Bash Language Report

submitted to
Teresa Cole
Department of Computer Science
Boise State University
November 18, 2011

by
James Kress
CS354
Fall 2011

Table of Contents

1. Introduction

Bash is a general purpose command line interpreted language. The name is an acronym for the ‘Bourne-Again SHell', a pun on Steve Bourne, the author of the direct ancestor of the UNIX shell /bin/sh [3].

In terms of features, Bash has all of the most popular shell commands, as well as a suite of advanced commands like job-control. For the average user, Bash will exceed expectations. It is very easy to create automated scripts, customize the runtime environment with macros, and create more advanced programs typically left to other programming languages.

It is in the area of scripting that Bash really comes to shine. In terms of the general language evaluation criteria, Bash scores quite high. Readability of a Bash script is fairly good. It has most of the control structures that can be found in traditional languages like C, including ‘if’ statement selection, ‘while’ statement selection, ‘for’ loop generation, and ‘until’ loop selection. Bash even contains a number of data structures natively, including arrays, hash tables, and the ability to create stacks and queues. While all of these included traditional data structures and control statements aid in readability, they also really help with the writability of Bash scripts.

The only real difference between a Bash script and a traditional C program in terms of readability is a slight semantic difference, and the way that variables are declared and dealt with in Bash. Bash does not require that variables be typed, and generally most are not. This decreases the readability and writability just a bit. This is because it becomes possible to improperly compare variables, which when undeclared are initialized as strings, which can produce unexpected results. This minor shortcoming, however, is more than made up for by the increased freedom to the programmer and the ease with which Bash scripts can be created.

Another important feature of Bash is its portability. Bash has been ported to nearly every version on UNIX, and even a few versions of windows. Because of this, and the fact that Bash scripts are interpreted and not compiled, scripts are easily moved between machines that have Bash. This can become an invaluable feature when a script has been highly customized to create a favorite user environment or perform important functions like program testing and data collection.

2. The creators and Maintainers of Bash

Bash was originally developed in 1987 by Brian Fox, an employee of the Free Software Foundation [3]. The current developer is Chet Ramey, who works at Case Western University. New releases of Bash are introduced into the market at infrequent intervals of several years. The current version is 4.2, which was released on February 14, 2011. Apart from new releases, bug fixes and patches are often available multiple times a year. Many of these updates and bug fixes are not only intended to make Bash more user friendly and add functionality, but also to make it compatible with the IEEE POSIX Shell and Tools specification [4]. This group, whose acronym stands for the Institute of

Electrical and Electronics Engineers, publishes the POSIX Shell and Tools ("Portable Operating System Interface" based on UniX) specifications to make Unix shells more universal and compatible in their core concepts.

The early versions of Bash were concerned with meeting these standards even before they became an industry standard. A few examples of functions that the early releases of the POSIX specifications were seeking to standardize were basic flow control and programming execution constructs, I/O redirection and pipelining, argument handling, variable expansion, and quoting [3]. By following these standards, Bash and other UNIX based shells are able to provide the basic underpinning functions of a Shell in a uniform manner, and thus allow users to move between multiple Shells with ease.

3. Bash Shell

At its base, a shell is simply a macro processor that executes commands. A UNIX shell is both a command interpreter, which provides the user interface to the rich set of UNIX utilities, and a programming language, allowing these utilities to be combined. The shell reads commands either from a terminal or a file. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories like /bin, allowing users or groups to establish custom environments [3].

3.1. Command Line Interpreter

A shell allows execution of UNIX commands, both synchronously and asynchronously. The redirection constructs permit fine-grained control of the input and output of those commands, and the shell allows control over the contents of their environment. UNIX shells also provide a small set of built-in commands implementing functionality impossible (e.g. cd, break, continue, and exec) or inconvenient (e.g. history, getopts, kill, or pwd) to obtain via separate utilities [3].

3.2. As a Programming Language

While executing commands is essential, most of the power of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions. The basic syntactic element is a simple command. A simple command consists of an optional set of variable assignments, a command word, and an optional list of arguments. Operators to redirect input and output may appear anywhere in a simple command [3]. For example:

```
echo
echo "============ $x ================"
rm output.txt
./checkers impPlayer computer 1 2>>output.txt
```

More complex commands in Bash can create fully fledged programs that perform some of the more complex and useful features of the shell environment. A few examples of shell commands that are scripts are: sort, wc (word count), bc (basic calculator), grep, and echo.

3.2.1. Bash Language Syntax

The language base of Bash is a little bit more complex than that of the C language, but it follows along the same lines. The only differences are semantic. For instance, semicolons are optional in Bash if no more than one command is on each line, square brackets are used instead of parenthesis in if statements, and most significantly of all variables are not typed in Bash. This means that every variable is inferred to be a string when it is instantiated, unless the user specifically declares it to be one of the supported types of included variables. The following, Figure 3.2.1-1: Bash Script, is a script written in Bash to run a checkers playing program a given number of times, direct and append standard output to one file, direct and append standard error to a different file, print a running message to the console, and then call a java program after the script is finished to summarize the results from the output files.

```
1 #!/bin/bash
2 # checkers automated testing bot
3 # version 1.01
4 #       checkers.c
5
6 make clean
7 make all
8
9         rm output.txt stanOutput.txt
10        for ((x = 0; x<10; x++))
11            do
12            echo "============ Test $x ================"
13            echo $n>>output.txt
14            echo "============ Test $x ================">>output.txt
15            ./checkers impPlayer minMax 3 >>stanOutput.txt 2>>output.txt
16        done
17
18 javac testProgSum.java
19 java testProgSum output.txt
20
```

Figure 3.2.1 - 1: Bash Script

As can be seen from the example above, semicolons are not necessary to complete blocks of code. In addition, ‘for’ loops operate a little bit differently than they do in C. In Bash we use the ‘do’ keyword after the ‘for’ statement, and then to close out the loop, the word ‘done’ is employed. This allows the script to be easily parsed upon execution.

At the bottom of the example script, a java program is also being complied and called as though it was being done right in the command line. This is one of the beauties of Bash. Since it is an interpreted language, it is possible to put commands in scripts that

would normally be put into the command line. This enables things like a script to be run, and then immediately afterwards a different program written in a completely different language to be run, generating a statistical report of the output of the script. While automated testing of other programs is not the only benefit of Bash scripts, it is a major plus.

3.2.2. Data Types

Bash, like many other scripting languages, is unique in the way that variables need not have a declared type. That is because, in Bash, all variables are treated as strings. This convention gives the user a lot of power over how they program, but it can also introduce some major problems. Since most variables are not typed, the script is unable to do any type checking upon execution. With no type checking, it is only too easy to make a mistake when comparing variables. A main reason for this is that Bash has different comparison operations for the desired output of the comparison (See Table 3.2.2 – 1: Relational Operators & Table 3.2.2 – 2: Arithmetic Comparisons).

When the desired comparison is a character string comparison, the regular relational operators may be used. However, if the variable is un-typed, yet the user still wants to compare it as a number, special arithmetic comparison operators need to be used.

Operator	Meaning
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

Figure 3.2.2 - 1: Arithmetic Comparisons [1]

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&&	Logical and
||	Logical or

Figure 3.2.2 – 2: Relational Operators [1]

For example, “6” is greater than “57” lexicographically, just as “p” is greater than “ox.” But of course the opposite is true when they’re compared as integers [1]. This can turn into a major potential problem in the event that the user is not aware of the ability to declare variables as typed, or that there are two different sets of relational operators. In this event, debugging problems with loops or control structures can prove to be almost impossible. This is where the use of typed variables can make life much easier.

To make Bash more reliable and safe, it is possible to declare the type of the variable that you want. There are several types that can be declared (See Table 3.2.2 -3: Variable Declaration Options). By using these different options, it is possible to make assignments type safe, and have Bash check for incorrect assignments at runtime. This is not always foolproof, however.

Option	Meaning
-a	The variables are treated as arrays
-f	Use function names only
-F	Display function names without definitions
-i	The variables are treated as integers
-r	Makes the variables read-only
-x	Makes the variables for export via the environment

Table 3.2.2 - 3: Variable Declaration Options [1]

The following (Figure 3.2.2 -1: Bash Variable Assignment) is a short script written in Bash to demonstrate the declaration of different typed and un-typed variables. Notice that there are a couple of different ways to create a string variable. One way is to just assign letters or numbers to an undeclared variable, while the other is to create a literal quoted string. All three will create a string variable with the assignment being a string. This way of operating, however, can produce unexpected results if variables are compared incorrectly or a cross assignment takes place. To mitigate this, the user can declare a type.

This script utilizes two of the declaration types to create an integer ‘a’ with the value of ’13’, and a read-only string ‘d’, with the string “5”. The output section shows the results of the data manipulation. In short, the value of the ‘string’ variable is reassigned to be the string ’13’, an attempt to assign a string value to the variable ‘a’, which was declared an integer, fails and returns 0, and an attempt to modify the contents of the read-only variable ‘d’ fails with a runtime error.

Program

```
1  #!/bin/bash
2
3  string=apples                 # String
4  declare -i a=13               # Integer
5  declare -r d=5                # read-only
6
7  echo string $string
8  string=$a
9  echo string = $string
10
11 a="integer"
12 echo a = $a
13
14 d=$b
15 echo d = $d
16
17 exit $?
```

Output

```
string apples
string = 13
a = 0
variableDecloration.txt: line 14: d: readonly variable
d = 5
```

Figure 3.2.2 - 1: Bash Variable Assignment

Taking all of the different ways to assign and manipulate variable values into account, Bash has the propensity to be a bit confusing. However, with a little bit of practice, it can become very easy to get used to not needing to declare variables, and become proficient with utilizing the wide array of tools available for comparisons on un-typed values.

4. Language Evaluation

An important concept when considering whether or not to use a language for a particular task is the readability, writability, reliability, cost, and portability of the code. Following is a breakdown of the various aspects of Bash.

4.1. Readability

The readability of a program is defined as how easy it is to read and understand any given program within the language. This is one of the most important aspects of a language, and can affect all future uses and users of the code as it is maintained or updated. Some of the main concepts that are evaluated when the readability of a program is being discussed are syntax design, data types, and overall simplicity.

In terms of readability, Bash does fairly well. Most of its syntax is similar to that of C, and as such, if the user knows the basic language constructs of the C language, it is not difficult to read through a Bash program. The main differences in syntax from C are the different methods of implementing selection and loop statements, the lack of semicolons, and the shell specific features that can be called from within the script. A few examples of shell specific operations are redirection to files based on the output stream, calling other programs from within the script, and usage of existing shell scripts to perform functions like word count.

One other area that Bash is more ambiguous than other languages is variable declaration. Because of the way that Bash instantiates variables, it creates a large amount of ambiguity when the variables are used in conjunction with other variables later on in the application. This, above everything else, is what brings down the readability of Bash scripts.

4.2. Writability

The writability of a program is defined as how easy it is to create a program for a given problem in some domain. Most if not all of the aspects that go into the readability of code, go into the writability of that same code. Writability is important because when a language is easy to write, programs can be more organize, and the programmer does not need to strategically plan each piece of code that is implemented.

The writability of Bash directly follows from its readability. Most aspects of Bash are fairly easy to grasp. In addition, Bash does not have very many natively supported data structures. Thus, the user does not have to memorize a large API library just to use the language. This simplicity makes Bash ideal for scripting and shell task automation. If tasks like shell task automation or automated program testing were implemented in

another language like C, then that code would become ungainly and unreadable, since C was not designed for that task.

The only thing that reduces the writability of Bash from a beginner's perspective, is the propensity to not define variables. This means that variables can be declared anywhere in the program, making the task of debugging incorrect output a difficult task, However, for the tasks that Bash was designed to do, it creates an easy to use and customize environment within the shell.

4.3. Reliability

The reliability of a programming language is to what extent that a program meets its performance expectations under all conditions. This involves several different aspects of the language. A few of the major points that reliability addresses is type checking, exception handling, readability, and writability.

The reliability of a Bash script depends greatly on how it is implemented. However, some generalities can be made about all Bash scripts that are rather revealing. First and foremost, Bash scripts can be very type un-safe. This is because the majority of people do not declare types when they are creating variables in Bash. This is mainly due to the fact that Bash prefers to deal with variables as strings, and in fact, when no variable definition is given, the default value is a string. This use of every variable is a string unless it has been explicitly typed to a defined and recognized type, creates some problems when it comes to making Bash code reliable.

The primary aspect of reliability that is decreased with no variable types, is type checking and run-time exception handling. There are many examples of this behavior being a problem. First, consider the even in which a variable has been given a value that was supposed to be a constant value throughout the entire program. Consider further that if the variable were to change or get reassigned, that the error may go unnoticed, but that it would cause a drastic difference in the data that the script produces. If the variable had been declared as read-only upon inception, then upon execution, the script interpreter would have been able to recognize that an attempted assignment to a read-only variable had taken place, and would have thrown an exception in the output from the script. This then would allow the programmer to easily fix the error, and move on with the program. However, since variable types are not required, danger ensues.

The next aspect of Bash that reduces reliability slightly is the semi ambiguous natures of both readability and writability. While neither of these is nearly as bad as type checking, they both add to a slight decrease in overall reliability.

Overall, if Bash is used in a safe and sane manner, it is a highly reliable scripting language, and is able to do many things that will aid its users for years to come.

4.4. Cost

The ultimate total cost of a programming language is a function of many of its characteristics [1]. A few of these costs include the time and resources that go into

training programmers to use the language, the cost that is associated with writing a program in that language (heavily related to writability), and the cost of executing the program.

Bash scores well in terms of cost in many categories. First, if a programmer has had any exposure to C, picking up Bash will not take too long. Many of the control structures and selection statements have syntax that is most closely related to C. In addition, many programmers will be happy with freedom that comes from not having to declare a type for every variable that they want to use in a script.

Secondly, Bash receives a moderate score in the areas of writability. While being free to not need to declare every variable with a type, it creates somewhat ambiguous code the longer the project, and can really detract from the future maintenance and use of the code. Lastly, Bash gets a very high score in the area of execution time. Since it is an interpreted language, it is not compiled which saves an entire step each time a programmer wants to run the code. This does come with its drawbacks, specifically it is slower than compiled languages, but in most instances for short programs like word count, this is more than made up for in the fact that a Java or C program would need to go through two entire steps to accomplish a task that a shell can generally complete in under a second with a single call.

4.5. Portability

Portability is the ease with which programs can be moved from one implementation to another. Portability is most strongly influenced by the degree of standardization of the language [1].

Bash is free software, and as such, is very easy to obtain. In addition to being free, Bash has been ported to many different operating environments. Not only has Bash been ported to nearly every version of UNIX, it is also available for a number of other operating systems. Most notably, QNX, LynxOS, and some versions of Windows using the Win32 programming interface [2].

Apart from being free and easily ported to many different operating systems, Bash conforms to many of the specifications in the IEEE POSIX Shell and Tools specifications publication. This standardization makes switching between Bash and other shells a snap. Also, as some versions of Bash are not yet available on some operating systems, this standardization allows Bash programs to be easily moved between systems. If the target and base system are both using the same version of Bash, then the scripts are directly portable, if the target system is running a later version of Bash than the base system, this script too should be directly portable. The opposite however is not necessarily true. In many instances it is, but if the user is taking advantage of any of the updates and upgrades to flow control or similar core revisions code may need to be modified.

Taking the facts about the availability and easy portability of Bash into account, it is easy to see that Bash is a highly functional and portable shell and scripting language. With its support on so many different operating systems, conformance to the IEEE

standards, simple syntax, and easy writability, Bash is a great choice for a general purpose scripting and shell environment.

5. Conclusion

Bash can be used as a very full featured language for many different tasks. It has all of the facilities for creating control and loop structures, arrays, hash tables, and a suite of impressive shell interaction protocols that can make tasks trivial. If the goal of a Bash script is to get a repetitive shell task done quickly and easily, then Bash is probably a great choice.

It is important to remember however, that there are other languages that programs can be written in. This is a particularly important concept to grasp at the beginning of a project. Different languages were designed to be more functional in one area or another compared to other languages. If there comes a time when you start writing a script and after several hours of work find that you've created a monster with many hundreds of lines of complicated code, then you may need to reassess the situation. While this is not always a bad thing, it is a good idea to be thinking about whether the job could be done in a better way [1].

Overall, over the last semester, I have found Bash to be a very useful, if not helpful, in my programming classes. It has allowed me to get excited about exploring the different ways that tasks can be accomplished with programs, and to save a lot of time in the testing of programs. This was particularly notable in my artificial intelligence class when we wanted to run a gamut of about 500 test runs on each iteration of our checkers playing program.

Addressing the possible shortcomings noted above about Bash in its evaluation as a programming language, I find that it is actually a very enjoyable and easy to write language. Its only shortcomings really come from the limited range of what types of variables can be typed. It is not really designed for floating point numbers, and as such isn't typed to use them. So apart from variable typing, I feel that Bash is exceedingly close to the syntax of C, and is not hard to pick up after investing a little bit of time into it. This in the end, will give great returns in time saved by automating futre testing of programs and other tasks.

6. Author Information

James Kress is a Computer Science student at Boise State University. He currently has junior standing, and is expecting to graduate in May of 2013.

7. Bibliography

[1] Newham, Cameron and Bill Rosenblatt. Learning the Bash Shell. Sebastopol: O'Reilly Media, Inc, 2005.

[2] Ramey, Chet. Bash FAQ. 14 2 2011. 16 11 2011 <http://tiswww.case.edu/php/chet/Bash/FAQ>.

[3] —. "Bash, the Bourne-Again Shell." 1994.

[4] —. "What's GNU: Bash - The GNU Shell." Linux Journa; (1994).