Hush: A Lua-Based Shell Language

Gabriel Bastos
UFMG
Belo Horizonte, Minas Gerais, Brazil
bastos.gabriel@dcc.ufmg.br

Fernando M. Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

Abstract

Shells are one of the core tools in modern computer systems, and although traditional shells are powerful and mature tools, their age starts to show. As they lack support for some common programming constructs, such as data structures and floating point numbers, companies such as Google recommend avoidance of these tools for anything beyond trivial usage. In this paper, we present Hush, a modern shell programming language, which aims to surpass known limitations in traditional shells. Hush is inspired by Lua, a well established embedded script language, and thus supports common programming paradigms. We provide a prototype interpreter capable of support evaluation of the language in real scenarios.

Keywords: Shell, Programming Language, Unix

1 Introduction

Shells are one of the core tools in modern computer systems, and they fundamentally consist of a command interpreter capable of coordinating external programs. The concept of this tool was born in 1965 with the work of Glenda Schroeder [11], and evolved into dozens of shells that exist at the present moment. One of the most popular shells is the Borne Again Shell [6], which was originally developed by Brian Fox in 1989, predating even the Linux kernel. The traditional concept of a shell is stabilized in the POSIX standard [8], and most implementations abide by such standard.

Although traditional shells are powerful and mature tools, their age starts to show. Dealing with arrays can be clumsy, associative arrays are limited if not completely unavailable, there is no robust support for floating point numbers, many arithmetic operators are lacking, among further limitations. [7] In practice, beyond basic data structures are intractable, and thus rarely used due to the lack of ergonomics. [9]

> If you are writing a script that is more than 100 lines long, or that uses non-straightforward control flow logic, you should rewrite it in a more structured language *now*. – Google's shell style guide [7]

These issues have motivated many efforts to make modern shells that are capable of dealing with structured data and general purpose programming. While some of these extend traditional shells with new features, others relinquish the POSIX [8] standard and implement novel syntax and semantics. But most of these have one common characteristic, they are shells that strive to become full-featured programming languages. In this paper, we present Hush, a programming language that strives to become a shell.

Hush is designed as the conjunction of a well established embedded scripting language, and standard shell functionality. We expect this approach to provide a powerful tool for building robust shell scripts. An interpreter prototype was developed and is available for evaluating the language in practical real scenarios.

2 Overview

Being inspired on the core concepts of Lua, Hush elaborates and extends them by adding shell syntax and capabilities. By combining the power of a full-featured programming language with the expressiveness of shells, Hush strives to be a shell that is well suited for building robust solutions that orchestrate external programs.

Hush provides basic support for three major programming paradigms. For imperative programming, basic constructs such as sequential statements, mutable variables, conditionals, loops and procedures are supported. Functions are first-class values, supporting higher-order parameters, variable capturing and recursive calls, which are the basic tools for functional programming. Finally, objects can be implemented with dictionaries, which supports methods with instance access and shorthand call syntax, also allowing inheritance and specialization.

As in Lua, Hush provides strong and dynamic typing based on a handful of built-in types, and does not provide user-defined types. Row polymorphism is provided through duck typing. The intent is to keep the language simple, without compromising too much expressiveness. Advanced Lua features like *metatables* were disregarded in Hush.

Shell capabilities are expressed through command blocks, which allow execution of external programs as first-class expressions. Fundamental shell features such as pipes, redirections, and variable substitution are provided with traditional and intuitive syntax.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

UFMG, 2021, Belo Horizonte, MG, Brazil

© 2021 Copyright held by the owner/author(s).

3 Syntax and Semantics

Hush's syntax and semantics are extensively inspired in those of Lua, aiming to provide a dynamic and agile programming environment. Additionally, Hush provides specialized syntax for invoking and interconnecting external programs, a feature that is essential to shells.

3.1 Syntax

Identifiers may be composed of alphanumeric and underline characters, but must not start with a number, and neither be a keyword. Single line comments are supported.

```
let if then else end for not do while return
in and or true false nil break self function
```

Listing 1. keywords

```
# This is a comment.
```

Listing 2. comment syntax

All types except the *error* type have corresponding literal syntax. *Dict* keys in literals must be valid identifiers.

```
let dict = @[
    _nil: nil,
    bool: true,
    int: 42,
    float: 3.14,
    string: "abc",
    char: 'c',
    array: [ 1, '2', 3.14 ],
    _function: function(x, y)
        x + y
    end,
]
```

Listing 3. literals syntax

Variables are declared with the `let` keyword, and can be assigned with the = operator.

```
let x
x = 1
```

```
let y = 2
# syntax sugar
```

Listing 4. variable declaration syntax

if conditionals are expressions, and may have an optional *else* clause.

```
if <expr> then
    <block>
end
```

```
if <expr> then
    <block>
else
    <block>
end
```

Listing 5. *if* expression syntax

Loops have a single and straightforward syntax:

```
while <expr> do
    <block>
end
```

```
for <ident> in <expr> do
    <block>
end
```

Listing 6. *for* and *while* loop syntax

Where <ident> is a valid identifier.

Specialized syntax is defined for invoking external programs, which are grouped in blocks of one or more commands:

```
# Standard command block
{
    <cmd> <args>; # args are optional
    <cmd> $<var> ${<var>}; # variable access
    <cmd> > file.txt; # output redirection
    <cmd> 2>1; # descriptor redirection
    <cmd> < file.txt; # input redirection
    <cmd> << "input"; # inline input
    <cmd> | <cmd> | <cmd>; # pipes
    <cmd> ?; # try operator
}
# Capture command block
# (supports all constructs described above)
${ <cmd> } # trailing semicolon is optional
```

Listing 7. command blocks

Hush provides the following operators:
Unary:

- Logical: **not** (prefix)
- Arithmetic: - (prefix)
- Field access: [], . (postfix)
- Function call: () (postfix)

Binary:

- Arithmetic: *, /, %, +, -
- Relational: >, <, >=, <=
- Equality: ==, !=
- Logical: **and**, **or**
- String concatenation: ++ (right associative)

Expressions may be of the following kinds:

- <lit> # literal
- <ident> # variable access
- **self** # self value access
- <unop> <expr> # unary operator
- <expr> <binop> <expr> # binary operator
- <if> # if conditional expression
- <expr>[<expr>] # field access
- <expr>.<ident> # field access
- <expr>(<args>) # function call
- <cmdblock> # command block

Where <args> is a comma-separated list of expressions.

Statements can be any of the following:

- `let <ident> # variable declaration`
- `<lvalue> = <expr> # assignment`
- `return <expr> # expr is optional`
- `<loop>`
- `break # only in loops`
- `<expr> # expression statement`

Finally, a Hush program consists of a block of statements, which need not be separated by any token.

3.2 Semantics

Although Hush follows the base semantics of Lua, many aspects have been revised, both to accommodate the shell capabilities, and to circumvent known quirks of the base language. Like Lua, Hush relies heavily on the expressiveness of it's builtin types, as it doesn't support user-defined types. In such type system, we have the first major change from Lua's semantics, as some types have been redesigned, and the novel `error` type has been added. The types in Hush are:

- `nil`: the unit type, used for missing values.
- `bool`: the boolean type.
- `int`: a 64 bit integer type.
- `float`: a 64 bit floating point type.
- `char`: a C-like unsigned char type.
- `string`: an immutable homogeneous array of chars.
- `array`: a 0-indexed heterogeneous array.
- `dict`: a heterogeneous hash map.
- `function`: a first-class callable function.
- `error`: a special error type, to ease distinction of errors from other values.

Therefore, comparing to Lua, both `number` and `table` types have been separated into the `int`, `float`, `array` and `dict` types. Such decision aims to allow finer grained control over values, allowing distinctions between such types when necessary. Additionally, arrays in Hush are 0-indexed, in contrary to Lua, where arrays are 1-indexed.

Hush performs type coercion only in arithmetic operators. Logical operators, *if* expressions and *while* loops expect boolean arguments, and will panic if supplied with values of any other type. Arithmetic operators may only be applied to numeric types, and will promote *int*s to *float*s if necessary. Relational operators may only be applied to pairs of numeric, *char* or *string* values. The `[]` operator accepts integer indices for *string*s and *array*s, and all types for *dict*s. Finally, the dot field access operator is syntax sugar for *string* indexes, as in `my_dict.field == my_dict["field"]`.

Scope in Hush is static, and closures are supported. As all variables must be declared, the scope of identifiers is directly stated in the code. All variables are references, and all types except *array* and *dict* are immutable.

Functions are declared with a precise amount of parameters, and such exact amount must be supplied when calling. Unlike Lua, functions return a single value. Function bodies and *if* expression blocks result in the value of the last statement. Non-expression statements result in `nil`. Additionally, function bodies can access the `self` variable, which corresponds to the dict containing the function when calling in conjunction with the dot operator, as in `obj.fun()`, or `nil` otherwise.

Hush has two distinct error mechanisms, namely `error`s and *panic*s, which exist to distinguish between recoverable and irrecoverable errors. For handling recoverable errors, Hush adopts a simple strategy, very similar to the one implemented in the Go programming language. [2]

The `error` type is provided for recoverable errors, and can be instantiated by the `std.error` function. Values of the `error` type are immutable, but otherwise act like `dict`s, containing two fields: a message `string`, and a context of any type. Examples of recoverable errors are:

- file not found
- permission error
- invalid format
- command not found
- command returned non-zero exit status

*Panic*s are irrecoverable errors, caused by ill-formed programs. When a *panic* occurs, Hush immediately halts the script execution, and displays the error message to *stderr*. Examples of errors that cause a *panic* are:

- integer division by zero
- array index out of bounds
- attempt to call a value that is not a function
- missing arguments in function call

3.3 Command Blocks

In Hush, command blocks are composed of one or more commands, which themselves are composed of one or more basic commands.

A basic command consists of a program name, the list of arguments for such program, optional redirections, and the optional *try* operator, as shown in Listing 7. On execution, the arguments are evaluated, and *path* environment lookup determines the program executable location. Hush then forks, sets up the requested redirections, and *exec*s the target executable with the given arguments. Execution is then awaited, and the status code is obtained. If the status is different from zero, an `error` is produced, and the command block execution bails unless the *try* operator is present.

Commands are either a single basic command, or a pipeline. Pipelines are a mechanism of data flow parallelism, where the standard output of a program is attached to the standard input of the following program, through an Unix pipe. After all pipes are attached, all programs are executed simultaneously, and execution ends when the last program terminates. In Hush, a basic command results in a single

result value (zero or error), and pipelines result in a list of those values.

Finally, command blocks are a list of commands, which are executed sequentially. Currently, there are two kinds of command blocks: standard and capture. In standard blocks, all commands inherit the shell's standard I/O. In capture blocks, the output of all commands is captured through an Unix pipe, and stored by Hush. In this case, the result of the block's execution is a dictionary containing the execution status, the *stdout*, and the *stderr*. Therefore, when using the capture block, one can access the produced output as strings through such fields.

4 Tooling

The standard implementation of Hush currently supports all features described in this paper. Due to the context sensitive lexical grammar of the language, a state-machine lexer was manually implemented. In order to provide good error messages, and allow full error reports for Hush programs, the implementation also features a recursive descent parser with custom synchronization strategies. Static program analysis is also implemented, not only to prevent invalid programs, such as those with usage of undeclared variables, but also to transform variable access into stack relative addresses, eliding variable lookup during runtime.

As for the runtime, simpler techniques have been applied. The interpreter is implemented as a basic recursive tree walker, and garbage collection is provided with the aid of a library. Command execution is manually implemented on top of system libraries, and tail call is implemented as the only notable optimization of program execution.

5 Related Work

There are two main families of modern shells. The first of them is based on traditional shells like the Bourne Again Shell, the C Shell, the Korn Shell, or the Almquist Shell. Some examples of these are the Z Shell [10] and the Debian Almquist Shell [13]. The second family consists of languages that have completely relinquished the POSIX standard [8]. Such languages include academic works like Choc [9], Magritte [1] and the Directed Acyclic Graph Shell [12]. Also in this family we place not-exactly academic, but independent initiatives like the Fish Shell [3], the Oil Shell [5], and the NuShell [14]. While most of these are shells that strive to become full-featured programming languages, Hush is a rework of a successful programming language into a shell. By adding shell syntax and capabilities on top of a language based on the core concepts of Lua, Hush constitutes a novel tool on the shell universe.

6 Final Thoughts

In this paper, we have presented Hush, a Lua-based shell language. Hush features core shell capabilities, including variable substitution, pipes, redirections and output capturing, on top of a general purpose scripting language inspired by Lua. A reference implementation of the interpreter is provided for evaluating a preliminary version of the language [4].

There are many directions for future work. For instance, it would be interesting to augment the interpreter to allow interactive input, allowing the usage of Hush as an interactive shell. Further, in order to allow composition of complex scripts and libraries, a module system would be a valuable addition to the language. Finally, a usability study of the language in real-world scenarios would provide insights on the usefulness and possible shortcomings of the tool.

References

[1] Jeanine Miller Adkisson. 2019. Magritte. In *Proceedings of the 3rd International Companion Conference on Art, Science, and Engineering of Programming - Programming '19*. ACM Press. https://doi.org/10.1145/3328433.3328467

[2] The Go Authors. 2021. The Go Programming Language. https://golang.org/ Acesso em agosto de 2021.

[3] Axel Liljencrantz, et al. 2005. Fish: a smart and user-friendly command line shell. https://fishshell.com/ Acesso em janeiro de 2021.

[4] Gabriel Bastos and Fernando Magno Quintão Pereira. 2021. Hush: a unix shell based on the Lua programming language. https://github.com/gahag/hush Acesso em setembro de 2021.

[5] Andy Chu. 2017. Oil: a new Unix shell. http://www.oilshell.org/ Acesso em janeiro de 2021.

[6] Brian Fox. 1989. Bash is in beta release! https://groups.google.com/group/gnu.announce/msg/a509f48ffb298c35 encaminhado por Leonard H. Tower Jr. em 8 de junho de 1989. Acesso em março de 2021.

[7] Google. 2021. Google's Shell Style Guide. https://google.github.io/styleguide/shellguide.html Acesso em março de 2021.

[8] IEEE. 2017. IEEE Std 1003.1-2017, Standard for Information Technology – Portable Operating System Interface (POSIX). https://pubs.opengroup.org/onlinepubs/9699919799/ Acesso em janeiro de 2021.

[9] Michael MacInnis. 2010. *Choc : a command and programming language*. Ph.D. Dissertation. https://doi.org/10.22215/etd/2010-09226

[10] Peter Stephenson, et al. 1990. Zsh: a shell and a powerful scripting language. https://www.zsh.org/ Acesso em janeiro de 2021.

[11] Louis Pouzin. 2021. The Origin of the Shell. https://www.multicians.org/shell.html Acesso em agosto de 2021.

[12] D. Spinellis and M. Fragkoulis. 2017. Extending Unix Pipelines to DAGs. *IEEE Trans. Comput.* 66, 9 (2017), 1547–1561. https://doi.org/10.1109/TC.2017.2695447

[13] Herbert Xu. 1996. The Debian Almquit Shell. https://git.kernel.org/pub/scm/utils/dash/dash.git Acesso em março de 2021.

[14] Jonathan Turner Yehuda Katz. 2019. Nushell: a new type of shell. https://www.nushell.sh/ Acesso em janeiro de 2021.