Wilfred Hughes

Oosh: An Object Oriented Shell

Computer Science Part II

Churchill College

May 12, 2010

Proforma

Name:	**Wilfred Hughes**
College:	**Churchill College**
Project Title:	**Oosh: An Object Oriented Shell**
Examination:	**Computer Science Part II, 2009-2010**
Word Count:	**9700**
Project Originator:	**Wilfred Hughes**
Supervisor:	**David Eyers**

Original Aims of the Project

I planned to write a Unix shell from scratch in Python. It would enforce structure on data piped between processes, so that programs could reason about data at a higher level. It would also enable transparent network-based access using a lazy data fetching methodology. Finally, the shell syntax would be streamlined so as to minimise command length for common interactions.

Work Completed

I created an interactive shell with support for both new metadata-aware commands and traditional Unix commands. It also offered a basic command history and pretty prints all structured data.

I created a set of commands that understand the data structures that I defined. These included commands that produce data, commands that manipulate data and an automatic graphing utility.

For networking I created a server that allows users to log in and execute commands as if they were being run locally, with the client dynamically moving commands to minimise bandwidth.

Special Difficulties

None.

Declaration

I, Wilfred Hughes of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

There have been three previous Part II projects in this area that I am aware of. I have examined them briefly, but they explored different shell topics to Oosh. These are J. Tippell in 2006, J. Harbin in 2005 and G. Beasley in 2003.

Chapter 1

Introduction

1.1 Historical Motivations

The command line shell (henceforth *shell*) is one of the earliest user interfaces in the history of computing, with recognisable shells having appeared by 1965 on the Multics system [1]. A shell offers the user a convenient abstraction to interact with his system. He may perform common tasks using considerably shorter commands than using his programming language of choice. Whilst a shell scripting language is a programming language in its own right, it tends to be optimised for writing simple scripts and does not always feature major programming language conveniences such as function definition.

The earliest shell that included modern shell features was the Bourne shell (known as `sh`), developed by Stephen Bourne. It was released in 1977 as a part of Version 7 Unix. The most common shell in use today is Bash, which offers a superset of the features of `sh`. Bash seeks backwards compatibility with the Bourne shell wherever possible. Sadly the Bourne shell was a result of hacking and experimentation, so later attempts to retroactively produce a formal grammar that describes its behaviour have failed [2].

This pursuit of backwards compatibility has limited the scope for radical change to shell metaphors. The language syntax they accept is unnecessarily complex, making it hard for the user to exercise confidence in writing scripts. Further the language has not been expanded or changed to make tasks that are typical today easier.

I revisited these assumptions. How could I make today's shell usage simpler? What other use cases would be desirable but are not supported by shells that are currently available?

1.2 The Purpose of the Shell

I wanted to reconsider what it meant for a program to be a shell. A shell is not as expressive as a mainstream programming language. Shell scripting languages typically have far fewer primitive data structures, with some shells only offering a string data type. They also only offer a very limited range of arithmetic operations, usually only basic integer maths.

Today's scripting languages fill a different niche. Python and Ruby have both become popular and both offer an interactive mode of operation. However they are more general purpose and so their syntax is too heavyweight for the typical system administration tasks that shells are used for. A simple directory listing in Python requires the user to

```
>>> import os
```

to import the necessary libraries then to run

```
>>> for x in os.listdir('.'): print(x)
```

A shell is designed to suit precisely this type of task.

A shell is therefore optimised for a specific type of usage. Almost everything is expected to be handled by an external process according to the Unix design philosophy of "do one thing and do it well". As a result arithmetic is handled by `bc`, file deletion by `rm`, compression by `gzip`, and so on. Consequently administration tasks become much easier in a shell than using a full scripting language interpreter.

1.3 Existing Solutions

I evaluated several well-known shells available today. One of the most popular options is Bash, which is the default on most GNU/Linux distributions and Mac OS X (since version 10.3).

A more recent alternative to Bash is the Friendly Interactive Shell ('Fish'), released in 2005. Fish is not a substantial departure from Bash, but breaks backwards compatibility. The design of Fish [3] is motivated by a desire to create a minimal, orthogonal scripting language. Whilst it achieves these goals it still requires the user to perform dumb manipulations on unstructured text.

Another recent development is PowerShell, a Microsoft scripting language and shell released in 2006. PowerShell is a more radical departure from traditional shell design, offering an object-oriented approach. Whilst this offers an increase

in expressive power, the PowerShell design does not seek to optimise the interface for interactive usage. Commands within the PowerShell system ('cmdlets') use a more verbose noun-verb naming scheme (contrast `ps` with `Get-Process`) and the programmer is sometimes exposed to underlying Windows API for simple tasks such as basic networking.

None of these existing solutions have support for transparent networking. Bash-like shells require the user to `ssh` into the desired system, so building a pipeline with commands running on varying hosts is non-trivial. PowerShell solutions are more heavyweight and use inconsistent calling conventions for different cmdlets.

1.4 My Design

I decided to create a completely new shell with a new language that would be familiar to experienced shell users. I would re-evaluate design decisions made by previous shell developers. My design would seek to make today's common shell tasks easier to invoke.

In addition to the language I would develop a data format that enables command line tools to output data in a more structured manner and include metadata that would be machine-readable. To demonstrate the versatility of this data structure I would create a number of programs that could produce and/or manipulate data in this format.

Since networking is a much more prominent part of today's computer usage, I concluded that my shell should include networking as built-in functionality rather than leaving it to an external program. This would require me to develop a simple client/server architecture. Since the shell itself would be aware of where commands were running, it would be able to use an intelligent data transfer scheme to minimise the bandwidth required to execute shell commands.

Chapter 2

Preparation

2.1 Requirements

I needed to create a shell that would feel familiar to an experienced shell user, so my shell would follow existing shell conventions, except where there was a clear benefit from deviating. Any shell that hopes to be useful for a majority of users must be compatible with existing Unix commands as there are simply too many to reimplement all of them for a new design. My shell therefore also needed to support the commands already available on the user's system.

The scripting language that my shell would accept needed to feature a syntax that was reminiscent of today's common shell scripting languages. This included examining the pipe metaphor in some depth and expanding upon it. One common technique is developing a shell command incrementally, whilst only being interested in the output of the final command. I concluded that the shell required the ability to save intermediate results in a lightweight fashion.

I also needed to develop a data structure that would enable my programs to process data in a manner that took advantage of the metadata present. Due to my desire to work with current programs and aiming to maintain useful shell metaphors, I needed to preserve the basic concept of a pipe as far as possible. This meant local commands needed to use the POSIX pipe primitive when pipes were local. Since I wished to support interleaving of my commands and other existing commands, I required a data structure that would be transmitted as structured text.

With my data structure in place I needed a set of commands that would demonstrate the versatility of my design. This set needed to include commands that would produce structured data, commands that could manipulate this data and commands that would offer visualisations of this data.

Finally, I wanted my shell to support a convenient network abstraction. I examined a number of currently available options. It is possible to mount a remote filespace with an 'SSH filesystem', but this requires substantial setting up. PowerShell has some built-in networking but the facility is very general and so the syntax is undesirably complex. I concluded that Oosh should have a simple client/server architecture and a lightweight syntax to go with it.

2.2 Development Approach

Since my design aimed to build on traditional shell design, extending an existing shell was an option worth considering, particularly since most Unix shells are readily available under an open-source license. I investigated building on top of Bash or Fish, but both of these were substantial pieces of C code, much of which would require major changes to work with my intended features. For example, Fish uses a custom parser which is approximately 2,000 lines of C code. There were no clear benefits for the additional effort. I concluded that starting afresh was my best option.

I chose a workflow that emulated programmers whose work I admire. I chose to write my shell in Python, a modern scripting language, used Emacs as my editor and used Git, a distributed version control system to track changes. Later on I needed a parser and lexer so chose Python implementations of Lex and Yacc as these are also well respected tools.

My initial specification was very broad, which enabled me to use a fairly exploratory style of coding. My final grammar is in section 3.3.1 and was developed iteratively after researching the syntax used in Bash and Fish. This grammar was created to provide a useful subset of Fish functionality and be syntactically similar.

Ultimately a mainstream shell is a very large body of code (for example, Bash 4.1 contains over 140,000 lines of C code[1]) and so I could not hope to replicate all this functionality in the time available. My objective thus became to produce a proof-of-concept shell that demonstrated the advantages of my design.

[1]Calculated by running `find . -name '*.[ch]' | xargs wc -l` on the source code of Bash 4.1.

2.3 Software Engineering

2.3.1 Revision Control

Since my project was going to a medium sized piece of code developed over a number of weeks it was clear I needed to use a version control system. I chose to use Git, an industry standard distributed version control system used by major software projects such as the Linux kernel and GNOME.

Rather than running a Git server myself I used GitHub, a Git hosting service that is free for open source projects. In order to qualify for this I needed to pick an appropriate open source license to release Oosh under. I selected the GNU General Public License version 3 [4] (GPL) which is a strong copyleft license so Oosh code may only be used in other GPL licensed software.

The benefits of using Git were so striking that I decided to also place the dissertation TEX source in the repository, which I put under the Creative Commons Attribution license [5] (CC-BY). CC-BY is a weaker copyleft license that permits reuse of the material in any form provided that the author is attributed.

GitHub also provides a web-based front end to the Git repository which is publically viewable. This enabled anyone who was interested in either the code or the write-up of the project to view the very latest version and monitor progress. This resulted in a wide number of people viewing the project, including my supervisor, my director of studies, interested students and prospective employers.

2.3.2 Backup Policy

Using a remote Git server gave me an off-site automated backup system for no additional effort or expenditure. This proved extremely useful when I had a major hardware failure on my primary PC in the later stages of the project. Although it ultimately turned out to only be a motherboard problem, I was able to simply retrieve my work from the server and work on another system.

2.3.3 Development Model

I used the spiral development methodology for producing the Oosh codebase. The features I implemented tended to be sufficiently independent that I was able to write and test each one without being forced to change many other parts of the code. In addition to this I had a final stabilisation stage to fix any bugs that emerged later. These final bug fixes were a result of problems only encountered during benchmarking with larger data sets or unusual feature interactions.

For the dissertation itself I used an approach inspired by agile programming methodology. I used timeboxing to ensure the writing was produced at a steady rate.

2.3.4 Reaching Milestones

Milestones were clearly laid out in my project proposal and so tracking project progress was simple. The coding portion of the project took all the time allotted and also consumed all the slack time. This was largely due to unanticipated dependencies between features, particularly the interpreter itself.

Before I implemented the interpreter I needed something to interact with so I used a simplified prototype. However when I came to replace it with a proper parser and evaluator I used third party tools and so a number of internal interfaces were unsuitable. In the end it became necessary to write the interpreter before the networking features, contrary to my original plan. The interpreter was such a core feature that all other features were much easier to write once the interpreter was in place. I would not have been able to have written the interpreter any earlier as the initial stages were crucial to developing my ideas about what syntax to implement.

When I was half way through the coding phase it became clear that I wasn't reaching my milestones at the times I had hoped to. This was solved by using up the slack time and cutting back on the plan to write an extensive test harness for Oosh.

My plan also did not anticipate bugs discovered whilst I wrote the dissertation. Fortunately I had sufficient time set aside for writing that I was also able to fix these bugs as necessary.

2.4 Summary

The feature set I decided to implement was as follows:

- A read-evaluate-print loop (REPL), the fundamental component of a shell interface.
- A history facility that enables users to re-run earlier commands without typing them in again.
- A language specified in Backus-Naur form that the shell would accept.
- A lexer and parser that accepted this language.

- An evaluator that implemented the semantics of my language.
- A data format specification that was text based, line-oriented and allowed inline metadata.
- A collection of commands that could produce data in this format.
- A collection of commands that could manipulate data in this format.
- A graphing command that could analyse data in this format.
- A pretty print in the shell such that data in this format could be displayed in an attractive, uniform manner.
- A simple networking protocol for running commands on remote systems.
- A standalone server that supported this protocol.
- The ability to use the shell as a client supporting this protocol.
- The ability to use both commands that I had written and other Unix commands on the local system from within the shell.

Chapter 3

Implementation

The final result of my project was the shell client itself, a simple server and a selection of commands that understood the Oosh data structures. I also wrote some simple scripts to demonstrate the power and flexibility of my design.

The remainder of this chapter is broken into sections, with each describing a major feature of Oosh. In each section I also give an example of this feature being used, along with its resulting output. This should clarify how these techniques work in practice.

3.1 Pipes

A major focus of Oosh's design is making the concept of pipes more powerful than the options available on current shells. Oosh generalises the concept of pipes to enable multiple commands to have their output piped into a single input. It also permits convenient pipe saving for later access. Pipes also work over a network transparently to the user.

3.1.1 Pipe Saving

A common way of using today's shells is to iteratively write instructions. Many Unix commands recognise this approach and so print a simple help message if called without any arguments. Oosh commands also follow this convention. A system administrator, trying to fix a KDE application, may build up a Bash command as follows:

```
$ ls /usr/lib
$ diff /backup/20100120/usr/lib /usr/lib
$ diff /backup/20100120/usr/lib /usr/lib | grep libkde
```

In this example he is repeatedly reading `/usr/lib`. Each command rereads from this folder unnecessarily. As the command gets longer he is no longer interested in the earlier processes. The Oosh solution here is to allow him to save this data so that he can access it again later, whilst only worrying about the current filter operations he is performing. In Oosh this would become:

```
$ oosh_ls /usr/lib |1
$ oosh_ls /backup/20100120/usr/lib |2
$ |1+2 oosh_difference |3
$ |3 grep libkde
```

The administrator is now able to reason about previously generated data in a simpler manner. He can also refer back to previously generated results even if he cannot generate the data again. Unlike generating temporary files, he does not need to consider where to save the data and can easily use the saved pipes in a multipipe.

An experienced Bash user could use output redirection to achieve the same effect as the Oosh example. This is discussed in section 4.1. However the Bash redirection syntax is less concise, so the equivalent commands are not as short or as readable. The Bash equivalent of the Oosh example is:

```
$ ls /usr/lib >/tmp/1
$ ls /backup/20100120/usr/lib >/tmp/2
$ diff /tmp/1 /tmp/2 >/tmp/3
$ grep libkde </tmp/3
```

I decided to reuse the pipe character for pipe saving, to clarify that pipe saving is similar to the Unix pipe metaphor and can be used in a similar way.

3.1.2 Multipipes

The traditional pipe metaphor only allows one pipe into a program. A Fish or `sh` user is forced to write each output out to a file somewhere (or use named pipes) if he wishes to use commands such as `cat` or `diff`.

The Oosh approach is rather different. Oosh enables programmers to write programs that take two inputs in order to do more powerful computations. Initially I considered allowing an arbitrary number of inputs but the additional complexity offered no clear benefits so I limited my design to commands that could accept at most two pipes in. Multipipes differ from Bash named pipes in that multipipes use Oosh saved pipes for their input, whilst named pipes do not retain a copy of the data.

Since I modelled my data manipulation on the relational algebra, I implemented `oosh_union`, `oosh_difference` and `oosh_product` based on their corresponding operators. This 'multipipe' design seemed natural for these operators. To demonstrate that this was a generalisation of a pipe I chose a syntax that was very similar to the pipe saving syntax. To use the result in saved pipe 1, the user begins his command with `|1` and to use a multipipe from saved pipes 1 and 2 he begins his command with `|1+2`. An example of this syntax can be seen in figure 3.1.

```
4$ oosh_ps |1; sleep 1m; oosh_ps |2; |1+2 oosh_difference
Command CPU % Memory % PID  User
python3 0.0   0.3      6081 wilfred
sh      0.0   0.0      6082 wilfred
ps      0.0   0.0      6083 wilfred
```

Figure 3.1: An example of sequential commands, pipe saving and multipipes

3.1.3 Pipes Over Networks

A design principle used in Oosh was to examine how computer usage has changed and how today's shells could work better in these different use cases. It quickly became clear that networking is cumbersome and it would be better if pipes worked over networks as if they were local. Although a user may use `rsh` or `ssh` to use pipes over the network, the shell itself is unaware this is going on. Since networking is a part of the Oosh shell there are various optimisation possibilities open that are not possible in Bash.

A Bash user has two options when working with remote systems: he can either mount the remote system on his local system, or use `ssh`. If he mounts the remote system all computation must be performed on his local system. If he uses `ssh` he must either start a shell session remotely and create files to `scp` (the remote copy program) back or use `ssh` to build a pipeline.

The Oosh solution is easier to work with. The user must `connect` to each of the remote systems he wishes to work with, then any commands may be run on any system at the user's discretion without further authentication. For this I chose the simplest syntax I could, requiring the user to only write `mycommand@myserver` to instruct Oosh to run a command on a specific system. As discussed in section 3.5, if the user does not specify a server then Oosh at-

tempts to minimise bandwidth consumption. This syntax is modelled on that of email, which should make Oosh scripts readable to new users.

3.2 Data Structures

Although my initial design specified that I would have structured data pass through pipes between the commands, I still had a variety of implementation possibilities. The data I wanted to manipulate turned out to fit a tabular format well. Therefore as I iteratively developed the data structure, I focused on representations in which the data was divided into rows, with metadata stored inline.

One of my initial ideas was to use an object oriented programming approach, so my first prototype used an object that encapsulated a row. However successive refactorings reduced this object to the point wherein I was able to replace it entirely with Python `dict`s (the mapping structure primitive in Python, which acts as a mutable hash table).

Another design criterion was the data structure had to be passed as text between processes if I wanted to use underlying Unix inter-process communication. This again pushed me towards using standard Python primitives so that I was able to write out a text representation using `__repr__()` and convert it back using `eval()`. Both of these are provided by Python, saving me from having to write any substantial conversion code myself. Using `eval()` introduces the potential for arbitrary code execution. Since Oosh is only a proof of concept program I decided that developing thorough data sanitisation methods was not a focus of the project.

My final design separates lines of structured data with the newline character in the text stream. This enables the interleaving of existing line-oriented Unix commands and Oosh commands whilst still preserving the data format. The examples in figure 3.2 and figure 3.6 demonstrate interleaving. Since all of the metadata is stored within each line, there is no risk of it being lost when we remove lines. Consider the following Bash interaction:

```
$ ps
  PID TTY          TIME CMD
 2068 pts/1    00:00:00 fish
 2189 pts/1    00:00:00 bash
 2204 pts/1    00:00:00 ps
```

Any command which removes the first line (such as `tail`) will remove the column names.

A piece of Oosh conformant data in the final design looks like this:

```
{‘Name’: ‘strawberry’, ‘Colour’: ‘red’, ‘Family’: ‘fruit’}
{‘Family’: ‘vegetable’, ‘Name’: ‘potato’}
```

3.2.1 Pretty Printing

Since traditional Unix commands output data in an unstructured way, output forms vary widely. In contrast, Oosh requires a tabular format, so the same pretty print method can be run each time on all conformant data.

The pretty print functionality I developed is coloured, pads each column to give a perfect alignment and supports heterogeneous lines of data. Rather than simply using tab characters to separate columns, the pretty print finds the longest entry in each column and ensures that the column is exactly one character wider than this entry. This ensures that each column is printed at optimum width for the given data.

The Oosh data format is relatively permissive, permitting columns to be in any order within each row or even omitted completely so supporting this heterogeneity is crucial in order to ensure that every item in each row is printed in a manner consistent with the user's expectations.

The algorithm I designed takes lines of Oosh conformant data as input and finds the optimum layout. This has a time complexity of $O(C \log C + CL)$, where L is the number of lines, and C the number of columns. Scanning through each row in each column position contributes the CL term, which generally dominates. However to ensure the columns are always printed in the same order, Oosh also sorts the columns once, using a quicksort algorithm to give the additional $C \log C$. A thorough speed analysis and stress test is performed in section 4.3.2.

The disadvantage of my design is that it requires that a command entered by the user must have finished being evaluated before the output can be pretty printed and shown to the user. A better solution would be to only pretty print the data currently visible on screen, which would improve latency. If the Oosh shell was rewritten with a `curses`[1] based interface then the shell could learn how much data was visible onscreen and use this better approach.

Furthermore, if we consider the following data set:

```
{‘Name’: ‘Jim’, ‘Age’: 10}
{‘Name’: ‘John’, ‘Age’: 36}
...many more lines...
{‘Name’: ‘Slartibartfast the Magrathean’, ‘Age’: 68}
```

[1] `curses` is a library used for creating full screen interfaces for text tools.

This would always print the name column very wide in Oosh, even if the last line is not visible on screen. A curses interface would work better in this case as it would find the optimum display for that screenful rather than the global optimum.

3.2.2 Data Example

This example was generated by running `oosh_ls src/`.

Raw Data

```
{‘Owner’: ‘wilfred’, ‘Size’: 1013234, ‘Filename’: ‘uucp-1.07.tar.gz’}
{‘Owner’: ‘wilfred’, ‘Size’: 1152997, ‘Filename’: ‘fish-1.23.1.tar.gz’}
{‘Owner’: ‘wilfred’, ‘Size’: 1837, ‘Filename’: ‘smlnj.tar.gz’}
{‘Owner’: ‘wilfred’, ‘Size’: 6598300, ‘Filename’: ‘bash-4.1.tar.gz’}
{‘Owner’: ‘wilfred’, ‘Size’: 196529, ‘Filename’: ‘xwrits-2.26.tar.gz’}
{‘Filename’: ‘plasma-weather-0.4.tar.gz’, ‘Size’: 2480099}
```

Note we have rudimentary types here, distinguishing between strings and integers. The sort command in Oosh takes advantage of this.

Pretty Print Output

```
Filename                    Owner   Size
uucp-1.07.tar.gz            wilfred 1013234
fish-1.23.1.tar.gz          wilfred 1152997
smlnj.tar.gz                wilfred 1837
bash-4.1.tar.gz             wilfred 6598300
xwrits-2.26.tar.gz          wilfred 196529
plasma-weather-0.4.tar.gz   -       2480099
```

3.3 Syntax

When I developed the first parts of Oosh I did not specify a grammar and simply modified the syntax that the prototype accepted as I went along. When I implemented a full interpreter I formalised the syntax that I had incrementally developed in the initial prototype. This approach enabled me to experiment without having to radically restructure the code at each iteration.

Oosh supports conditionals based on process return codes, two different types of loops, string variables, pipe variables (which I refer to as ‘saved pipes’) and

command pipelines. This is a smaller feature set compared with more mature offerings such as Bash and Fish. However it is sufficient for basic shell interaction, and the parsing and evaluation code has been written in a sufficiently flexible manner that any desired expansion could be added with minimal difficulty.

The design for variables deserves closer attention. The Oosh design can be seen as two variable namespaces, one that holds strings and the other that saves the output of pipes. Saved pipes may be emulated in bash by executing `command1 >/tmp/foo` and then later `command2 </tmp/foo`. However this approach requires the user to explicitly create temporary files and does not conveniently generalise to network access, which requires `scp` as well. Saved pipes are therefore only a convenience for the user, not a completely novel feature.

Other notable omissions from the final syntax include subshells, arithmetic operations, function definitions and array variables. This turns out to rarely be a problem for the user, since subshells can be emulated with pipe saving. Arithmetic operations may be performed using the Unix `bc` command, which is arguably a more Unix style design (conforming to the "do one thing and do it well" design mantra). The remaining omissions are more difficult to work around but fall outside of the usage targeted by Oosh.

3.3.1 Formal Language Specification

I list below the language that the parser accepts. Non-terminals are marked with angle brackets, terminals that are reserved words are in a `monospaced` font and other terminals are written in capitals.

⟨commands⟩ ::= ⟨commands⟩; ⟨commands⟩
| ⟨commands⟩ NAMEDPIPE
| ⟨while⟩ | ⟨if⟩ | ⟨assign⟩ | ⟨for⟩ | ⟨command⟩ | ϵ

⟨for⟩ ::= `for` STRING `in` ⟨values⟩`;` `do` ⟨commands⟩`;` `end`

⟨while⟩ ::= `while` ⟨command⟩`;` `do` ⟨commands⟩`;` `end`

⟨if⟩ ::= `if` ⟨command⟩`;` `do` ⟨commands⟩`;` `end`
| `if` ⟨command⟩`;` `do` ⟨commands⟩`;` `else` `do` ⟨commands⟩`;` `end`

⟨assign⟩ ::= `set` STRING ⟨value⟩

⟨command⟩ ::= ⟨simplecommand⟩
| NAMEDPIPE ⟨simplecommand⟩
| MULTIPIPE ⟨multicommand⟩ PIPE ⟨simplecommand⟩
| MULTIPIPE ⟨multicommand⟩

⟨simplecommand⟩ ::= ⟨values⟩ | PIPE ⟨simplecommand⟩

⟨multicommand⟩ ::= ⟨values⟩

⟨values⟩ ::= ⟨value⟩ ⟨values⟩ | ⟨value⟩

⟨value⟩ ::= STRING | VARIABLE | QUOTEDSTRING

3.3.2 Simple Pipeline Syntax

I performed an examination of an online command-line snippets repository [7] and the official Bash reference manual [8]. I concluded that the pipeline is the most frequently used and earliest taught shell construct. The Oosh pipeline syntax should be immediately familiar to a shell user, without any different syntax required for structured data or unstructured data. It is so similar that many simple shell invocations will work without modification in Oosh. Figure 3.2 gives an example of this feature in use.

```
1$ oosh_ls /usr/bin | oosh_project Filename Size | oosh_sort Size
 | tail
Filename Size
mencoder 10984340
inkview  11074508
inkscape 11100236
mplayer  11758932
bibtexu  17312116
xetex    17799216
xelatex  17799216
skype    18567888
```

Figure 3.2: A simple pipeline showing contextual data manipulation. Note that `tail` is a part of the GNU coreutils package and is working in Oosh without modification.

3.3.3 Control Structures

Control structures vary quite widely between shell offerings. One perceived weakness of Bash is that blocks are closed using different keywords depending on the control structure used. For example, a `for` loop syntax ends with `end`, an `if` block ends with `fi` and a function declaration ends with }. Fish fixes this weak-

ness (at the expense of POSIX compliance) by always using `end`. I felt that this made writing shell scripts easier so the Oosh syntax is very similar to Fish here.

Figure 3.3 shows an example using a `for` loop.

```
1$ for x in 192.33.4.12 128.8.10.90; do host $x; end
12.4.33.192.in-addr.arpa domain name pointer c.root-servers.net.
90.10.8.128.in-addr.arpa domain name pointer d.root-servers.net.
```

Figure 3.3: An example `for` loop in Oosh, demonstrating variable assignment and unstructured data

3.4 The Shell

The Oosh client is the largest single piece of code within my project and can function without a server as a standalone shell. Basic command line interaction is provided by the `cmd.Cmd` object in the standard Python library. This meant the shell supported basic user interaction and command history from very early on, allowing me to test new features within the shell rather than testing new functions in isolation. I was then able to add functionality and complexity iteratively, with sufficient independence between features that new features generally didn't stop previously developed code from working.

The shell itself takes input from the user on a line-by-line basis. Each line is lexed, parsed and the resulting parse tree is evaluated. For this job I took advantage of PLY (Python Lex-Yacc), which is a reimplementation of Lex and Yacc in Python but offers more helpful error messages. Once the syntax of language was finalised, using the library was a simple matter of writing regular expressions for lexing and converting my grammar specification to a format that Yacc accepted. Evaluation of an expression is then a relatively simple recursive descent tree evaluator.

During development it quickly became clear that I needed to use the pipe functionality provided by the operating system to allow multiple processes to communicate in the standard shell manner. However this native pipe is an interface offered by the kernel on the local system and consequently cannot be used for pipes running over the network. To get round this I developed a `PipePointer` abstraction that enables the evaluator to treat all Oosh pipes in the same fashion. The `PipePointer` is an object that either encapsulates a native pipe address or the location of a remote process. In the case of remote data the `PipePointer`

handles fetching the data and decoding it appropriately. This was the central element of my 'lazy data fetching' approach, enabling Oosh to only fetch the data from the remote system at the last moment.

3.5 Networking

The networking functionality was designed with two intentions: a concise, convenient syntax and efficient bandwidth usage. Today a user can use Bash and SSH to run commands remotely, but the user must explicitly mark every command he wishes to run remotely.

```
home$ ssh wilfred@foo.com ps | grep root
```

Figure 3.4: Collecting data on one remote system and performing analysis on a different remote system using Bash

```
$ connect@remote.com wilfred password
$ ps@remote.com | grep@localhost root
```

Figure 3.5: Collecting data on one remote system and performing analysis on a different remote system using Oosh

An experienced Bash user may observe that `grep` will reduce the amount of data and may be safely moved, so would instead use the command `ssh wilfred@foo.com "ps | grep root"`. However Oosh makes specifying which system the command will be run on optional, with the shell itself performing this optimisation.

To create this functionality I firstly extended the Oosh syntax to allow this simple location syntax using a hook in the evaluator (which is why it is not shown in the grammar specification). I then required a server/client architecture that would enable the user to run remote commands from his Oosh client as if they were local.

3.5.1 The Server

As a proof-of-concept, only single user interaction was necessary for my server. I chose to use TCP sessions, but since the server does not handle multiple users I

was able to avoid using threading within the server itself. The user authenticates himself to the server and is then able to issue commands to run on the server. Whilst I did not do a thorough security evaluation, I attempted to use best practices and so I used a salted SHA1 hash for password storage.

Once the user has authenticated himself to the servers he wishes to use, he can specify (on the client) the location where he wishes each command to be run. The client will automatically initiate commands on the server and requests the resulting data as late as possible. This enables the server to use native pipes locally.

The server supports the following commands: `connect`, to log in, `disconnect`, to log out, `send`, to send the results of the last command back to the client, `receive`, to send data to the server for it to use as `stdin` for the next command, and `command`, to execute a shell command on the server.

3.5.2 The Client

The client initiates a connection on the Oosh port, 12345, then authenticates and begins sending `command` strings for the server to execute. The server runs these commands with the privileges of the user who started the server.

Each command sent to the server corresponds to a new process to be run on the remote system (referred to as a ⟨simplecommand⟩ in the grammar), so if the user enters the command `a@remote | b@remote | c@remote` this will become three separate messages to the server, `command a`, `command b`, `command c` with the establishment of the pipes between them being implicit. Therefore a loop (since loops are always evaluated by the client) where every ⟨simplecommand⟩ in the body is specified to be run remotely will result in every ⟨simplecommand⟩ producing a `command` message to be sent to the server. This is not a performance problem if it is the case that the amount of data sent between processes vastly exceeds the number of commands that need sending.

Since the Oosh server only sends the results of computation back on demand, it enabled me to use a lazy data approach. Again if we consider `a@remote | b@remote | c@remote`, only the result of the computation after `c` has terminated will be sent back to the client. This optimisation is transparent to the user (assuming the network has sufficiently low latency) and does not affect the results.

A more interesting optimisation in Oosh is that of 'command moving'. If the user runs the command `oosh_ps@remote | oosh_project Command PID` the `oosh_project` command acts as a filter and is location independent. Oosh recognises these commands (using a simple whitelist) and runs them on the same system as the previous command to reduce bandwidth consumption. This is safe

```
Client:
2$ connect@193.60.95.74 wilfred mypassword;
     oosh_ls@193.60.95.74 /home/wilfred/logs | grep kde
Filename               Owner   Size
freenode_#kde-i18n.log wilfred 879
freenode_#kde.log      wilfred 53318

Server:
$ ./ooshserver.py
Starting server at soup.linux.pwf.cam.ac.uk (193.60.95.74) on port
 12345
request is [‘connect’, ‘wilfred’, ‘mypassword’]
request is [‘command’, ‘python3’, ‘oosh_ls.py’,
 ‘/home/wilfred/logs’]
request is [‘command’, ‘grep’, ‘kde’]
request is [‘send’]
```

Figure 3.6: Example of commands being run over a network with command moving. Note that `grep` has not had its location specified and so runs remotely.

(it does not affect the results) and is transparent to the user. The user may however explicitly state where a command is to be run and Oosh will respect that. The actual logic to decide whether or not to move the command is within the client, so this added no additional complexity to the client/server interaction.

This ‘command moving’ optimisation assumes that the bandwidth to the server is limited and the server has sufficient resources to run the additional commands. A common usage of Bash with `ssh` is that of running the remote server on a desktop or on a large server system (given that two of the largest server vendors, IBM and Cisco, ship servers with SSH daemons) so this assumption is probably valid.

Note that `grep` is run on the server despite the user not specifying the host. Also note that the output of `oosh_ls` is not sent immediately, the client explicitly requests it later on.

3.6 Command Selection

I needed to choose a selection of programs that would make a useful subset of basic command line functionality. I examined the selection of commands offered by BusyBox [6], as it aims to be a fairly minimal set of Unix commands required to make a system useful. Many commands were either interactive, set system settings or started daemons. None of these types of programs sent sufficient data to `stdout` for them to benefit from my new structured data approach. Ultimately since my shell supported normal Unix commands, it was unnecessary for me to replicate a lot of functionality that existing tools provide.

In addition to the shell with its built-ins, I needed a selection of programs that understood the structured data and demonstrated its flexibility and versatility. Clearly with data in a tabular format, a set of commands that offer database-like functions would be both familiar to the user and broadly applicable. Another advantage of structured data is the ability to draw graphs, so an automatic graphing facility was created.

There were three types of commands that I implemented. I implemented data sources, which write data to `stdout` in the Oosh data format. I also implemented data manipulators, which perform a variety of database-like manipulations. By implementing a minimal subset of the relational calculus I was able to produce tools whose expressive power was relationally complete. I also implemented a data analyser that would draw graphs based on input data given in the Oosh data format.

3.6.1 Data Sources

I wrote `oosh_ls`, `oosh_echo` and `oosh_ps` based on their standard Unix counterparts. These Oosh commands produce data that conforms to my data format and produce interesting, useful data.

My implementations are simpler than their Unix counterparts, offering no configurability in terms of output format or quantity of information returned. These output format flags on Unix vary because the user is forced to choose between having human-readable data or data which is easy to manipulate with other commands. For example, `ps` has the `-o` option which permits a wide range of formats. This compromise is unnecessary with the Oosh data structure which is both machine-readable and pretty-printed by the shell.

I also opt for outputting more information from my data sources than may be necessary. `oosh_project` enables the user to easily discard unwanted data

columns so this is not a problem. This approach has been also used by the designers of PowerShell.

3.6.2 Data Manipulators

Since my data is in a tabular format, I approached data manipulation using database style operators. Rather than using SQL style operators (whose syntax differs substantially from that of Oosh) I looked at the relational algebra and based my operator selection on it.

I wrote `oosh_select` for selection, `oosh_project` for projection, `oosh_product` for the Cartesian product, `oosh_union` for set union, `oosh_difference` for set difference and `oosh_rename` for attribute renaming. This is the minimal subset of the relational algebra needed to give the user full expressive power. I also added `oosh_sort`, a sort operator, as this is a common shell task that benefits substantially from structured data.

3.6.3 Data Analyser

The pretty print I wrote is a great improvement over the simplistic line-by-line printing used by traditional shell commands, giving the user a consistent view of his data. Further, once we have pervasive metadata being produced, we can perform more sophisticated analyses of the data. I wrote a graphing utility that takes full advantage of this structure.

My grapher, `oosh_graph`, simply requires the user to name the data series he wishes to plot and the type of graph he wants. `oosh_graph` then extracts the specified columns and passes them to a graphing library. This is not impossible in Bash, but Oosh enables a more elegant approach since the user can describe the data in terms of data labels. An example of the grapher in use can be seen in figure 3.7

3.7 Backward Compatibility

Unix systems already have a very large number of command line tools. The PWF offers over 4,000 commands[3] to its users. Replacing all of these with Oosh equivalents would be intractable (within the timeframe of the project) and many applications would not benefit from the Oosh design approach. I therefore created the facility for any Unix command to be run from within Oosh.

[3]Calculated by pressing tab twice in an SSH session on the PWF. This is unlikely to be very accurate, since some commands offer duplicate functionality and some programs (such as

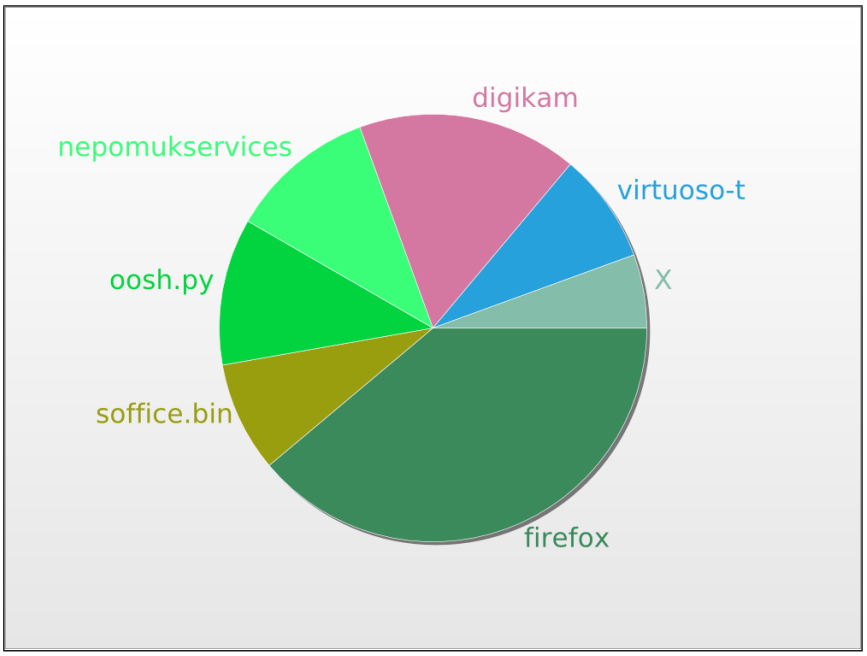

Figure 3.7: A pie chart generated with `oosh_ps | oosh_graph pie Command 'CPU %' cpu_consumption.svg`

I implemented this using a simple namespace style design. All Oosh commands start with the prefix `oosh_`. This enabled me to reuse command names such as `ls` without clashes. The Oosh shell recognises the `oosh_` prefix and searches in the Oosh directory rather than the directories listed in the `$PATH` variable. This namespace scheme permits interleaving of Oosh commands and other Unix commands. Since the Oosh data structure is line oriented, this interleaving ability is frequently useful. Commands such as `grep` filter on a line-by-line basis, resulting in the Oosh data structure being preserved even when being operated on by non-Oosh line-oriented tools without modification.

In figure 3.6 we see `grep` being used without modification but still producing valid Oosh data structures so the data may be pretty printed.

3.8 Problems Overcome

The largest problem that I faced during coding was dealing with the limited specification I started with. I could not implement a scripting language for Oosh until I had a grammar, which required careful and subjective judgements.

BusyBox) provide multiple commands in one binary.

My language choice of Python proved to be a good decision due to ease of coding, but was not without problems. Choosing the most recent version (Python 3) seemed to be logical but I was unaware that many Python libraries are not yet compatible with it, which limited the number of third-party libraries I could use. For my graphing command, the library I wished to use was only Python 2 compatible. Since each command is spawned as a separate process, I was able to use Python 2 for my grapher without changing the rest of my codebase by hard-coding this exception. This was not ideal since my resulting codebase became heterogeneous.

Python's documentation is generally good, but I did encounter bugs. I worked around these by examining the source of the libraries I was using and establishing the correct way to use them.

Unix shells have been around for so long that they have acquired a lot of tradition and legacy behaviour. Some of these legacy traits caused problems for me, particularly escape codes. Today's shells support a wide range escape codes which are not used by Oosh. Sadly as they are non-printable characters they presented an interesting challenge for debugging. Certain escape characters crashed Oosh but could not be printed the console for me to examine.

The other limitation I encountered was Python's handling of signals, which is limited. Spawning subprocesses requires the ability to emit and receive a number of interprocess signals, and the Python base libraries do not support `SIGPIPE`. I was only partly able to work around this limitation, and there still exist corner cases where Oosh crashes on large inputs. The correct way to fix it would be to replace a number of library functions with equivalents that handle this signal, but I concluded this was too time consuming for a rare bug.

Chapter 4

Evaluation

To evaluate the expressive power of commands within Oosh, I have prepared a selection of character count comparisons. I also include some data on the number of processes spawned between a typical Bash command and a typical Oosh command.

I have also collected a set of networking benchmarks, which measure the effectiveness of my bandwidth optimisations versus a naïve implementation. I measure both networking overhead and scaling.

Finally, I also include some simple performance benchmarks based on system resource consumption and latency from the user's perspective.

4.1 Command Length

Since Oosh uses structured data with inline metadata, it should produce more concise commands, since no intermediate data formatting is necessary to use the output of one tool as the input for another. The labelling of data offered by Oosh data format allows us to refer to column names. This should enable the user to write scripts that are easier to read.

I conducted a series of tests that compared the length of Oosh commands and their Bash equivalents. Defining a typical set of commands is extremely difficult so instead I considered a set of commands in which the advantages of Oosh should be particularly apparent.

I compare Oosh commands with Bash commands by length. In addition to simple character count I compare the Halstead length [10] as a simple measure of complexity. The Halstead length is defined as the sum of the number of operators and operands and so avoids penalising conceptually simpler commands that happen to have longer names or arguments.

	Length	
Command sequence	Characters	Halstead

`ls /bin >/tmp/bin1.txt`
`ls /usr/bin >/tmp/bin2.txt`
`cat /tmp/bin1.txt /tmp/bin2.txt | grep python`
	96	14

`oosh_ls /bin |1`
`oosh_ls /usr/bin |2`
`|1+2 oosh_union | grep python`
	66	13

Figure 4.1: Comparing standard Unix tools with shell redirection versus pipe saving

In the example shown in figure 4.1, Oosh produces a character saving despite command names being longer. The Halstead length is almost unchanged, so Oosh is not producing a substantial difference in command simplicity for the user. The pipe redirection in Bash requires the user to choose a name for temporary data, an unnecessary overhead. The character count reduction therefore corresponds to a modest decrease in complexity for the user.

The Bash example gives the closest analogue of the Oosh command. A number of alternatives are possible. One popular feature of Bash is regular expression support, so a Bash user may write the last line as `cat /tmp/bin[12].txt | grep python`, but this decreases the character count at the expense of increasing the Halstead length. Bash also supports named pipes, so the whole command may be simplified to `cat <(ls /bin) <(ls /usr/bin) | grep python`. This is both shorter and simpler but does not give precisely the same outcome, as we cannot reuse the results from `ls /bin` or `ls /usr/bin` unlike the Oosh version which saves these results.

	Length	
Command sequence	Characters	Halstead
`ps aux | tail -n +2 | sort -k 3,3 | \` `tr -s ‘ ’ | cut -d " " -f 1,3,11-`	70	25
`oosh_ps | oosh_sort ‘CPU%’ |` `oosh_project User Command ‘CPU %’`	64	9

Figure 4.2: Comparing data filtering between Bash and Oosh

The example in figure 4.2, which uses filtering based on tabular data, benefits hugely from the Oosh design. The Oosh example uses slightly fewer characters but its Halstead length is dramatically shorter. This example also slightly favours

`oosh_ps` since we have to pass an argument to `ps` to achieve the same output, but even discounting for this the Bash command is substantially more complex.

The Bash example works as follows: list all processes running, remove the header, sort on column 3, convert all whitespace blocks to single spaces then finally cut out columns 1, 3 and 11 onwards. Since the header is just another line of text to `sort`, we must remove it whilst `oosh_sort` does not require this input massaging. `cut` is used to only split the line based on spacing, so we must use `tr` to ensure spacing is uniform. Even though `ps` does primitive formatting, we are forced to remove it.

An experienced Bash user might be tempted to use `sed` and `awk` to solve this problem, which are scripting languages in their own right. However, we cannot distinguish between spaces that denote column separation and commands which include spaces (such as '`uname -a`'). This forces us to use a loop in `awk`. The resulting command is therefore:

```
ps aux | sed 1d | sort -k 3,3 |
  awk '{printf("%s %s ",$1,$3);for(i=11;i<=NF;i++)
    printf("%s ",$i);printf("\n")}'
```

which is both longer and more complex.

The metrics here do not capture another simplification offered by Oosh's high level data view. The Oosh example enables the user to refer to column by name rather than the incidental position. "Sort on the CPU % column" is more readable than "sort on column 3" despite the Halstead length being identical. A thorough user test would be required to measure the benefit of this.

4.2 Network Benchmarks

To measure the effectiveness of my networking design choices, I compared the following Oosh command with its equivalent using Bash and SSH. Whilst SSH is a substantially more complex piece of software, it is the most popular remote shell utility [11] and so acts as the major 'competitor' to Oosh.

Bash Session

```
wilfred@alto:src:1$ ssh wilfred@127.0.0.1 ls test_data/100 | grep 0
wilfred@127.0.0.1's password:
10.txt
100.txt
```

```
20.txt
30.txt
40.txt
50.txt
60.txt
70.txt
80.txt
90.txt
```

Oosh Session

```
1$ connect@127.0.0.1 wilfred mypassword
2$ ls@127.0.0.1 test_data/100 | grep 0
10.txt
100.txt
20.txt
30.txt
40.txt
50.txt
60.txt
70.txt
80.txt
90.txt
3$ disconnect@127.0.0.1
```

4.2.1 Design Differences

The Oosh client/server protocol is simple, sending messages as text between the client and the server. SSH is more complex, sending most data in a binary format. SSH also ensure authenticity of the server to prevent man-in-the-middle attacks.

SSH encrypts all traffic, increasing CPU consumption although not meaningfully changing bandwidth consumption. Oosh sends data in plaintext.

4.2.2 Bandwidth Usage

Using Wireshark, a popular network traffic sniffer, I recorded all network traffic for Bash with SSH, Oosh with command moving, and Oosh without command moving. This enabled me to examine both the number of packets and total amount of data transmitted in each test.

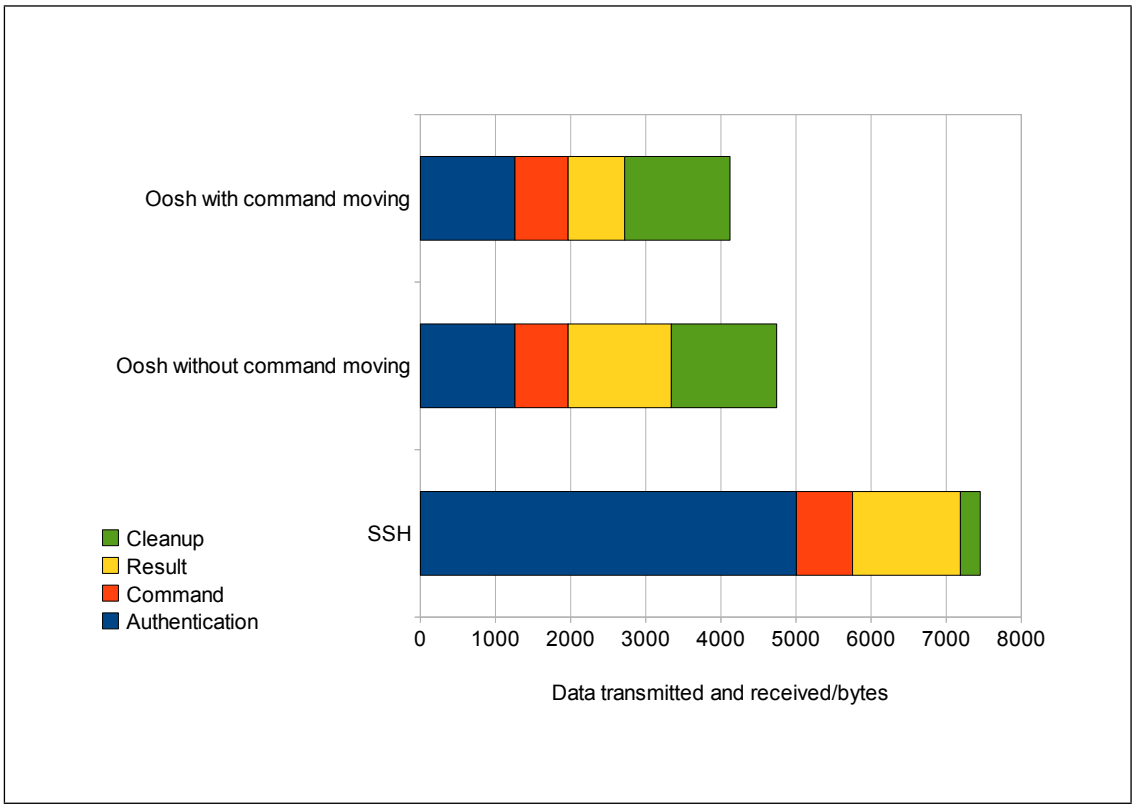

Figure 4.3: Contrasting bandwidth usage between Bash with SSH and Oosh, and measuring the effect of command moving

Figure 4.3 gives a side-by-side comparison of total data sent for each of the three tests. The Bash with SSH test demonstrates the additional complexity of the SSH authentication protocol. This would be exacerbated if the user was interacting with a larger number of servers. However this is a one-off cost and more substantial commands running remotely would ensure the data returned would be much larger than the authentication.

In the Oosh tests shown on the graph, there is noticeably less traffic at the authentication stage due to the simpler protocol. Using the command moving optimisation reduced the amount of data sent back from 1373 bytes (still slightly less than the 1436 bytes of SSH) to 698 bytes. However moving `grep` to the remote system incurred a cost (not shown on the graph) of 698 additional bytes for the second command. Command moving in this example therefore only provides a very modest bandwidth saving so savings are highly dependent on usage.

There is also a marked contrast between the 264 bytes required to close the connection with SSH and the 1403 bytes required by Oosh. This is partly due to Oosh requiring an explicit disconnection message, but examining the traffic

logs shows that Oosh initiates too many TCP sessions. When the client initiates a session to the server the server also starts a separate session to the client. This is clearly wasteful because a single TCP connection supports duplex communication. Much of the Oosh bandwidth is therefore wasted on TCP setup and teardown. The 1403 bytes consisted of 20 packets, much larger than SSH which only used 4. Consequently the average packet size during Oosh cleanup was only 70 bytes. Since an empty TCP packet is 66 bytes, clearly most of the 1403 bytes are superfluous.

Other optimisations become visible here as well. Oosh server commands are sent as strings, and verbose messages reporting errors or lack thereof are sent back as strings, even though successful execution tends to be the most common outcome. The Oosh protocol is too chatty and a shorter encoding for commands would also improve efficiency.

4.3 Oosh Overheads

I also conducted a series of benchmarks that measure the performance overhead of using Oosh instead of standard tools.

4.3.1 Overheads Due to Oosh Design Choices

My first test compared running `oosh_ls` with `ls` 6.12 from GNU coreutils on a collection of directories with contents of varying size. I generated these directories using the Bash command `for x in {1..n}; do echo -n ‘‘’’ >$x.txt; done` to ensure that all the files were identical. Figure 4.4 shows the performance difference between the two.

This graph demonstrates that there is very little performance overhead when using Oosh for large datasets. At the lower end, Oosh commands have a startup time of approximately 0.02 seconds which can be attributed to the startup time of the Python interpreter. This difference is below the level of human perception, although pathological cases probably exist, such as an Oosh script with many short lived processes. Humans cannot perceive delays of less than 0.1 seconds [9] so this delay is acceptable. Both `ls` and `oosh_ls` can be seen to be scaling linearly with respect to the number of files.

Testing the scaling of `oosh_ls` allowed me to analyse the costs introduced by my data structure. If we consider a sample piece of output from `oosh_ls` being run on the root directory, we obtain figure 4.5.

Data format efficiency was not a focus during the project, and as a result the Oosh data structure contains some redundancies. One obvious example is that

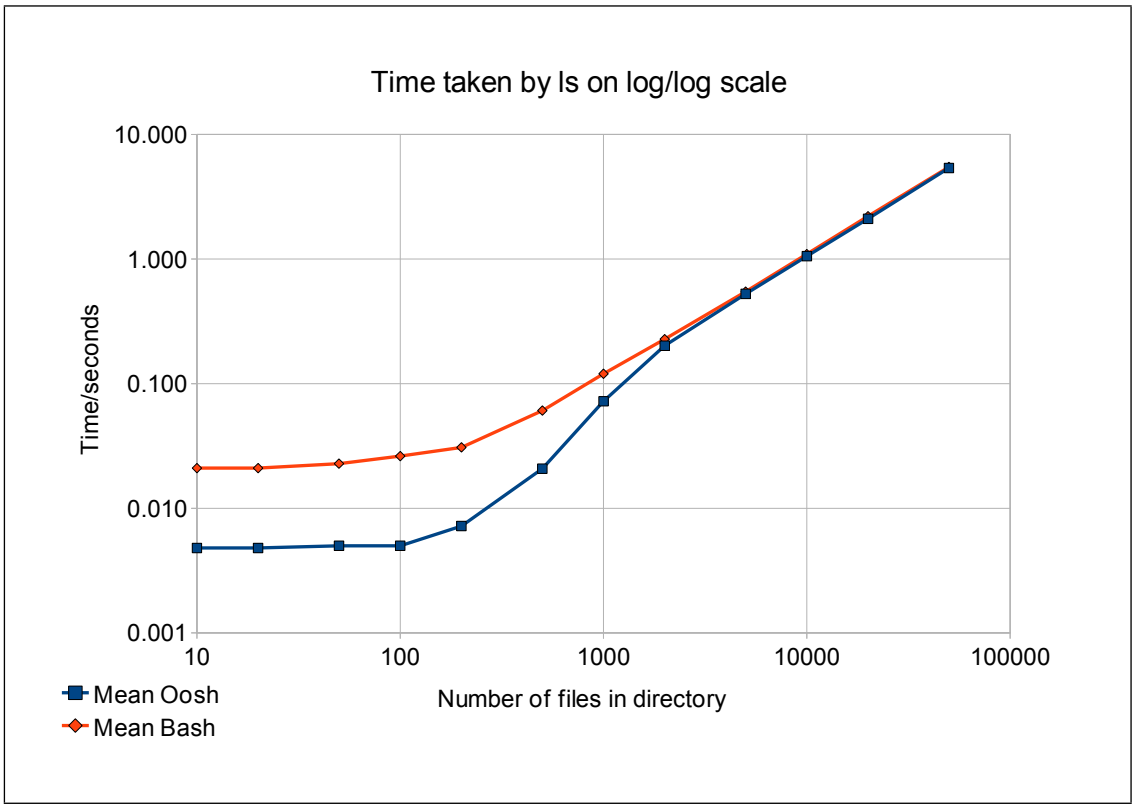

Figure 4.4: Contrasting performance between `ls` from GNU Coreutils and `oosh_ls`

of using curly brackets to mark the beginning and end of lines when they are also separated by newline characters. However shell data is typically sent locally via local inter-process communication which is sufficiently high bandwidth that we can afford the additional overhead of Oosh structured data. To minimise data size I could potentially use compression on each line, but as discussed in section 4.3.3, CPU consumption is more of a concern with Oosh so introducing compression would be unlikely to be a useful optimisation.

4.3.2 Latencies Introduced by Pretty Printing

As discussed in section 3.2.1, my pretty printing algorithm has complexity $O(C \log C + CL)$ in time, where C is the number of columns and L the number of lines. Using the assumption that a typical workload will have vastly more lines than columns in the output, I measured the additional time taken for output to be shown to the user if my pretty print algorithm is used.

This graph was produced using the same set of directories as in the previ-

```
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘root’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘media’}
{‘Owner’: ‘root’, ‘Size’: 0, ‘Filename’: ‘.servers’}
{‘Owner’: ‘root’, ‘Size’: 0, ‘Filename’: ‘.ux’}
{‘Owner’: ‘root’, ‘Size’: 3072, ‘Filename’: ‘boot’}
{‘Owner’: ‘wrah2’, ‘Size’: 512, ‘Filename’: ‘apps’}
{‘Owner’: ‘root’, ‘Size’: 12288, ‘Filename’: ‘etc’}
{‘Owner’: ‘root’, ‘Size’: 4540, ‘Filename’: ‘dev’}
{‘Owner’: ‘root’, ‘Size’: 0, ‘Filename’: ‘proc’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘opt’}
{‘Owner’: ‘root’, ‘Size’: 0, ‘Filename’: ‘servers’}
{‘Owner’: ‘root’, ‘Size’: 36864, ‘Filename’: ‘tmp’}
{‘Owner’: ‘root’, ‘Size’: 12288, ‘Filename’: ‘authcon’}
{‘Owner’: ‘root’, ‘Size’: 12288, ‘Filename’: ‘sbin’}
{‘Owner’: ‘root’, ‘Size’: 0, ‘Filename’: ‘sys’}
{‘Owner’: ‘root’, ‘Size’: 12288, ‘Filename’: ‘lib’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘bin’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘usr’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘var’}
{‘Owner’: ‘wrah2’, ‘Size’: 512, ‘Filename’: ‘ux’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘srv’}
{‘Owner’: ‘root’, ‘Size’: 4096, ‘Filename’: ‘mnt’}
{‘Owner’: ‘root’, ‘Size’: 16384, ‘Filename’: ‘lost+found’}
{‘Owner’: ‘root’, ‘Size’: 12288, ‘Filename’: ‘home’}
```

Figure 4.5: Output of `oosh_ls /` without pretty printing

ous benchmark, and similarly using `oosh_ls`. I then compared the time for the command to terminate with and without pretty printing.

This graph conceals the fact that the pretty print algorithm requires the full input before producing any output, whereas the simplistic Bash-style approach of writing output asynchronously induces no latency. The algorithm performs as expected, scaling roughly linearly as the number of lines increases. Since printing 1000 lines introduces only 33 milliseconds of delay (on top of the 93 milliseconds required by the rest of the command execution), the performance of Oosh's pretty print is completely acceptable for small or medium sized quantities of data.

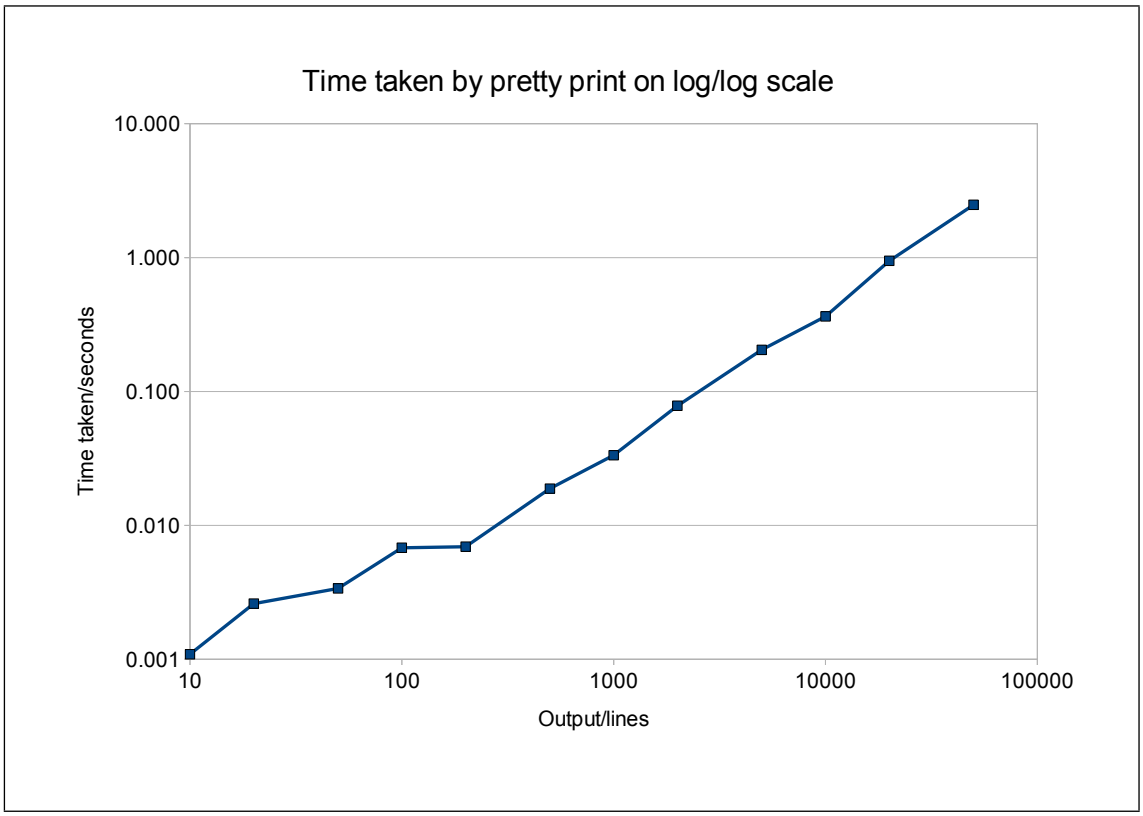

Figure 4.6: Added latency due to running the pretty printing algorithm on Oosh output

4.3.3 Other Measures

I also performed a few simpler measures of resource utilisation. Whilst performing the pretty print benchmarks on the most recent build of Oosh, I used the Linux resource monitor to measure peak system usage. Memory usage peaked at 43 MiB, and CPU usage at 40% (on an Athlon 64 X2 6000 clocked at 3 GHz). By contrast, Bash did not exceed 2 MiB of memory and 5% CPU. The memory consumption is acceptable on modern systems which tend to have at least 2 GiB of memory. However CPU usage is rather high and since many experienced shell users often have a number of shells open at once, this may become a problem.

Chapter 5

Conclusions

5.1 Overall Conclusions

Developing Oosh demonstrated that it is possible to produce a more effective shell metaphor provided that you only consider a limited subset of shell functionality. The final product supports much of standard Unix shell functionality plus a number of novel features.

My structured data approach produced tangible benefits and making networking part of the shell's built-ins produced new scope for optimisations. Oosh is capable of dynamically determining whether it is safe to move commands and transparently performing the move.

My graphing tools and data analysers offered particular clear benefits from using structured data, and I would like to see these capabilities incorporated into mainstream shells in the future.

As a standalone shell Oosh is not yet capable of replacing other popular shells but as a proof of concept I believe it clearly displays the potential of the Oosh design decisions.

5.2 Potential Future Enhancements

Throughout the project I found many features in common shells that are widely used but were outside the scope of Oosh. However to use Oosh as your primary shell would be very inconvenient without these.

One obvious enhancement would be pervasive use of the Oosh data structure. There are a number of areas that would benefit from this. One good example of this is the `/sys` directory which holds kernel generated hardware information.

The precise format of the Oosh data structure was not chosen for language independent manipulation. The Oosh data structure would benefit from using a data format for which many tools exists, such as XML or JSON (the format used currently is very nearly valid JSON).

Other missing luxuries include line continuations, regular expressions, subshells ('backticks' in `sh`) and tab completion. Tab completion in Oosh could be substantially more powerful than that of Bash, since commands can know column names.

I also did not take advantage of any of the optimising flags or profiling tools within Python, so there probably is some potential for improvement in Oosh's performance. My benchmarks showed that Oosh performance was generally not much worse than that of Bash but large workloads may well benefit from some careful optimisation.

Bibliography

[1] Louis Pouzin *Multics: The Origin of the Shell* `http://www.multicians.org/shell.html`

[2] Tom Duff *Rc – A Shell for Plan 9 and UNIX systems* `http://doc.cat-v.org/plan_9/4th_edition/papers/rc`

[3] Axel Liljencrantz *Design Document* `http://fishshell.org/user_doc/design.html`

[4] Free Software Foundation *The GNU General Public License* `http://www.gnu.org/licenses/gpl.html`

[5] Creative Commons *Creative Commons Attribution 3.0 Unported License* `http://creativecommons.org/licenses/by/3.0/`

[6] Denys Vlasenko *BusyBox: The Swiss Army Knife of Embedded Linux* `http://www.busybox.net/about.html`

[7] David Winterbottom *commandlinefu.com is the place record those command-line gems that you return to again and again* `http://www.commandlinefu.com`

[8] Free Software Foundation *GNU bash manual* `http://www.gnu.org/software/bash/manual/`

[9] S. K. Card, T. P. Moran and A. Newell. *The Psychology of Human-Computer Interaction*

[10] Halstead, Maurice H. *Elements of Software Science* Elsevier North-Holland, New York

[11] Nicholas Rosasco and David Larochelle *How and Why More Secure Technologies Succeed in Legacy Markets: Lessons from the Success of SSH* Department of Computer Science, University of Virginia

Appendix A

Project Proposal

Wilfred Hughes
Churchill
wrah2

Part II Computer Science Project Proposal

oosh: An object oriented shell

May 12, 2010

Project Originator: Wilfred Hughes

Resources Required: See attached Project Resource Form

Project Supervisor: David Eyers

Signature:

Director of Studies: John Fawcett

Signature:

Overseers: Jon Crowcroft and Pietro Lio

Signatures:

Introduction and Description of the Work

Unix shells offer powerful ways of working with data and files using a command line interface. Traditionally they were only a simplified interface to the kernel, passing unstructured text streams between programs.

I intend to extend this idea to an 'object stream' where the data passed between programs contains semantic structure and metadata. This will enable powerful data manipulation and processing that is not possible with a traditional shell.

Due to the age of the Unix shell design (with many design decisions unchanged from the original Unix shell circa 1971 [1]), the design is simple since the computer systems it first ran on offered limited power. A traditional shell always copies data to the next program in the pipe.

My design will modify the basic shell design so that programs are presented with file pointers that point to local or remote files. The shell may then dynamically decide which host to run each part of the pipeline. The addition of file pointers enables me to distinguish destructive and non-destructive operations, permitting the shell to cache non-destructive pipeline operations to improve responsiveness.

Resources Required

I plan to perform most of my work using my personal computer. The code will be stored in a remote git repository so I need an Internet connection to push my changes there regularly. If necessary I can move to any Internet connected Unix system that offers a Python interpreter.

Starting Point

My personal computer has run Linux for some time so I have some expertise with Unix systems and shells. I also attended the Unix tools lectures given by Dr Marcus Kuhn during part IB. I already have some initial experience writing Python code. Finally, I learnt Emacs over the summer vacation.

Substance and Structure of the Project

The project will be written in Python due to it being a mature interpreted language that supports object oriented programming. I intend to fully exploit the wider Python ecosystem, using the tools available to perform tasks such as parser building and test framework development.

The shell itself will offer a full interpreter for shell scripts, using a syntax similar to that found in bash [2]. POSIX compliance requires a standard shell syntax [3], but other projects in this area have not aimed for compliance to improve readability [4] or power [5]. I will therefore modify the bash syntax to make shell scripts more expressive.

Many Unix utilities output data in columns, largely for readability. I intend to make this convention a requirement, so that data is always structured in a tabular format. This will enable me to generalise the idea of utilities such as `grep` to offer the user more powerful data manipulation facilities. By adding tagging functionality, information for the user may be separated from numerical data.

With the addition of metadata to what was previously unstructured text data I can develop applications that are content-aware; capable of modifying their behaviour based on data format. Semantic information available also enables the development of tools that understand the data that flows through them. This will enable the creation of tools that visualise the data in a natural way.

To demonstrate the additional power due to manipulating structured data, I will also develop a selection of tools that provide interesting data to analyse through the oosh facilities. Obvious sources of interesting data include the file system, logs, system maintenance tools and network testing tools.

Finally, the shell will also offer network aware computation. Pipelines may be constructed that dynamically move computation between local and remote hosts to improve responsiveness.

Success Criteria

The final outcome of the project should provide:

1. A oosh shell syntax interpreter

2. Some example scripts, which demonstrate
    - powerful data mining
    - graphing of data produced
3. Network transparent file access and computation for programs run within oosh
4. Programs which exploit the fact that the object stream contains semantic information, including data manipulation functionality
5. Automatic shell optimisation based on statistics collected by the shell
6. A quantitative evaluation between oosh and other shells
7. A clearly structured dissertation

Timetable and Milestones

Weeks 1 and 2 (8/10/2009 to 24/10/2009)

Write the project proposal. Set up code repository on public facing server. Write skeleton code and test code repository. Design and implement object stream structure. Set up a test harness to facilitate unit testing. Research Python console libraries.

Milestones: Proposal written and submitted. Some basic code available in git repository.

Weeks 3 and 4 (25/10/2009 to 4/11/2009)

Research most used Unix command line tools. Select a collection to implement, plus some others which demonstrate the power of object streams. Implement them.

Milestones: Several commands available to be run within oosh.

Weeks 5 and 6 (5/11/2009 to 18/11/2009)

Research Python networking facilities and develop network independence features. Add command line tools which demonstrate its usefulness. Start benchmarking, particularly with networking.

Milestones: Commands available within oosh that use the networking facilities. Some raw benchmark data collected.

Weeks 7 and 8 (19/11/2009 to 2/12/2009)

Finalise oosh syntax for shell scripting. Implement interpreter. Research most common bash shell scripts. Create demonstration oosh shell scripts.

Milestones: Formal specification of oosh shell syntax. Example scripts written and working.

End of Michaelmas term.

Weeks 9 to 11 (3/12/2009 to 23/12/2009)

Slack time. Extend test coverage. Add UI polish. Implement data visualisation facilities.

Milestones: Higher test coverage. Shell tab completion. Oosh reaches feature completion.

Beginning of Lent term.

Week 12 (14/1/2010 to 20/1/2010)

Write progress report. Prepare demonstration. Review timetable.

Milestones: Written and submitted progress report. Prepared and given demonstration. Finalised timetable available.

Weeks 13 to 15 (21/1/2010 to 10/2/2010)

Write main body dissertation including preparation and implementation. Write text, draw figures, prepare bibliography.

Milestones: Main body of dissertation written.

Weeks 16 to 18 (11/2/2010 to 3/3/2010)

Write evaluation section of dissertation. Perform any remaining benchmarks as necessary.

Milestones: Text of dissertation complete and proof read by at least supervisor and director of studies.

Week 19 (4/3/2010 to 10/3/2010)

Solicit dissertation feedback. Exchange dissertations with other students and proof read. Print dissertation.

Milestones: Paper copy of final dissertation draft.

End of Lent term.

Dissertation and source code due on the 14 May 2010.

Bibliography

[1] Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*. `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html` fetched 15th October 2009.

[2] The GNU Project, *Bash Reference Manual*. `http://www.gnu.org/software/bash/manual/bashref.html` fetched 15th October 2009.

[3] The Open Group, *POSIX:2008 (IEEE Std 1003.1-2008) Shell and Utilities*. `http://www.opengroup.org/onlinepubs/9699919799/` fetched 15th October 2009.

[4] Axel Liljencrantz, *Fish user documentation*. `http://fishshell.org/user_doc/index.html` fetched 15th October 2009.

[5] Microsoft Developer Network, *Windows Powershell Quickstart*. `http://channel9.msdn.com/wiki/windowspowershellquickstart/` fetched 15th October 2009.

Appendix B

Code Samples

I have included the source code for `oosh_sort` as an example of a command that understands the Oosh data format and acts accordingly. I have also included the source code for the recursive descent tree evaluator used within the shell itself.

B.1 oosh_sort.py

```
import sys
import operator
import oosh
from oosh import OoshError

args = sys.argv[1:]
if len(args) < 1:
    print("Usage: oosh_sort columnname")
    sys.exit(1)

sort_on = args[0]
pipein = sys.stdin.read().splitlines()
lines = oosh.get_from_pipe(pipein)
# check we are sorting on a valid column
if len(lines) > 0 and sort_on not in lines[0].keys():
    print(sort_on,"is not a valid column name")
    sys.exit(1)
```

```
# treat data numerically if possible, it comes in as a string
# preferring whole numbers if possible
is_numeric = True
for line in lines:
    try:
        line[sort_on] = int(line[sort_on])
    except ValueError:
        try:
            line[sort_on] = float(line[sort_on])
        except ValueError:
            is_numeric = False
            break
# if we have mixed data, we may have converted to float/int, so revert
if not is_numeric:
    for line in lines:
        line[sort_on] = str(line[sort_on])

lines.sort(key=operator.itemgetter(sort_on))
for dic in lines:
    sys.stdout.write(dic.__repr__() + '\n')
```

B.2 Tree Evaluator

```
def eval_tree(self, ast, pipe_pointer):
    # recurse down tree, starting processes and passing pointers
    # of pipes created as appopriate.

    # returns a tuple (stdout_pipe_pointer, return_code)

    if ast is None:
        print('Evaluated empty tree')
        return (PipePointer(), 0)

    if ast[0] == 'sequence':
        (stdout, returncode) = self.eval_tree(ast[1], PipePointer())
        self.output_for_user.append(stdout)
        return self.eval_tree(ast[2], PipePointer())
```

```
    elif ast[0] == 'savepipe':
        (pipe_out, return_code) = self.eval_tree(ast[1], PipePointer())
        pipe_number = ast[2][1:]
        self.saved_pipe_data[pipe_number] = pipe_out.read()
        return (PipePointer(), return_code)

    elif ast[0] == 'for':
        old_variables = self.variables.copy()
        for value in self.flatten_tree(ast[2]):
            self.variables[ast[1]] = value
            (stdout, return_code) = self.eval_tree(ast[3], PipePointer())
            self.output_for_user.append(stdout)
        self.variables = old_variables
        return (PipePointer(), return_code)

    elif ast[0] == 'while':
        while return_code == 0:
            (dont_care, return_code) = self.eval_tree(ast[1], PipePointer())
            (stdout, final_return_code) = self.eval_tree(ast[2], PipePointer())
            self.output_for_user.append(stdout)
        return (PipePointer(), final_return_code)

    elif ast[0] == 'if':
        (stdout, return_code) = self.eval_tree(ast[1], PipePointer())
        if return_code == 0: # 0 is true for shells
            return self.eval_tree(ast[2], PipePointer())
        else:
            return (PipePointer(), 0)

    elif ast[0] == 'if-else':
        (stdout, return_code) = self.eval_tree(ast[1], PipePointer())
        if return_code == 0:
            return self.eval_tree(ast[2], PipePointer())
        else:
            return self.eval_tree(ast[3], PipePointer())

    elif ast[0] == 'assign':
        self.variables[ast[1]] = self.flatten_tree(ast[2])[0]
        return (PipePointer(), 0)
```

```
    elif ast[0] == 'derefpipe':
        # create a process to give us a stdout to pass
        # to whatever is next in the pipeline
        old_pipe_name = ast[1][1:] # e.g. |2
        try:
            old_pipe_data = self.saved_pipe_data[old_pipe_name]
            old_pipe_pointer = self.pipe_from_data(old_pipe_data)
            return self.eval_tree(ast[2], PipePointer(old_pipe_pointer))
        except KeyError:
            raise OoshError("You have not saved a pipe numbered " +
                            old_pipe_name)

    elif ast[0] == 'simplecommand':
        command = self.flatten_tree(ast[1])
        return self.shell_command(command, pipe_pointer)

    elif ast[0] == 'derefmultipipe':
        pipe_names = ast[1][1:].split('+')

        first_pipe_data = self.saved_pipe_data[pipe_names[0]]
        first_pipe_pointer = self.pipe_from_data(first_pipe_data)

        second_pipe_data = self.saved_pipe_data[pipe_names[1]]
        second_pipe_pointer = self.pipe_from_data(second_pipe_data)

        # we call command, appending argument of second pipe
        command = self.flatten_tree(ast[2][1])
        command.append(str(second_pipe_pointer.fileno()))

        # check user hasn't tried to run this remotely
        if self.specifies_location(command[0]):
            raise OoshError("Cannot run multipipe commands remotely")

        return self.shell_command(command, PipePointer(first_pipe_pointer))

    elif ast[0] == 'pipedcommand':
        (stdout, return_code) = self.eval_tree(ast[1], pipe_pointer)
        return self.eval_tree(ast[2], stdout)
```

```
    elif ast[0] is None: # occurs with trailing ;
        pass

    else:
        raise OoshError("Unknown tree " + str(ast))
```