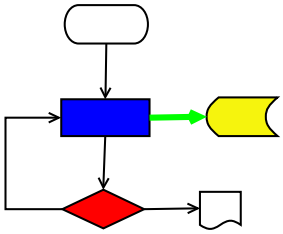
VICI



VISUAL CHART INTERPRETER

Preliminary Analysis

Publication History

Date	Who	What Changes
6 April 2014	Brenton Ross	Initial version.

Table of Contents

1 Introduction

1.1 Scope

The preliminary analysis task is to investigate the implications and consequences of the Vision Statement. It has the task of setting the boundaries for the project and for documenting the external interfaces to the systems that the project will connect with.

Understanding the scope of the problem is the first step on the path to true panic.

1.2 Overview

The UNIX desktop environments, from CDE on, have focussed on running GUI programs. The result is that the amazingly rich set of command line UNIX/Linux programs is now almost invisible from the desktop.

It is difficult for desktop users to use and combine these programs to automate their work. Many of us have seen office workers manually using a computer to perform the same tasks every day. It is often the case that these tasks can be automated with some simple shell scripting, but there is rarely the opportunity for experienced programmers to assist with this sort of work. The users have the motivation and probably the opportunity, but are often unaware that scripting is even a possibility.

The aim of this project is provide a set of tools that will give a novice user the ability to build and run scripts that can be used to automate some of their routine tasks.

In addition to actually constructing a script there are some other problems that must be overcome:

- There are literally thousands of programs available on a typical Linux computer. There must be some way for the novice user to select the programs required for their problem.
- Each program has its own set of parameters, and there is often very little that is common or standard in the selection of the identifiers for these parameters. Help in setting these parameters must be part of the solution.
- It must be possible to install the scripts into the desktop environment so that they can be used like any other program.

1.3 Audience

This document is intended for anyone with an interest in the VICI project.

It will be used by the developers to record the reasoning behind some of the decisions made during development.

It can be used by those wishing to determine if the software created by this project would be useful.

1.4 Project Name

The name chosen for this project is “VICI” for VIsual Chart Interpreter. For a while the name was to be GUISH but on reflection the operation is not like that of a shell interpreter, so the project has been re-established as VICI.

Interestingly for such a common sounding name there are no existing projects on Source Forge or Savannah. There are quite a few projects on GitHub with the VICI name but it appears that would not be an issue if we decide to use GitHub as our repository

2 Survey Of Other Visual Shells

This section will summarize the attempts made by others to create similar programs to what is proposed for this project. This might take a while since Wikipedia lists 86 entries under Visual Programming Languages.

2.1 Visual Programming Languages

The following descriptions were drawn from Wikipedia pages, and sometimes the linked web sites of the actual products.

2.1.1 AgentSheets

An object oriented visual language in which active agents are placed onto a spreadsheet like grid where they can communicate with each other.

The programming seems to be limited to what the original designers envisaged and the entire thing runs in Java so it is not what this project is intending.

The original ideas were first developed in 1991 and it was still an active product in 2010 so it appears that the concept is robust.

2.1.2 Alice

Alice uses a drag and drop environment to create computer animations using 3D models. Users can place objects from Alice's gallery into the virtual world that they have imagined, and then they can program by dragging and dropping tiles that represent logical structures.

While the execution is visible (using Java), it appears that the scripting is done with a conventional text approach.

The aim of the environment is to better allow students to visualise what is happening while their program is running and are thus better able to correct problems. It appears to be quite popular as a student's introductory language.

We will need to include some way for our users to test their scripts.

2.1.3 Analytica

A visual software package for creating, analysing and communicating quantitative decision models. Analytica includes hierarchical influence diagrams for visual creation and view of models, intelligent arrays for working with multidimensional data, and Monte Carlo simulation for analysing risk and uncertainty. The design of Analytica, especially its influence diagrams and treatment of uncertainty, is based on ideas from the field of decision analysis. Analytica includes a computer language, which is notable in being declarative (non-procedural) for referential transparency, supporting array abstraction, and providing automatic dependency maintenance for efficient sequencing of computation.

It appears to be quite popular in the finance and risk management industries.

Our analysis will need to weigh a declarative approach against the procedural approach to defining our programs.

2.1.4 AppWare

This was a graphical language that allowed its user to route events between display and function objects using a graphical editor. The product underwent several changes of name and ownership and eventually died, which seems unfortunate since it appears to be a solid idea that would allow a market for the objects and would make it easy for end users to construct simple programs.

If we view the UNIX system commands as functions and the files and variables as objects (and zenity commands for other GUI objects) then this project might almost be considered a revival of AppWare, but done in an easily extensible manner.

2.1.5 AudioMulch

This uses a graphical editor to connect together modules which perform various sound processing tasks.

Not immediately applicable to vici but it does demonstrate the utility of graphical languages for the non-programmer.

2.1.6 Authorware

Authorware used a visual interface with icons, representing essential components of the interactive learning experience. "Authors" placed icons along a “flow-line” to create a sequence of events. Icons represented such components as Display—put something on the screen, Question—ask the learner for a response, Calc—perform a calculation, read data, and/or store data, and Animate—move something around on the screen. By simply placing the icons in sequence and adjusting their properties, authors could instantly see the structure of program they were creating and, most importantly, run it to see what learners would see. On-screen changes were easy to make, even while the program was running.

The software was discontinued, apparently replaced within Adobe Systems by Flash.

This product is more evidence that a flowchart based visual programming language will be a solid foundation.

2.1.7 Automator

An application developed by Apple for OS X that implements point-and-click (or drag-and-drop) creation of work-flows for automating repetitive tasks into batches for quicker alteration, thus saving time and effort over human intervention to manually change each file separately. Automator enables the

repetition of tasks across a wide variety of programs, including the Finder, the Safari web browser, iCal, Address Book and others.

This program is almost exactly what is envisaged for vici, but with bindings into the main Apple programs. The interface, however, does not do a very good job of showing the structure of the script.

2.1.8 Befunge

An experimental language in which the commands are laid out in a grid (possibly of more than two dimensions) and the next command to be executed depends not only on the current command, but also on the previous direction that the command was arrived at from. Multiple threads of control can navigate in arbitrary directions over the set of commands. (The language also includes the ability to rewrite its commands.)

A bit too complex for our purposes, but there is nothing stopping the flow of control being in arbitrary directions if we are going to follow a flowchart model.

2.1.9 Blockly

A web based graphical language where blocks containing commands and control structures are assembled by drag and drop.

Quite a nice implementation of what we are proposing, but the web based nature makes it unusable for our purposes.

2.1.10 CODE

This uses a graphical environment to define the data flow between components of programs. The system is designed to allow the creation of parallel execution models from conventional programs.

The parallel processing part is not applicable to our proposal and their approach that reaches into the component programs to access their internal subroutines is more than we need or want. However, the use of a flowchart to describe the flow is encouraging.

2.1.11 CiMPLE

A graphical programming language for students designing control programs for robots.

The range of commands is limited to those necessary for the robot domain, but the control structures are a good illustration of what we are proposing for vici.

2.1.12 DRAKON

A graphical programming language developed for the Russian space projects. It is used a graphical editor that can produce code in various languages for normal compilation.

An excellent example of flowchart based programming but not directly applicable to our purpose. It may be useful as a guide for flowchart creation.

2.1.13 Flow-Based Programming

A programming paradigm that defines applications as networks of "black box" processes, which exchange data across predefined connections by message passing, where the connections are specified *externally* to the processes. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. FBP is thus naturally component-oriented.

This constitutes a subset of what we are proposing. The vici system will include port communications between processes, as in stdin and stdout, but we will also have control over what processes are executed and the ability to buffer results into files.

2.1.14 Game Maker

GameMaker's primary development interface uses a drag-and-drop system, allowing users unfamiliar with traditional programming to intuitively create games by visually organizing icons on the screen. These icons represent actions that would occur in a game, such as movement, basic drawing, and simple control structures.

It would seem that the graphical aspect allows the construction at an object level, with some functions (actions), but that the details are done with a conventional scripting language.

2.1.15 Google App Inventor

It allows anyone, (including people unfamiliar with computer programming), to create software applications for the Android operating system (OS). It uses a graphical interface that allows users to drag-and-drop visual objects to create an application that can run on the Android system, which runs on many mobile devices.

Similar to Blockly. We may need to decide between a tile like mechanism, used by these programs, or a flowchart mechanism.

Perhaps the Nassi–Shneiderman diagram might be able to be adapted to a tile like system.

2.1.16 Helix

Helix is a pioneering database management system for the Apple Macintosh platform, created in 1983. Helix uses a graphical "programming language" to add logic to its applications, allowing non-programmers to construct sophisticated applications.

While Helix's visual programming is possibly easier for novices to learn (because it uses a flowcharting paradigm that is intuitively understood by non-technical individuals), it becomes tedious when the amount of code to be written becomes significant, especially for an individual who can write code much more easily and conveniently than if forced to drag icons from a palette. Helix has consequently suffered from the lack of developer support and third-party applications.

This is something that we need to keep in mind. The vici system is intended for simple tasks, not major programming projects. It might be advisable to provide a means of exporting vici script to bash script so that projects can start small and grow.

2.1.17 Illumination Software Creator

A graphical IDE that allows you to drag code blocks and connect them in order to create an application program. To create a program, you simply drag code blocks onto the canvas and configure the block. Then, you connect outputs to inputs. The code blocks are categorized to make them easy to find. Clicking on a code block allows you to set various parameters for the given block. Variables, which appear in the lower-left panel, come in three types: text, number and text file.

This product is quite close to what we are proposing, and some system administrators are using it to add a GUI interface to their shell scripts. It is a proprietary program but it does illustrate that what we are proposing might be useful.

2.1.18 LabVIEW

The programming language used in LabVIEW, also referred to as G, is a data flow programming language. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

2.1.19 Limnor

A generic-purpose code-less and visual programming system. The aim is to enable users to create computer software without directly coding in a texture programming language. It can be extended by software developers.

Yet another attempt to create a visual programming environment for complete GUI applications. It has one of the most terrifyingly cluttered user interfaces I have ever seen.

2.1.20 Microsoft Visual Programming Language

This is a message passing system between active objects. A visual editor is used to connect the objects, and thus define the message paths.

Not particularly applicable to our purpose.

2.1.21 OpenWire

A visual programming environment built on Delphi in which pins on components are connected by wires to relay messages with the data types taken into account. It appears to be an attempt to produce an open source version of LabVIEW (which make the choice of Delphi a bit odd).

Limited to Delphi (and hence not available for Linux) and uses coded components within a program, rather than acting as a shell.

2.1.22 Piet

Like Befunge but with coloured coded areas representing the commands. Quite pretty to look at, but not easy to read for humans.

2.1.23 Prograph

A visual, object-oriented, data-flow, multi-paradigm programming language that uses iconic symbols to represent actions to be taken on data.

Prograph introduced a combination of object-oriented methodologies and a completely visual environment for programming. Objects are represented by hexagons with two sides, one containing the data fields, the other the methods that operate on them. Double-clicking on either side would open a window showing the details for that object; for instance, opening the variables side would show class variables at the top and instance variables below. Double-clicking the method side shows the methods implemented in this class, as well as those inherited from the superclass. When a method itself is double-clicked, it opens into another window displaying the logic.

Developers had to pay attention to routing of wiring, and to commenting inputs and outputs, to keep their diagrammatic code clean, but in general terms there was no way to avoid this sort of literal spaghetti code.

Another problem was a profusion of windows. When moving around the

Prograph workspace, the IDE generally required a new window to be opened to see the contents of methods.

Our solution should pay attention to the problems with this language. Lots of small windows can be very tedious to use, and some means of automating the layout process might be an advantage.

2.1.24 Simulink

Simulink, developed by MathWorks, is a commercial tool for modelling, simulating and analysing multi-domain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it.

2.2 Summary

There is a very wide range of visual programming languages that have been developed over the years.

One of the main issues is that they are seen as being too basic for professional programmers to use, and hence they tend to not get much attention unless there is a strong commercial incentive, such as the LabVIEW product.

Most of the examples were either for creating 3D worlds for games or for creating simple business applications by extending the usual drag-n-drop GUI construction process into the underlying actions.

The Apple Automator program comes very close to what we are proposing, perhaps too close.

Two basic styles of graphical editing seem to be applicable for our purpose. We can either create a flowchart diagram or use some sort of blocks or tiles that “clip” together. If we use the tile approach then a Nassi-Shneiderman technique might be applicable. However, the flowchart is the more commonly used technique and is, perhaps, less likely to intimidate the novice user.

The use of the dot program (or similar) to automate the layout of a flowchart might be a useful option.

Rather than pop-up dialog boxes for setting command parameters it might be better to have a semi-fixed window pane that shows the details of which ever command is selected. Alternatively, allow it to be floating or pinned according to the user's desires.

A common theme among many of the examples is the ability of the user to visualise the execution of their program – to be able to watch what is happening step by step. Our solution should include a debug mode as part of the editor that will allow this sort of visual testing.

3 UNIX Commands

This section examines the commands that are likely to be used in a scripting system. In particular we look at the styles of parameters that are passed to these commands so that our interface will be able to cope with as many different styles as possible.

My system has 3816 executable programs installed on the defined paths. It is too big a task to examine each and every one. Our goal is to select a useful subset of commands that a novice user might want in their scripts. However, we also need to look at all the different styles used for the program parameters so that other commands can be added as necessary without having to rewrite or extend the edit program.

3.1 Useful Commands

The approach to determining which are the most useful commands is to refer to books like “Linux in a Nutshell” and consider the ones that they think are worth knowing about. I will leave out commands which can better be handled by other GUI programs (such as vi) and programs that are more suited for system administration which probably should not be available for the novice user. Also I have left out networking programs, like telnet and ftp, for similar reasons.

3.1.1 Linux In a Nutshell

This first edition book is a bit dated now, and some of the commands referenced are probably not suitable for or scripting.

The useful commands include:

```
apropos, at, atq, atrm, bc, cal, cat, chgrp, chmod, chown, cksum, cmp,
colrm, column, comm, cp, cpio, cut, date, df, diff, diff3, echo, env,
ex, expand, expr, false, file, find, fmt, fold, free, grep, gzip,
gunzip, head, hostname, id, info, ispell, kill, killall, less, ln,
locate, logname, lpq, lpr, lprm, ls, man, mkdir, more, mv, nice,
paste, pidof, pr, ps, pwd, rm, rmdir, sed, sleep, sort, split, tac,
tail, tee, test, top, tr, true, unexpand, uniq, uptime, users, w, wc,
whatis, who, whoami, xargs.
```

The bash internal commands include:

```
alias, break, case, cd, continue, echo, eval, exec, exit, export, fc,
for, function, history, if, jobs, kill, let, pwd, read, return,
select, set, shift, source, test, times, trap, typeset, ulimit, umask,
unalias, unset, until, wait, while.
```

Some system administration commands that might be useful:

```
cpio, crontab, ifconfig, netstat, ping, tar, traceroute.
```

3.1.2 UNIX Tutorial

The following commands are discussed in an on-line web tutorial on UNIX:

```
ls, mkdir, cd, pwd, cp, rm, rmdir, cat, less, head, tail, grep, wc,
sort, who, man, whatis, apropos, chmod, jobs, kill, ps, df, du, gzip,
zcat, file, diff, find, history, info, echo, printenv, set, source.
```

3.1.3 Linux Commands, a Practical Reference

The following commands are briefly described:

```
apropos, man, which, time, cd, pushd, popd, alias, ls, find, locate,
grep, tar, bzip2, wget, echo, hostname, whois, netstat, sed, sort, tr,
join, paste, units, seq, cal, date, du, df, rpm, tail, lsof, ps, last,
uname, head, cat, lsusb.
```

3.1.4 Linux in a Nutshell

This list was drawn from the fifth edition of this book. I have selected those that would be useful for a novice user:

```
apropos, at, atq, atrm, basename, bc, cat, cat, chmod, chown, cksum,
cmp, column, comm, cp, cpio, crontab, cut, date, df, diff, diff3,
dirname, du, echo, env, ex, expr, false, file, find, fold, free, grep,
gzip, gunzip, head, hostname, id, ifconfig, join, kill, killall, last,
less, ln, locate, logname, lpr, lpq, lprm, lpstat, ls, lsusb, man,
mkdir, mkfifo, more, mv, netstat, nice, paste, ping, pr, ps, printenv,
pwd, rm, rpm, sed, seq, sleep, sort, split, strings, tac, tail, tar,
tee, test, time, top, touch, tr, true, uname, uniq, uptime, usleep,
vmstat, w, wall, wc, wget, whatis, which, whoami, xargs, yes.
```

3.1.5 Summary

This is a consolidation of the above lists, containing 150 commands. They may not all make it into the final list, and some will be implemented as built ins for our interpreter, but they should provide a good cross section for our analysis.

```
alias, apropos, at, atq, atrm, basename, bc, break, bzip2, cal, case,
cat, cd, chgrp, chmod, chown, cksum, cmp, colrm, column, comm,
continue, cp, cpio, crontab, cut, date, df, diff, diff3, dirname, du,
echo, env, eval, ex, exec, exit, expand, export, expr, false, fc,
file, find, fmt, fold, for, free, function, grep, gunzip, gzip, head,
history, hostname, id, if, ifconfig, info, ispell, jobs, join, kill,
killall, last, less, let, ln, locate, logname, lpq, lpr, lprm, lpstat,
ls, lsof, lsusb, man, mkdir, mkfifo, more, mv, netstat, nice, paste,
pidof, ping, popd, pr, printenv, ps, pushd, pwd, read, return, rm,
rmdir, rpm, sed, select, seq, set, shift, sleep, sort, source, split,
strings, tac, tail, tar, tee, test, time, times, top, touch, tr,
traceroute, trap, true, typeset, ulimit, umask, unalias, uname,
unexpand, uniq, units, unset, until, uptime, users, usleep, vmstat, w,
wait, wall, wc, wget, whatis, which, while, who, whoami, whois, xargs,
yes, zcat.
```

These can be classified as relating to the control of scripts, relating to the state of the computer, or relating to processing text files.

3.2 Parameters and Options

The options and parameters of a command are effectively the symbols of a command specific language. If we use language definition techniques, such as EBNF or Syntax Charts then we should be able to describe the options for any command, and also provide the user with some guidance on the correct order of the parameters and options.

Alternatives thus include:

1. A simple text entry dialog where the user types in the options and parameters. We can display the man page while they do this.
2. A palette of options specific to the command that the user selects from to build up the full set of options. Again the man page would be displayed.
3. A palette of options, with the inapplicable ones disabled, that the user selects from to build up the full set of options. The man page and a syntax diagram for the command would be displayed.

Since there are several instances of programs that generate syntax diagrams from EBNF, it seems that a possible option is to store the EBNF, perhaps within an XML file, and create the diagram on demand.

We can also use these options as an opportunity to deliver a simple version first, and then extend to the palette and EBNF in later versions. In any case, it would seem prudent to offer the simple text entry mode for all versions as it would allow a user to place any command and options that they wanted in the script without having to have a new command added to the collection first.

3.2.1 Parameter List Styles

The options and parameters for commands come in several different styles. These usually serve as either boolean flags to control the program, or as string parameters.

- Short Options. These are options or switches that are generally represented within the program as a boolean value. On the command line they are represented as a single character, possibly with a leading hyphen, or occasionally a plus sign. They may also, in some programs, be grouped together, e.g. -tz rather than -t -z.
- Long Options. These are again options or switches that are normally represented by boolean values within the program, but on the command line a more meaningful word is used. These are normally preceded by two hyphens.
- Parameter. These are string values that are passed to the program where they may be reinterpreted as numbers, or file names etc. Parameters may be prefixed by an option that indicates to the program what the parameter is to be used for.
- Name and Value. These are option and parameter pairs, usually

separated by an equals sign (=).

3.2.2 Example EBNF

The following is an example that demonstrates how EBNF can be used to describe the options and parameter for a command.

There are several alternative EBNF syntaxes so to avoid confusion we will document our particular dialect here:

```
Grammar ::= { Production } ;
Production ::= Name “::=” Symbol { Symbol } “;” ;
Symbol ::= Terminal-Symbol | Nonterminal-Symbol ;
Terminal-Symbol ::= Quotation [ “...” Quotation ] ;
Quotation :: Quote Characters Quote ;
Quote ::= '”'| “'” ;
Nonterminal-Symbol ::= Name | Choice | Option | Repetition ;
Choice ::= Symbol “|” Symbol ;
Option ::= “[“ Symbol “]” ;
Repetition ::= “{“ Symbol “}” ;
```

Using this EBNF we can define the syntax for the cal command as follows:

```
Options ::= [ Period ] [ DayOption ] [ Julian ] [ Date ] ;
Date ::= [ [ [ Day ] Month ] Year ] ;
Period ::= Single | Triple | Annual ;
Single ::= “-1” | “--one” ;
Triple ::= “-3” | “--three” ;
Annual ::= “-y” | “--year” ;
DayOption ::= Sunday | Monday ;
Sunday ::= “-s” | “--sunday” ;
Monday ::= “-m” | “--monday” ;
Julian ::= “-j” | “--julian” ;
Day ::= “1” ... “31” ;
Month ::= “1” ... “12” ;
Year ::= “1” ... “9999” ;
```

3.2.3 Example Syntax Chart

The following syntax diagram corresponds to the previous EBNF example.

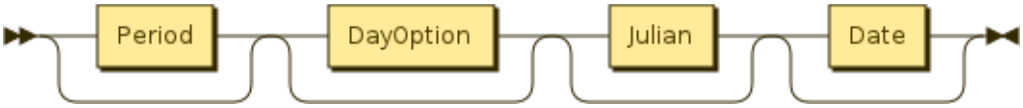


Figure 1: cal options

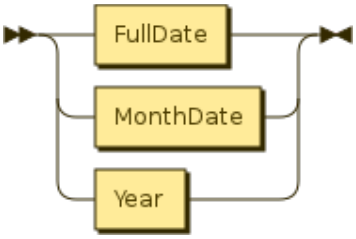


Figure 2: Date

Figure 3: FullDate

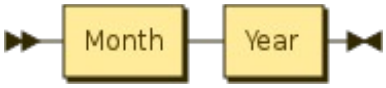


Figure 4: MonthDate

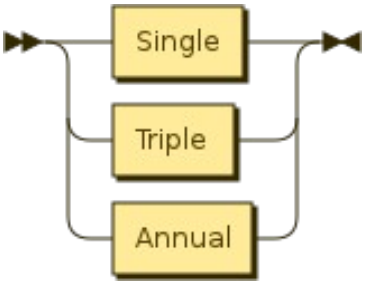


Figure 6: Period

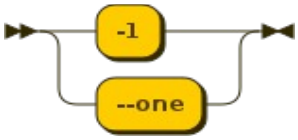


Figure 7: Single

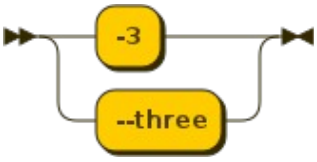


Figure 5: Triple

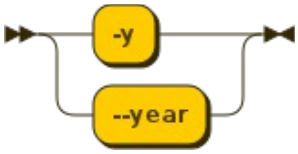


Figure 8: Annual

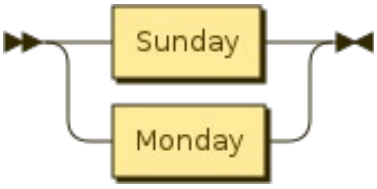


Figure 10: DayOption

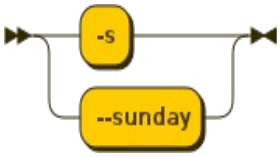


Figure 11: Sunday

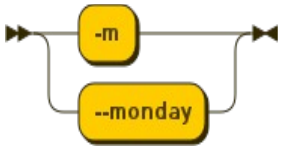


Figure 9: Monday

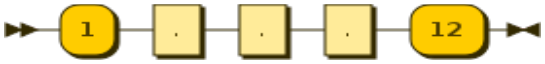


Figure 13: Month

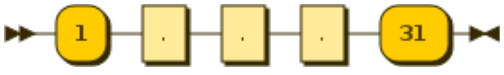


Figure 14: Day

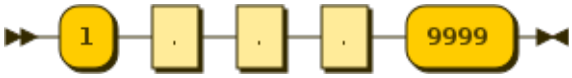


Figure 15: Year

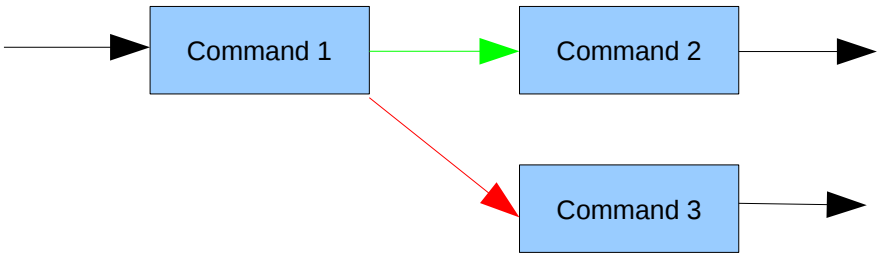
4 Visual Scripting Language

This section looks at some of the options and issues involved in the design of a visual scripting language.

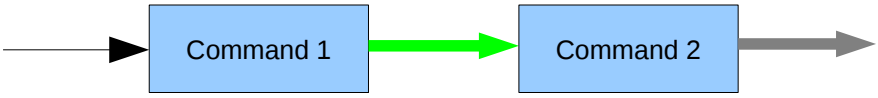
4.1 Commands

A command has several properties that we need to address. There is the flow of control within the script, the flow data between the commands via stdin, stdout, and stderr, and the exit status of the commands. Additionally it would be nice to have commands run as background processes, and hence access to the process id might be useful.

Simple flow of control is represented by a black arrow line. It states that when Command 1 completes the Command 2 should be run.

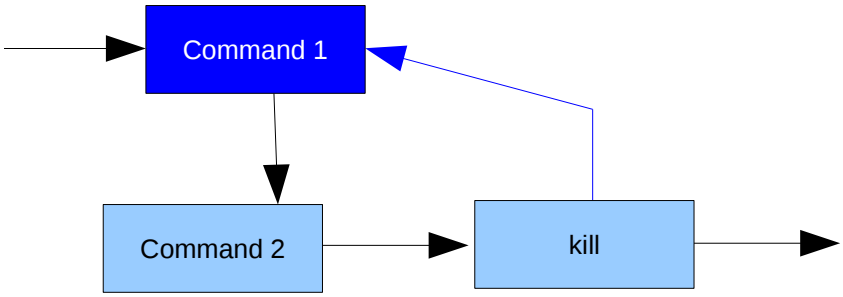


We can represent conditional execution using red (for fail) and green (for success) arrow lines. The above example executes Command 1 and if its exit status is 0 then Command 2 is executed, otherwise Command 3 is executed.



In the above diagram we show a pipeline from stdout of Command 1 connected to stdin of Command 2. Command 2 is started when data becomes available and terminates when the data reaches and end-of-file state.

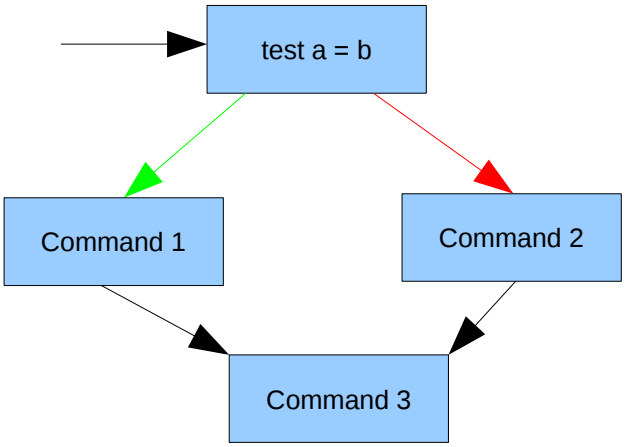
The grey arrow pipeline from Command 2 is the union of stdout and stderr for Command 2. A red arrow pipeline would represent stderr.



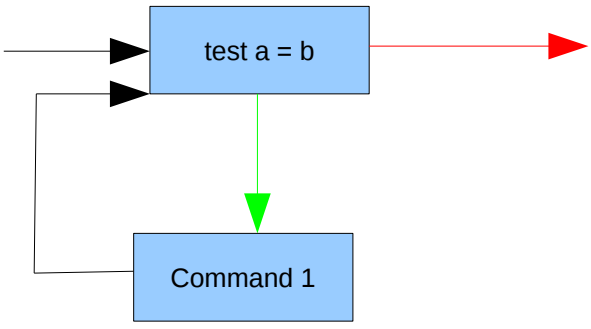
In this example Command 1 is started as a background process, indicated by the blue colour, and control is then passed to Command 2 and then to the kill command. The blue arrow line indicates the kill signal passed back to Command 1. Other signals can be specified as parameters to the kill command.

4.2 Control Structures

The commands already have an exit status and stderr vs stdout that can be used to create conditional control structures. A simple test command can be used as shown in this example where Command 1 is executed if a = b and Command 2 otherwise:

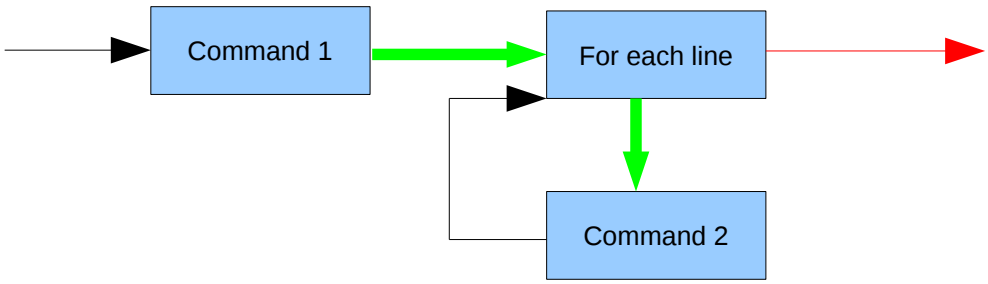


A simple loop can also be easily constructed from these components Command 1 is executed while a = b:

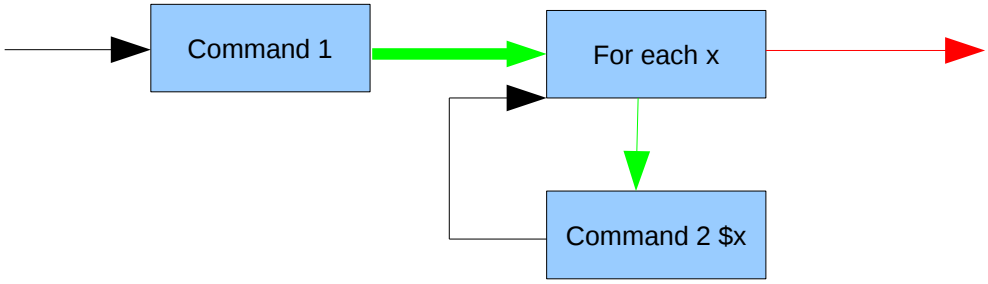


Two additional looping constructs need to be provided as built in commands. These are the “for each” and “for each line” commands that pass control to other commands before they complete themselves.

The “for each line” command would be used like this where Command 2 (probably the “read” command – see below) is executed once for each line of output produced by Command 1:

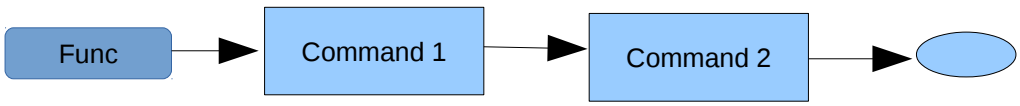


The “for each” command is similar, but executes its commands for each symbol (separated by white space):

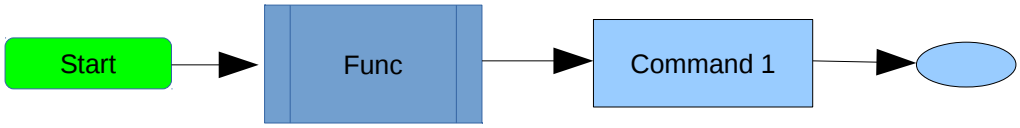


4.3 Functions

We need to group commands together so that they can be triggered from the menu or the drop action. It would also be useful to be able to reuse some groups of commands. We will therefore provide a function construct:



We can then refer to the function from within another function using the function symbol:



A function can also be run as a background task, which would be indicated by a blue colour (similar to background commands). Functions can flagged for entry into the runtime menu, as indicated by the “Start” function in the above example.

4.4 Data Objects

Any useful language needs to store and retrieve data. The following symbols will represent data objects:

For data values as constants or variables we will use the “internal storage” symbols with a yellow colour to indicate a constant. The symbol looks a bit like a fragment of a spreadsheet to reinforce the concept. The above diagram shows the constant being “echoed” into the variable.

Variable can be referenced in the parameter list of commands by preceding their names with a $ symbol.

Multiple assignments are possible, as shown above. This is equivalent of the bash command “read”.

It might be useful to insist that variables have exactly one pipeline connection.

Variables will be global within a script. The user can devise a naming convention to localise their variables if required. It will probably be implemented as an associative array with all values stored as strings.

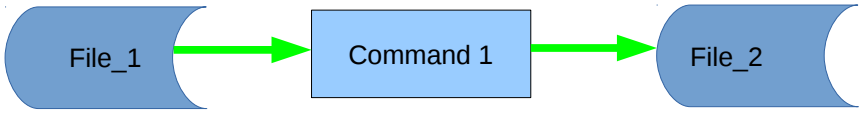
Functions will imply parameters for the functions, and thus we will need a stack, in fact a stack for each thread within the script. The usual numeric positional notation will be used for parameters, $1, $2, etc.

4.5 Calculations

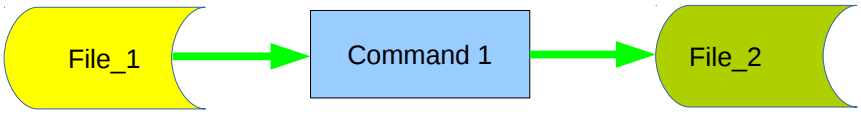
For scripting we usually only need a few simple calculations, such as counting. Hence we will include a built in command similar to the bash “let” command. It will do assignments and simple single operator calculations.

4.6 Files

A script will normally need to access a user's files. It will also often need temporary files and in-line files.



The above example shows Command 1 being used to process a user's file and having the results stored in another of her files.



Here, File_1 is an in-line file and File_2 is temporary. Temporary files will be deleted on exit. The editor will display a text dialog so that the contents of File_1 can be set. The in-line file may contain references to variables that will be expanded when the file is used.

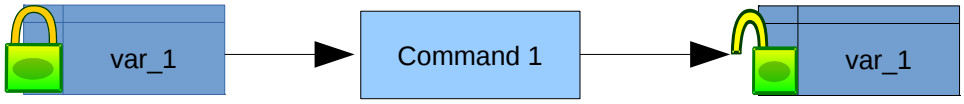
4.7 User Interfaces

The zenity command can be used to create user interface components. A couple of pre-packaged zenity commands might be useful for getting and displaying text vie pipelines.

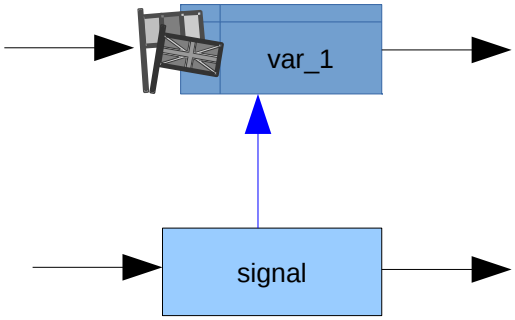
A special variable might be needed to handle the names of files that are dropped onto the interpreter GUI.

4.8 Process Synchronisation

A script will be able to run commands and functions as background tasks. The runtime will also have a menu system that will allow the user to initiate multiple functions, which may end up running simultaneously. Hence we need some means of synchronising actions so that the user can control what happens.



A lock symbol can be used to indicate that a variable is a mutex. A mutex must be granted before the process can continue, and can only be held by one thread at a time.



A flag symbol can be used to indicate that a variable is a semaphore. A semaphore is automatically decreased by one unless its value would be less than zero in which case processing is paused until a signal increases its value.

4.9 User Configuration

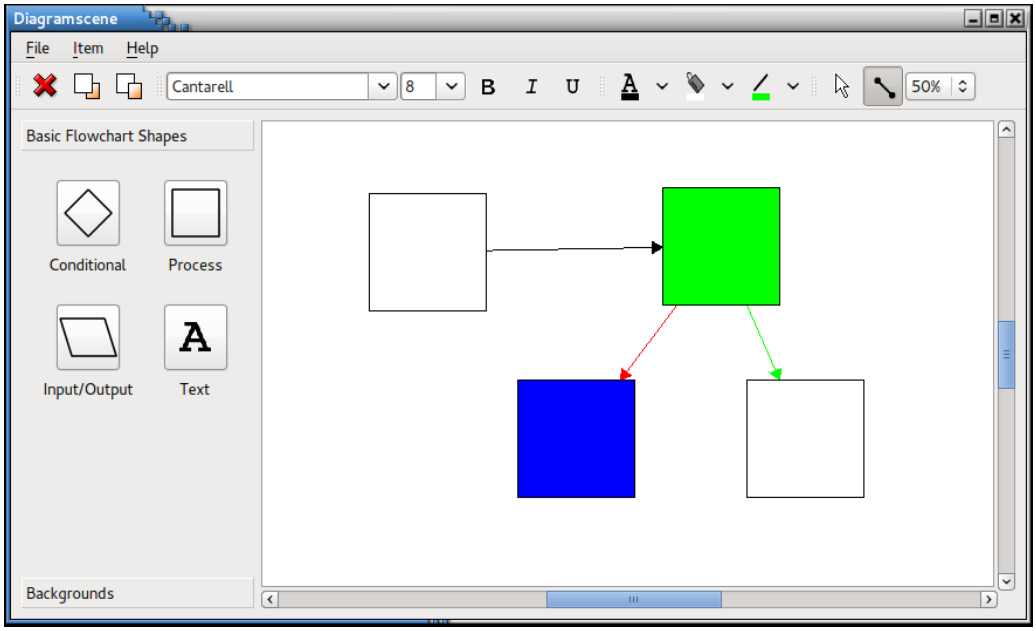
The colours shown in the preceding examples are an indication of the intent of the design. Since there are some users that may have more or less severe colour blindness we need to provide a means of allowing the users to set their preferred colour scheme, or to use patterns instead of colours.

Since this is a graphical application there will not be a requirement to support multiple languages in the diagrams. However, the help text will need to fully support internationalisation.

The user interface should be easily configurable to accommodate different screen sizes and aspects. For example, a user may want to display two flowcharts side by side if there are synchronisation flows between them.

4.10 Feasibility Study

An example program has been slightly adapted to demonstrate the viability of creating a graphical editing program.



5 Testing and Debugging

5.1 Observation

The user needs to be able to run subsections of the script, i.e. to define start and stop points. To support this it might also be useful to be able to save and restore the variables, and perhaps also the files that are in use.

Stepping through the script, one command at a time, is also a time honoured method of debugging programs. We might also consider a “slow motion” option where a small sleep step would be introduced between each command execution.

A stop point can be indicated by changing a flow symbol into one that includes some sort of “break” indication. The user interface can then be used to single step execution, or resume (either at full speed or slowly). A small icon can be used to represent the current execution position, and this can be dragged to other points on the diagram to restart and retry execution.

An ability to observe the variables, and file contents, as the program is executed would seem to be a requirement. While the value of a variable can be easily displayed, files might be a larger challenge. We might, for example, show the file size, or the last few lines if it is a text file, as the processing is performed.

The currently executing command, or the flow line pointing to the next to execute command, should be highlighted. This may be several objects if there are simultaneous threads running. Variables that are being referenced, or are about to be referenced, can also be highlighted.

The global variables can be displayed as a table. The stacks can be shown as a table showing the top few elements. Where there are multiple threads we can switch between stacks according to which thread is being viewed.

5.2 Testing

To perform tests the user may want to be able to change the values in variables before running a fragment of the script. Hence we should allow the displayed variables to be modified by the user.

Files can best be handled by taking snapshots and reverting to these backups when the fragment is restarted. The user can always use an editor to modify the files, or swap in alternative saved ones if they want.

6 Desktop Integration

This section documents the way that graphical programs are installed into a modern Linux desktop.

The specifications for the desktop are provided (for our purposes) by three documents from freedesktop.org.

- desktop-entry-spec-1.0.html which documents the contents of desktop files which are used to define an entry.
- menu-spec-1.0.html which documents the way that the menu structure is defined.
- basedir-spec-0.6.html which defines where files should be placed and how they should be located.

The editor will be responsible for creating suitable desktop files and menu entries and placing the files appropriately.

The editor should also allow the user to remove scripts that they no longer need. In doing so it should remove all associated desktop files and menu entries.

7 Command Search Capability

This section investigates the options and alternatives for enabling the user to easily find suitable commands.

7.1 Finding Commands

The simplest approach is to provide access to the “apropos” command, displaying the results into a text dialog.

We will be providing a set of prepared commands that should be available from a selection list. It should be possible to search for strings that occur in the short descriptions for these commands.

We can add tags (and allow the user to add their own tags) to the prepared commands and allow the user to select from the list of tags to find suitable commands. Allowing a user to select either the intersection or the union of sets of tags would allow the search to be refined. A hierarchical structure for the tags might be useful.

7.2 Help on Commands

The editor should allow the user to pop up a text dialog showing either the man page or the info page for the selected command.

These help pages are not the most friendly things for the novice script writer, but re-writing them seems like too much work, and it might be best if they eventually learnt to use these resources.

8 Interpreter

This section examines the issues surrounding the run time interpreter.

8.1 Structure

The core structure of the interpreter will be used by both the run-time and the editor (for testing). Placing it into a shared library might therefore be a good idea.

The script will most likely be provided as an XML file which will be unpacked into an internal representation of the script. Thread objects will then navigate this structure and initiate the system commands as necessary (or execute built-in commands such as “cd”).

The program will include a menu to which the scripts are attached. Options to restart and stop execution should be included.

The program should include an optional target for drag-n-drop of file names.

Text areas for accepting user input and displaying stdout and stderr will be optionally provided if the script specifies them.

8.2 Security

Normally when we download scripts off the internet they are confined to the browser sandbox and are not allowed (we hope) to access or modify the user's files (without explicit approval). However the scripts for a visual scripting language will be just ordinary XML files which could easily be downloaded from random places on the net and easily installed and run by an ordinary user. This could quickly become a bit of a problem.

One possible solution is to cryptographically sign the scripts. The editor would sign a script with the user's private key. The editor and runtime would only open or run scripts using the user's public key, or perhaps one of another set of public keys controlled by the system administrator. The latter set would allow scripts to be used across a site, or to be installed from the distribution's repository.

An administrator could make a script more widely available by temporarily accepting the developer's public key into the set of allowed public keys, opening the script and then saving it using the site wide private key.

9 Conclusion

There is nothing in this project that is new or unusual. It should prove to be a straightforward development project.

If brought to a successful conclusion the project should provide users with a useful tool for automating some of their more repetitive tasks.

9.1 Development Model

There is no large amount of infrastructure that needs to be built, nor is there a shifting set of requirements. A test driven iterative development model is probably the best approach.

Later experience has resulted in an incremental development model being adopted since the project is taking a lot more effort than originally expected.

Appendix A - Glossary

Term	Explanation
Administrator	A person responsible for ensuring that a computer system performs correctly. Usually an administrator will have additional privileges that allow them to perform tasks that an ordinary user is not allowed to.
CDE	Common Desktop Environment. An early graphical user interface shell that was standard on many UNIX systems.
Debugging	The process of determining the reason for a program not behaving as desired, and applying an appropriate fix.
Dialog	A window which is temporarily displayed by a GUI program, usually to allow the user to enter some information, or to advice of some change within the program.
dot	A computer program that calculates the layout of graphical symbols on a diagram.
Drag-n-drop	A technique that allows a user to drag a symbol (including text) from one program to another.
EBNF	Extended Backus–Naur Form. A language for specifying languages.
Editor	A program used to enter or modify the files used by other programs.
Flow Chart	A diagram that shows the sequence of events in some process, usually including special symbols for decision.
GUI	Graphical User Interface. A means of interacting with a computer system using windows, icons, menus and a pointing device (mouse).
IDE	Integrated Development Environment. A computer program designed to facilitate the creation of other computer programs. The VICI Editor might be considered to be a simple IDE.
info page	An help file that explains the purpose and usage of UNIX commands.
Java	An object oriented programming language which is normally executed by an interpreter.
JVM	Java Virtual Machine. A computer program which executes a Java program.
Linux	A UNIX like operating system for computers using free and open source development techniques.

Term	Explanation
man page	An help file that explains the purpose and usage of UNIX commands
Nassi-Shneiderman	A diagramming technique (similar to flow charts) that uses overlapping rectangles to show the structure and control of the process.
Object Oriented	A programming technique that associates data items and the functions that use them into packages with well defined interfaces.
OS	Operating System. The set of computer programs that provide a standard interface between the application programs and the hardware of the computer.
Parameter	Information passed to a computer program as it starts. This is normally used to modify the actions of the program.
Paths	The set of directories that a shell will examine when attempting to find a command.
Program	A set of instructions that control the actions of a computer.
Script	A set of commands in a file that is used by an interpreter.
Shell	A computer program which accepts user input, either directly from the keyboard or indirectly from a file, and starts other programs in response.
SourceForge	A web site where projects can be placed so that multiple developers can work on them.
stderr	The default error file for a program. This is typically connected to a terminal session so that the user can see any error messages generated by the program.
stdin	The default input file for a program. This is typically connected to the keyboard so that the user's input is fed to the program.
stdout	The default output file for a program. This is typically connected to a terminal session so that the user may see the results from the program.
Syntax Chart	A diagram describing the allowed syntax for a particular language.
Tag	An identifier that allows other objects to be classified in some way.
Testing	The process of determining if a program performs as it was intended to.

Term	Explanation
UNIX	An operating system for computers.
User	A person using a computer, usually to perform some business or personal task.
VICI	A project to provide a graphical method for creating an interpreted script.
Visual Language	A language that uses icons and other symbols rather than text to convey its meaning.
Wikipedia	An on-line encyclopaedia.
XML	Extensible Markup Language. A format for a data file which is intended to be self describing.