Programming Concepts

Programs, Languages, and Algorithms

What are programs?

- A ***program*** is a precise sequence of instructions, intended to solve a particular problem, that can be carried out by a machine.
- Programs do not have to be run on computers; even before the age of computers, the Jacquard loom used something like punchcards to automate the action of a loom. These punchcards were effectively programs to control the loom. (You can see a short video about it here: https://www.youtube.com/watch?v=lwozgRPLVC8)
- The important part is that the instructions are ***unambiguous*** and ***precise***.

Machine instructions

- A computer is a machine that can carry out a set of precise, unambiguous instructions, called ***machine instructions***. Because computers are digital, electronic devices, machine instructions are patterns of 0's and 1's. For example,

  `000000 00001 00010 00110 00000 100000`

  is a 32-bit instruction to add registers 1 and 2 and store the result in register 6 on the MIPS processor.
- The processor is the part of the computer that executes instructions. A register is a storage cell inside the processor.

Machine languages

- The complete collection of machine instructions that a processor can carry out is called its ***instruction set***.
- ***Machine language*** is the language that defines machine instruction programs.
- Machine language is to the instruction set as written language is to a dictionary – the instruction set is the collection of all instructions, like the words in the dictionary; and machine language specifies how the instructions are put together to make programs, in the same way that a written language has rules for how sentences are created.

Assembly languages

- In the dawn of the age of modern computers, computer designers invented ***assembly language.*** Assembly language made it possible to write computer instructions in a more human-friendly form. The machine instruction

  ```
  000000 00001 00010 00110 00000 100000
  ```

  would be written in an assembly language as something like

  ```
  add $6, $1, $2
  ```

Assemblers

- It is convenient to be able to write instructions in a more readable form, but to make this possible, someone had to create a software program that would read the assembly language program and convert it to machine language.
- The software program that reads an assembly language program and creates a machine language program from it is called ***an assembler***. ***It assembles the machine code from the human-readable instructions.***

Birth of high-level languages

- Assembly language programming was still very limiting. Imagine trying to write a program containing tens of thousands of instructions using instructions like

```
add  $6, $1, $2
```

- In 1957, a team from IBM invented the first ***high-level language*** which they named (***FORTRAN***). FORTRAN made it possible to write instructions that were more like the mathematical equations to which people were accustomed. FORTRAN paved the way for other high-level languages.

High-level languages

- In a high-level language, one can write an instruction such as

  ```
  z = x + y;
  ```

  that means, "add the number stored in **x** to the number stored in **y** and put the sum into a storage cell named **z**."
- High-level languages also made it easier to get input into programs and write output from programs, with instructions like

  ```
  print z;
  ```

Compilers

- A high-level language instruction is easy for a programmer to write, but it has to get converted into machine instructions that can be executed by the computer.
- An instruction like "**`z = x + y`**" would be translated first into an assembly language instruction sequence such as

```
movl x, %eax
addl y, %eax
movl %eax, z
```

  which would be assembled into machine instructions.
- The program that translates from high-level language to assembly language is called a ***compiler***.

From problems to programs: problems

- Programs are written to solve problems. This begs the question, "***What does 'solving a problem' actually mean?***"
- To solve a problem using a computer, the problem has to be stated precisely.
- This may sound obvious and it may sound easy, but it is often not.

From problems to programs: precision

- For example, the problem, "***find the shortest distance from New York to Chicago by car***" is not very precise.
- Does it mean, on major highways only?
- If not on highways, then on all possible roads, even private ones?
- If not, then what kinds of roads?
- Where in New York does it start? Where in Chicago does it end?
- Is distance in kilometers?

From problems to programs: inputs

- The problem is usually stated in a more general way. You would probably not want to know the distance from New York to Chicago only, but you would want to have a program that, given any two cities, finds that distance.
- Thus, a problem is a general statement that can have many possible input values. In this case, (New York, Chicago) would be an input to the problem. So would (Boston, Miami) for example.
- In this case, the input is a pair of names of cities.

From problems to programs: outputs

- For ***each input value***, there must exist a ***single, correct output*** value, otherwise we cannot define a solution.
- In our travel route problem, the output is a single number, such as 1023 miles. It might also be reasonable for it to be a range, such as 1020-1030 miles, or 1025, with error ±5. This may be acceptable in certain circumstances. In either case, the output must be a unique, clearly defined, set of values.

From problems to programs: computability

- There are some problems that cannot be solved by computers. This was proved in 1931 by Kurt Gödel, and in 1936 by Alan Turing. Informally, a problem is ***computable*** if there is a procedure that solves it for all possible inputs, and always in a finite number of steps.

From problems to programs: algorithms

- Suppose we have a well-defined computable problem. An ***algorithm*** that solves that problem is a set of instructions that, for each input, finds the output for that input.
- In other words, an algorithm is what you would call a solution to the problem, a method of solving it.
- Algorithms are not language specific -- they do not have to be written in a specific programming language or in any programming language. They just have to be precise.

Algorithms: example 1

- Suppose we are given the problem, "Given an integer N > 2, print "yes" if it is prime and "no" if it is not."
- The following algorithm will solve it for us:

```
1. Let x = 2.
2. If N divided by x has 0 remainder, print "no" and stop.
3. Otherwise,
4. add 1 to x.
5. If x > N/2, print "yes" and stop.
6. If not, go back to step 2.
```

Designing algorithms

- The challenge and excitement of software development lies in algorithm creation.
- Given a problem, finding an algorithm that solves it lies at the core of computer science. It is the creative step, the part that requires imagination and inventiveness.
- Once you have found the algorithm, the rest is getting it to work. It is exciting to have a finished product, of course, but the "bang per buck" is less.

Programs

- Once you have figured out what the algorithm is, the next step is to convert that algorithm into a program in a ***programming language***.
- A programming language is a language in which you can write programs that can be translated into machine instructions for the computer to execute.
- English is not a programming language. Perl is. So is C. What is the difference between English and Perl? ***Precision***.

Ambiguity in languages

- Natural languages, such as English, can be ***ambiguous***. Consider these sentences:

    The chickens are too hot to eat.

  Are the chickens too hot for them to eat, or to be eaten?

    I said I would tell you on Friday.

  Was it on Friday that I said I would tell you, or did I say at some unknown time that I would tell you on Friday?

    Students like annoying professors. (They do?)

  Do students like to annoy professors, or do they like professors who are annoying?

Precision in programming languages

- ***Programming languages must not be ambiguous***. In a programming language, every command or instruction has a single, precise, predetermined meaning.
- For any programming language, there is a document that says exactly the form and meaning of each instruction.
- Form is called ***syntax*** and meaning is called ***semantics***.
- As a programmer, your job is using the instructions correctly, which means understanding the ***syntax*** and ***semantics*** of the language and making them second nature.

Syntax and semantics

- The concepts of syntax and semantics are ideas that pertain to natural languages like the ones you speak, and to all ***formal languages***, such as programming languages.
- When you write a correct English sentence, you are observing the rules of ***English grammar***. The ***grammar*** defines the syntax of the sentence as well as part of its semantics. When you read a sentence and determine what it means, you are implicitly using your knowledge of English semantics.

Learning To program

- Learning to program moves in two parallel directions at the same time.
- On the one hand, you need to learn the syntax and semantics of some programming language. In this course, it will be Perl.
- On the other hand, you need to learn algorithm development, independently of any particular programming language. There are "tricks of the trade" that can be learned, and some methodology, but at all times, it requires analysis of the problem and thoughtfulness.

About Perl

- Perl was written by Larry Wall, and stands for either Practical Extraction and Report Language, or Pathologically Eclectic Rubbish Lister. It all depends.
- Perl can be used to do many of the tasks commonly done using shell scripts.
- Perl is also a full-fledged programming language.
- Perl is not a filter, but filters can be written in Perl. In fact, you can write a Perl program to do anything that can be done in *grep, awk, sed, sort, tr, or any other Unix filter you can think of.
- The Perl motto is, "***There's more than one way to do it***."

More about Perl

- Larry Wall designed Perl to be as natural as possible. He was a linguist by training, so it has a natural language feel.
- To make the most of Perl you need to have "the three great virtues of a programmer: ***laziness, impatience, and hubris***." (from Programming Perl, p.xiii)
- Perl tends to do what you would want it to do.
- You don't need to know a lot of Perl to use it.
- Perl has been called “Internet glue” and “Internet duct tape” because it is used for writing so many web servers and web pages.

Compilers versus interpreters

- Shells are ***interpreted*** -- a shell script is ***parsed*** and ***executed one line at a time***. This is a slow process. It also has a big drawback -- your script can have a syntax mistake that is not detected until after part of it has been executed. Your data can become corrupted as a result.
- Languages like C and C++ are compiled -- the entire program is translated into machine code before any of it is run. It runs faster than a script, but each time you want to make a change, however, small, it has to be recompiled.

Is Perl compiled or interpreted?

- Perl is both compiled and interpreted:

  A Perl program is first compiled into lower level code that is then interpreted by the Perl engine. This means that all syntax errors are reported before any instructions are carried out, and once it is compiled, the program runs much faster than a shell script.
- This makes Perl programs slower than C programs, but much faster and safer than shell scripts. You shouldn't build applications with Perl, but you certainly can build utilities and commands.

Running Perl: method 1

- You can write a Perl script on the command line in single quotes and execute it as a command, with the appropriate command line switches. For example:

  ```
  $ perl -e 'print "Voila.\n";'
  ```

  will print

  ```
  Voila.
  ```

  on the console.
- This is fine for short scripts.

Running Perl: method 2

- For longer scripts, you can put the script into a file and save the file.
- Create a file with a suitable name. I will call it **`voila`**. **`P`**ut the single line

```
print "Voila.\n";
```

into it. Then you can type

```
$ perl voila
```

and it will print

```
Voila.
```

on the console.

Running Perl: method 3

- You can make the file containing your Perl script executable, and make the following line the very first line in the file:

```
#!/usr/bin/perl
```

- The file would look like:

```
#!/usr/bin/perl
print "Voila.\n";
```

- You can then type the file name like a command

```
$ voila
```

  and it will print `Voila.` on the console.

Running Perl: method 4

- The preceding method would only work if the Perl interpreter was actually located in **`/usr/bin`**. If you copied the program to another computer in which the path to Perl was:

  ```
  /usr/local/bin/perl
  ```

  it would not run. The portable way to write this program is:

  ```
  #!/usr/bin/env perl
  print "Voila.\n";
  ```

- As long as the **`perl`** executable is in your **`PATH`** in the environment, this will work correctly.

What comes next

- In the next lesson, you will start to learn the syntax and semantics of Perl and see how a simple problem is solved.
- We will begin with the basics and build from there.