Overview of Computer Science

CSC 101 — Summer 2011
Low and High Level Languages
Compilation and Interpretation
Lecture 9 — July 18, 2011

Announcements

• Writing Assignment #3 Due Today
• Lab #3 Tomorrow
• Midterm Friday— Lectures 1 – 11
• Midterm Review on Thursday
• Does anyone know the highest level in Pac-man?

Pac-Man!

• Level 255 is the highest level in Pac-Man
• Beating it gives you this… why?
Objectives

- Low Level Programming Languages
  - Machine language
- Programming for the Web
  - Interactive (dynamic) Web pages
  - Client-side vs. server-side processing
- High-level programming languages
  - Different types for different situations
- Compilation vs. interpretation

CPU Processing

- The CPU is very good at executing simple binary instructions, one at a time
  - Computers were initially programmed directly using binary machine language
  - But, solving complex, real-world tasks using only very simple instructions is just not practical

Programming

- A program is a list of instructions that direct the computer’s processes
- CPUs can only execute very simple, binary-encoded machine language instructions (low-level language)
  - Very difficult for people to work with
- Programs are developed and written using symbolic programming languages (high-level languages)
  - Easier to comprehend
  - Easier to create
  - Easier to maintain (correct or expand)
Programming Languages

• Low-level languages
  – Machine language
    • Extremely simple executable commands in binary code
    • Example:
      add two simple numbers
      00010010001101111010 (load a number from main memory into Register “R1”)
      01100010110000111010 (add a different number into that same Register “R1”)
      00111111001110111010 (store the result from Register “R1” back into main memory)
  – Assembly language
    • Same as machine language, except mnemonic codes are used to make it easier for humans to read
    • Example:
      add two simple numbers
      LOAD 0237 R1 (load a number from main memory into Register “R1”)
      ADD 02C3 R1 (add a different number into that same Register “R1”)
      STOR 0F3B R1 (store the result from Register “R1” back into main memory)

• High-level language
  – Complex functions
  – Examples:
    • add two simple (or complicated) numbers
      \[ C + A + B \]
    • withdraw money from a bank account
      \[
      \begin{align*}
      \text{IF} & \quad \text{balance} > \text{withdrawal} ; \\
      \text{THEN} & \quad \text{balance} = \text{balance} - \text{withdrawal} ; \\
      \text{ELSE} & \quad \text{msg.out("insufficient funds"})
      \end{align*}
      \]
    • create a Web page
      <table width="100%" border="0" cellspacing="0" cellpadding="5">
        <tr class="headerbar">
          <td class="siteheader">Coffee Bean Shop </td>
          <td class="headerbar">Home | Catalog | Checkout | Contact</td>
        </tr>
        <tr>
          <td class="pagetitle">Welcome</td>
          <td>&nbsp;</td>
        </tr>
      </table>

• Programs written in high-level languages must be translated into machine language
  • Two methods of translation:
    – Compilation (or compiling)
    – Interpretation (or interpreting)
    – More on these later
Machine Language

- All operations in a processor’s instruction set are machine language operations
  - Each CPU type has its own, specific set of machine language instructions
- Machine languages have very limited capabilities
  - Even so, computers are able to do very complex things
  - Complex programs are translated from high-level languages into a very large number of very simple machine language steps by compilers and interpreters
    (More on this later when we talk about software)
- CPU designers can thus focus on optimizing the performance of a few simple operations
  - The challenge then is to be able to translate complex programs into the correct list of simple instructions

Machine Language

- Three types of machine-language instructions:
  1. Data movement operations
  2. Arithmetic and logical operations
  3. Program control operations

Machine Language

1. Data movement operations
   - Moving data from main memory to the CPU
   - Moving data from one memory location to another
   - Moving data to or from auxiliary storage
   - Input and output
Machine Language

2. Arithmetic and logical operations
   - Arithmetic on simple integers (+, −, ×, ÷)
   - Comparisons (<, >, =)

Machine Language

3. Program control operations
   - Start a program
   - End a program (halt)
   - Skip to a different instruction (instead of just fetching the next instruction)
   - Test data to decide what to do next
     
     if something is true
     then do one thing
     else do a different thing

Machine Language

- A simple machine language program might be
  
  LOAD A, R1
  MULT B, R1
  ADD C, R1
  STORE R1, D

- But, if a computer contains ONLY 0s and 1s, how can a program like this ever run?
  - Machine language program instructions are coded in binary
Many Web pages are **interactive or dynamic**
- The page changes depending on user input
- The page is generated by a small program, or **script**, that interacts with the user
  - You will be using scripts in lab tomorrow to make dynamic Web pages
  - A script can run either on the browser (the client) or at the Web site (the server)

---

Remember the **client/server paradigm**:
- Web clients are the browsers:
  - Request, receive and display information
- Web servers are the Web sites:
  - Store information
  - Process requests and send responses to the client
- A script can run on either the client or the server
  - **Client-side scripts** are best for simple tasks
  - **Server-side scripts** can handle much larger tasks

---

An Example Script

```html
<html>
<head>
<title>image script</title>
<script language="JavaScript">

var counter = 0
function changer() {
    counter = counter + 1
    if (counter == 8) { counter = 0 }
    document.images[0].src = counter + ".jpg"
}
</script>
</head>
<body>
<h1>Click on the image to change it.</h1>
<p>
<a onClick="changer()"> <img src="0.jpg"> </a>
</p>
</body>
</html>
```
Dynamic Web Pages

- Web Pages you interact with daily are usually dynamic web pages.
  - The page changes depending on user input
  - The page is generated by a script
- client-side scripts:
  - The script itself runs on the client (the browser)
  - Simple; good for small jobs
  - Portable; any browser can run the script
  - Client-side scripts are interpreted by the browser

Server-side Scripts

- Server-side scripts run on the remote server
  - Consider amazon.com
    - Want the server, not the client, to do the work of finding one book out of millions of possibilities
    - Would not make sense to download the entire catalog to the client
  - Banking, airline reservations, buying music files, etc.
    - Any task that requires large amounts of data or computation
    - Much more flexible than client-side scripts
    - The browser program can focus on being a browser
    - The server handles the big job that it is designed for
  - But, less efficient for simple tasks than client-side scripts
    - You wouldn't want to wait for a distant server just to change the picture when you click on it!

High-Level Languages

- Not all programs run on Web pages
- Most software still consists of separate application programs
  - High-level languages are used to create them
  - Usually applications are compiled
Programming Languages

- As computers have evolved over the years, so have programming languages
- One method of categorizing language development is the generational model
  - Languages are termed “1st-generation”, “2nd-generation”, etc. (up through “5th-generation” so far)

Low-level Languages

- First generation: machine language
  - Humans must program in the language of the CPU
    - Primitive, binary instructions only
    - Specific to a particular CPU type
- Second generation: assembly language
  - Humans still must use the language of the CPU
    - Same instruction set, but mnemonic codes replace binary
    - Still machine-specific

High-level Languages

- Third-generation languages (3GL)
  - Humans can use more intuitive commands which represent more complex operations
    - Each command corresponds to a (possibly long) sequence of machine language instructions
    - High-level commands translated into low-level instructions by a compiler or interpreter
  - Mostly platform independent
  - Examples:
    - FORTRAN, COBOL, BASIC, C, C++, Java, Perl, php
    - Many commonly used current languages
High-level Languages

- Fourth-generation languages (4GL)
  - Designed to be more conceptual and less procedural
  - Commands are somewhat similar to English
  - Examples:
    - PostScript, SQL, Maple, Mathematica
- Fifth-generation languages (5GL)
  - Problems are solved descriptively rather than algorithmically
  - Instead of commands describing steps to be followed, a problem is described as a series of facts and relationships
  - Examples:
    - Prolog, AI applications

Imperative Procedural Languages

- "Imperative" – an instruction or directive
  - “Do this.” “Do that.”
- “Procedural” – instructions combined in sequence to perform the overall task
- Mimic how the CPU operates
  - Step-by-step, one thing at a time
- Example languages:
  - FORTRAN, BASIC, C, JavaScript
Example: a JavaScript Program
(embedded in an HTML page)

```html
<html>
<head>
<title>image script</title>
<script language="JavaScript">
var counter = 0
function changer() {
    counter = counter + 1
    if (counter == 8) { counter = 0 }
    document.images[0].src = counter + ".jpg"
}
</script>
</head>
<body>
<h1>Click on the image to change it.</h1>
<p>
<a onClick="changer()"> <img src="0.jpg"> </a>
</p>
</body>
</html>
```

Example: a Pascal Program

```pascal
program Compound (input, output);
var
    APR, rate, amount, principal: real; { variables }
    years, time: integer;
begin { main }
    write('Enter starting amount of the investment: ');
    readln(principal);
    write('Enter the annual rate of interest (as decimal fraction): ');
    readln(APR);
    write('Enter the number of years for earned interest: ');
    readln(years);
    rate := APR / 365;
    time := years * 365;
    amount := principal * power((1.0 + rate), time);
    writeln(amount : 1 : 2)
end { main }
```

Object-Oriented Languages

• More abstract than imperative procedural high-level languages
• Programs are composed of units called objects
• Objects are programming units that
  – Have built-in procedures to do things
  – Can contain data elements
  – Interact with each other
    • Send and receive messages to and from each other
• Example languages:
  • C++, C#, Java
Example: a Java Applet

```java
// Calculating compound interest
import java.awt.*;
import java.applet.Applet;
public class Interest extends Applet {
    Label prompt1; // prompt user to input principal
    TextField input1; // input value for principal
    Label prompt2; // prompt user to input APR
    TextField input2; // input value for APR
    Label prompt3; // prompt user to input years
    TextField input3; // input value for years
    double principal, APR;
    int years;

    // setup graphical user interface
    public void init() {
        prompt1 = new Label( "Enter the initial amount of the investment" );
        input1 = new TextField( 10 );
        prompt2 = new Label( "Enter the annual interest rate (as decimal value)" );
        input2 = new TextField( 10 );
        prompt3 = new Label( "Enter the number of years invested and press Enter" );
        input3 = new TextField( 10 );
        add( prompt1 ); // put prompt1 on applet
        add( input1 ); // put input1 on applet
        add( prompt2 ); // put prompt1 on applet
        add( input2 ); // put input1 on applet
        add( prompt3 ); // put prompt1 on applet
        add( input3 ); // put input1 on applet
    }

    // process user's action on the input text fields
    public boolean action( Event e, Object o ) {
        if ( e.target == input3 ) {
            Double val1 = new Double ( input1.getText () );
            principal = val1.doubleValue();
            Double val2 = new Double ( input2.getText () );
            APR = val2.doubleValue();
            years = Integer.parseInt ( input3.getText () );
            printresult();
        }
    }

    // calculate and print the results
    public void printresult( Graphics g ) {
        double amount, rate, days;
        rate = APR / 365.0; // daily interest rate
        days = years * 365.0; // term of the investment
        amount = principal * Math.pow (1.0 + rate, days);
        g.drawString ( "The total amount accumulated in dollars: ", 50, 200 );
        g.drawString ( Double.toString ( amount ), 100, 220 );
    }
}

[Continued…]
```

Example: a Java Applet

```java
// Calculate and print the results
public void printresult( Graphics g ) {
    double amount, rate, days;
    rate = APR / 365.0; // daily interest rate
    days = years * 365.0; // term of the investment
    amount = principal * Math.pow (1.0 + rate, days);
    g.drawString ( "The total amount accumulated in dollars: ", 50, 200 );
    g.drawString ( Double.toString ( amount ), 100, 220 );
}
```

Declarative Languages

- Statements are **not** evaluated procedurally
  - Do not define a series of events or actions
  - Do not generally implement an algorithm
  - Instead generally represent a knowledge base
    - Statements of facts and relationships
      - Often used in artificial intelligence (AI) studies
- Example language:
  - Prolog
Example: A Prolog Knowledge Base

\[ \text{father(Who, Who)} \]

Example:

% One family's facts
father(padmé, luke).
father(padmé, leia).
father(anakin, luke).
father(anakin, leia).
male(ben).
male(luke).
male(anakin).
female(leia).
female(padmé).

% General family relationship rules
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \neq Y.
brother(X,Y) :- male(X), sibling(X,Y).
sister(X,Y) :- female(X), sibling(X,Y).

father(Who, father(Who, ben)).
Who=Who=luke
father(Who, father(Who, lukeluke)).
Who=Who=anakin
father(Who, father(Who, leialeia)).
Who=Who=padmé
father(Who, father(Who, leialeia)).
Who=Who=anakin
sister(sister(leialeia, , lukeluke)).
YES
grandfather(Who, grandfather(Who, ben)).
?? No rule for grandfather

Functional Languages

- Based on the idea of a mathematical function
  - A function accepts inputs and produces outputs
- Programs are created by combining functions
  - Functions are nested to send the output of one function into the input of another
  - Programs are created in a “top-down” fashion
- Example languages:
  - Lisp and derivatives (e.g. Scheme, ML)

Example: a Lisp Program

- The compound interest example again
  - Pseudocode (very procedural):
    ```
    BEGIN
    INPUT principal, APR, years
    rate = APR / 365
    time = years * 365
    amount = principal * (1 + rate)^time
    OUTPUT amount
    END
    ```
  - Functional programming (nested functions):
    ```(output (principal * \((1 + (rate \* time))\)))```
Example: a Lisp Program

- Another example: factorial
  - \(5! = 5 \times 4 \times 3 \times 2 \times 1 = 120\)
  - \(4! = 5 \times 4!\)
  - \(n! = n \times (n-1)! \quad \text{(with } 0! = 1 \text{)}\)
  - (this is an example of a recursive function)
  - Functional programming (with nested functions) is well-suited to recursive functions like this one
  - \(\text{(define } \text{(factorial } n)\}
  \quad (\text{if } (n = 0)\}
  \quad \quad \quad 1\}
  \quad \quad \quad (n \times \text{(factorial } (n-1) \text{)})\}
  \quad \})\}

Programming Languages

- Programs written in high-level languages must be translated into machine language
- Two methods of translation:
  - Compilation (or compiling)
  - Interpretation (or interpreting)

Compilation

- The entire program is translated by a compiler into machine language to form a binary executable file (*.exe)
  - The executable file contains all the binary machine language instructions for the program
  - Very efficient
    - Just happens once, at the place where the software is produced
    - The binary executable file is what you get when you buy software
  - Machine specific (not portable)
  - Compilation was invented in 1952 by Grace Hopper
    - (remember the computer bug she found?)
    - She retired from the Navy as an admiral in 1986 at age 80
Interpretation

- Although compilation translates a whole program, all at once, to create a binary executable file...
- Interpretation happens every time a program runs
  - The program is translated at run time, one line at a time, on the computer that’s running the program
    - No binary executable file is ever produced
    - Not as efficient as compilation
    - Translation happens every time the program is run
    - Portable (not machine specific)
      - Good for Web-based applications
      - JavaScript and HTML are both interpreted languages

Compilation vs. Interpretation

- Both are important and useful
  - Neither is more accurate or error prone
  - Choice depends on the software’s intended use
- Interpretation is better for
  - When portability is important
    - For example, interactive Web pages
    - Relatively simple programs
- Compilation is better for
  - Complex programs where processing speed is important
  - Both of them translate program instructions into machine language
    - They act on software, not on data
    - Compilation happens where and when the software is made
    - Interpretation happens on the computer running the program, when it’s run

Compilation vs. Interpretation

- Compilation
  - Translation all at once
    - Like translating a book
- Interpretation
  - Translation happens “on the fly”
    - Like telling a story to someone who doesn’t speak your language