Unix: A Programming Language Part 1: Train Schedules Introduction

Unix is more than a system for running applications and servers. It is a programming language. What does that term mean? Consider popular programming languages like Java and C++. These languages provide three things: (a) A set of functions and operations you can apply to data, (b) a syntax for combining those functions and operations into programs, and (c) a way of executing those programs.

Unix provides these three things, too. Unix provides a large number of tools that operate on data, Unix provides a syntax (the shell scripting language) for combining those tools into programs, and Unix provides a way to execute those scripts.

Unix programming consists of writing scripts that invoke tools to process data. Sometimes, though, there is no tool for a particular job. In that case, you can write a new tool and add the tool to the system. For Unix, tools are usually written in C. Programs in C are fast, portable, and compact.

In this course, we shall see how to use existing tools to build Unix applications, how to write new tools in C, and how to make these applications accessible via web pages. We begin with an example. The Problem: Processing Train Schedule Data

The MBTA commuter rail system runs lots of trains each day on several lines. The MBTA web site provides access to information about those trains and lines. Travelers can view maps and schedules on the web site. Travelers can also ask the system how to get from one place to another. The computer checks the schedules and reports which trains to take and where to change trains if necessary. How does that all work?

Imagine the MBTA had asked you to build their system. Consider the following plausible scenerio. Someone in the MBTA transcribed all printed train schedules into a spreadsheet and then saved the spreadsheet as a big text file. This person read down each column on each schedule recording every stop every of every train. Here is the format:
`TR=070;dir=i;day=m-f;TI=5:40;stn=greenbush;Line=greenbush TR=070;dir=i;day=m-f;TI=5:47;stn=north scituate;Line=greenbush TR=070;dir=i;day=m-f;TI=5:54;stn=cohasset;Line=greenbush` [illegible] name of the station, and Line is the name of the train line.

Our project is to build programs and a web interface to do three things: 1. Generate statistics and lists 2. Search the database 3. Plan trips Unix as a Data Management System

There are lots of data management programs and web interface systems. You may have used some. We shall learn about Unix and C programming by solving this problem by using Unix as a data management system and programming toolkit.

Some of the tasks are easy, and some require more sophisticated tools. As we progress, we shall learn about Unix tools and write some special purpose tools for this project. Getting Started: Simple Statistics and Reports

Some Questions:

1. When do trains leave from Braintree going to Boston?

2. What is the time of the earliest train from Ashland to Boston?

3. How many trains stop at West Medford on a weekday?

4. List the stations on the Fitchburg line.

5. List all the lines in the system.

6. List the train numbers of all trains passing through Beverly Depot.

7. [illegible]

8. Which line has the most stations?

9. During what hour does the greatest number of trains arrive at South Station? 10. When does the last train to Worcester leave Boston? 11. What is the most common train stop time? 12. What is the longest train trip (time, not distance) on the system? Some Tools: grep searches a file for lines that contain a specified pattern. The program can print out all matching lines or all lines that do not match. Ex: Find all trains stopping at natick. grep stn=natick sched cut treats each line as a sequence of delimited items and prints from each line only items in specific positions. Ex: Extract time and station fields from the file. cut -d";" [illegible]

grep stn=waltham sched | grep "dir=i" | grep day=m-f | cut -d";" -f4 | cut -d= -f2 | sort -n | head -3 wc counts words, lines, and characters of its input. The default output is to list all three. Using the -l, -w, or -c options limits counting to lines, words, and characters, respectively. Ex: 1) Count number of stops at ipswich, 2) count number of stations on the lowell line. grep stn=ipswich sched | wc -l grep "Line=lowell" sched | cut -d";" -f5 | sort | uniq | wc -l Puzzle1: Can you answer these questions using these six tools? Do you need any additional tools?

Puzzle2: How many other questions can you answer about this data set using these six tools?

Once you start programming with text-processing tools, you can discover how much data analysis you can do by combining special purpose, general tools in the correct order. Unix is designed to help you combine tools into programs.

Combining Tools: Pipelines and Scripts

Each tool performs a single function -- search, sort, count, cut. Unix provides two ways to combine tools: the pipeline and the script.

=> *pipelines*

A pipeline is like an assembly line in a factory: information passes from one worker to the next. Each tool performs one operation on the data. For example: grep TR=051 sched | wc -l combines the searching program and the counting program. The grep command outputs all the lines in the schedule for train number 051, and the pipe sign (the vertical bar) tells Unix to make that output the input to wc -l. The counting program reads the set of lines and outputs the number of lines. That number appears on the screen.

Any number of commands may appear in a pipeline. grep TR=051 sched | cut -d= -f5-6 | sort -n | cut -d";" -f1,2 | nl The commands in the pipeline run at the same time, just as all the workers on an assembly line work simultaneously. In practice, one tool may need to wait until the preceding tool completes part of the work, or until the CPU is available for the tool. Nonetheless, Unix tries to run the tools in the pipeline as close to simultaneously as it can.

A pipeline is one instance of a more general Unix programming feature: *input/output redirection*. In the case of a pipeline, you connect output of one program to the input of another program. The shell syntax also allows you to send output not to the screen, not to another program, but instead to a file on the disk. Similarly, you can arrange for a tool to read its input, not from the keyboard, not from another program, but instead from a file. Using input/output redirection allows you to read information from files, send it through several processing tools, and then output the result to another file.

=> scripts

A script is a file that contains a sequence of commands and pipelines. Unix executes the script by performing each command as if it were typed on the command line. The syntax for shell scripts includes all the features of a programming language: variables, control flow, and functions.

The following script prints out the times of all trains passing through a specified station: `#!/bin/sh # # train-times # purpose: list train times for a station # usage: train-times # action: script prompts for station name # echo "Which station? " read STATION echo "inbound or outbound (i/o)? " read DIR echo "Trains passing`

through $STATION"   grep "stn=$STATION" sched | grep "dir=$DIR" The output of this script is not in a user-friendly format. We could make this script part of a pipeline, but there is a problem. The script prompts for a station name and a direction. A cleaner way to pass those two values to the script is to pass them as command line arguments. That is, we can modify the program so we can type: train-times   wakefield   o where the station is the first argument and the direction is the second argument. A command-line argument version of the script is: #!/bin/sh # # train-times-args #   purpose: list train times for a station #   usage: train-times-args stationname direction #   where: direction is "i" or "o" #   STATION=$1   DIR=$2
        grep "stn=$STATION" sched | grep "dir=$DIR" The difference between these two versions of the script is important. The first script interacts with the user to get the values it needs. The second script gets the values from the command line.

Designing scripts that accept arguments on the command line makes them into tools that can be included in pipelines and in other scripts. For example: train-times-args salem o | cut -d";" -f1,3,4 But, sooner or later we need to add a nice user interface, don't we? Web Pages as Nice User Interface

The web lets you put nice user interfaces on your shell scripts. You need a way to get the values for the user and a way to read those values and pass them to the shell script that does the work.

Therefore, putting a shell script on the web requires only two things: (a) An HTML form, and (b) a script to process the form. Here is an example of how to put the train-times-args script on the web:

The HTML Form: train-times.html <html> <body> Find train times for a station <p> <form action="train-times.cgi" method="get">

```
    Station: <input type="text" size="25" name="station"><br>
    Direction (i or o): <input type="text" size="1" name="dir"><br>
    <input type="submit"> </form>
```

The Form Processing Script: train-times.cgi #!/bin/sh # processing script for train-times.html

```
    eval $(./qryparse)
    echo "Content-type: text/plain"
    echo ""

    echo "Train times for $station direction $dir"
    ./train-times-args "$station" "$dir"
```

Three Languages - One Big Idea This course teaches three languages, one for each level in the picture The user interface level is based on HTML web pages using HTML forms to collect user input. We shall learn how to design these pages and forms. Shell scripts combine tools to process data. Scripts can be used directly from the command line, and scripts can receive data from web forms and invoke scripts and tools to do the work. The shell is a programming language with variables, control flow, functions, and many other features. We shall learn how to program in this language. Most Unix tools are written in C. We shall learn how to program in C and how to design programs that can be used as software tools.