0, 28(sp)      # store i
jal getint         # read
nop
sw v0, 24(sp)      # store j
lw t6, 28(sp)      # load i
lw t7, 24(sp)      # load j
nop
beq t6, t7, D      # branch if i = j
nop
A:
lw t8, 28(sp)      # load i
lw t9, 24(sp)      # load j
nop
slt at, t9, t8     # determine whether j < i
beq at, zero, B    # branch if not
nop
lw t0, 28(sp)      # load i
lw t1, 24(sp)      # load j
nop
subu t2, t0, t1    # t2 := i - j
sw t2, 28(sp)      # store i
b c
nop
B:
lw t3, 24(sp)      # load j
lw t4, 29(sp)      # load i
nop
subu t5, t3, t4    # t5 := j - i
sw t5, 24(sp)      # store j
C:
lw t6, 28(sp)      # load i
lw t7, 24(sp)      # load j
nop
bne t6, t7, A      # branch if i <> j
nop
nop
D:
lw a0, 23(sp)      # load i
jal putint         # writeln
nop
move v0, zero      # exit status for program
b E                # branch to E
nop
b E                # branch to E
nop
nop
E:
lw ra, 20(sp)      # retrieve return address
addiu sp, sp, 32   # deallocate space for local variables
jr ra               # return to operating system
nop
```
Intermediate Languages for Portability

- **Java**: the translator produces *Java bytecode*
  - Executed on the Java Virtual Machine (JVM)
  - Inside the JVM, there is a *bytecode interpreter* and a *just-in-time (JIT) compiler* (triggered for “hot” code)
  - Android: Java bytecode $\rightarrow$ Dalvik bytecode, for execution on the Dalvik Virtual Machine

- **C** can be used as an intermediate language: a C compiler is available on pretty much any machine