`tini.flow`

A Python- and Shell-based Workflow Tool

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

—

Software Architecture & Design

presented by

Titus von Köller

under the supervision of

Max Schöfmann

co-supervised by

Prof. Cesare Pautasso

January 2018

I

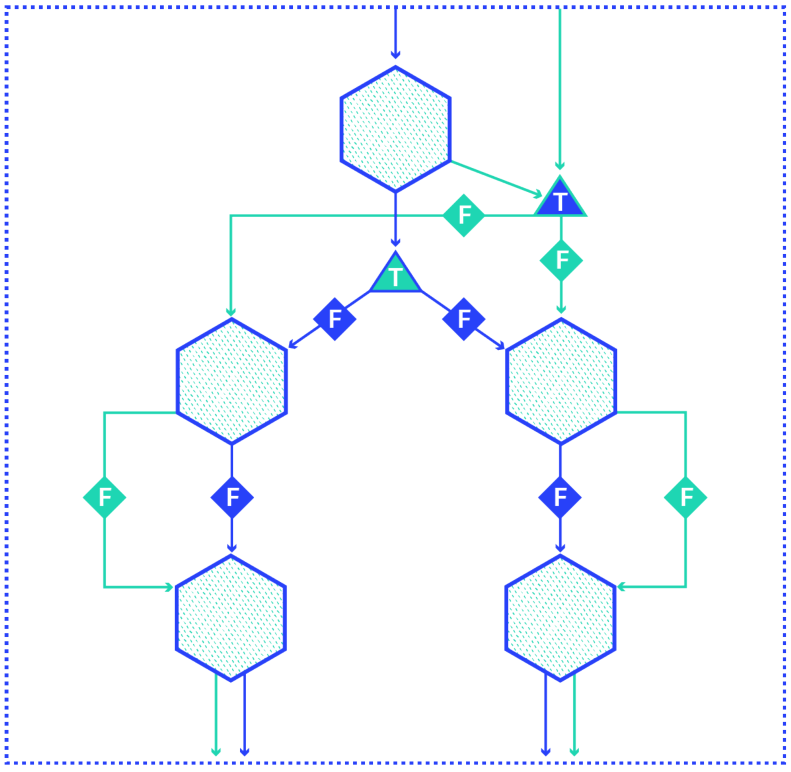


Figure 1. a `tini.flow` control & data flow graph: data edges in blue, control in green, **T**-triangle for tee, and **F**-diamond for FIFO.

[illegible]

Titus von Köller
Lugano, 30 January 2018

to Tini, who has been my partner and best friend through this difficult journey, without whom I would not have made it to the other side

to my parents, who have stood up for me countless times through the years and through their love and dedication enabled me in getting where I am today

to James and Tim who were my nerd friends throughout this journey and significantly contributed to my world of thought

to Container Solutions, who enabled me in pursueing this topic

In the long run, the greatest impact and the greatest “reuse” of software comes from building on what has been learned, so that one is not forced to relearn things that should already be known. Unix provides a variety of lessons that we can build on for the future.

Some of these are specific lessons about the design, implementation and organization of systems: the importance of a clean simple file system; the notions of pipes and I/O redirection; the value of a programmable command interpreter. These are not all new with Unix, of course, but they seem to have been combined in it most effectively.

Some lessons are more general. For example, it is now amply demonstrated that high-level languages are the only sensible way to implement systems. It is also evident that systems do not have to be big to be useful; indeed, quite the contrary is often true.

Finally, the experience with Unix shows that with a good operating system and a good set of tools, it is possible to make extensive reuse of programs and technology, instead of starting over from nothing with each new task.

The Unix System and Software Reusability
Brian W. Kernighan (1984)

Abstract

This thesis is about composability.

Composability is a broad topic in software design, and this thesis focuses on a specific case of composable programs.

Workflow systems are systems that decompose large human-defined tasks into small units that map to semantic entities in the context of the overall task. They include a first-class representation of these entities, usually in the form a graph, where nodes represent tasks and edges represent data or control flow connectivity between these entities.

The shell is one of the most popular tools for building ad hoc composable programs where the units of composition are Unix processes. It has withstood the test of time, and, in spite of its limitations, it contains all of the necessary functionality needed to implement a workflow system.

This thesis introduces a workflow system called `tini.flow` that is built upon the basic constructs of the shell. Great care has been taken in the design and implementation of `tini.flow` to address the limitations of the shell and properly extend them to a broader class of problems. As a system focused on composability, `tini.flow` also seeks to provide user affordances to provide a robust, extensible system for building workflows.

`tini.flow` proves that it is possible to extend the shell to a high-quality workflow system, thereby addressing the limitations of the shell, concerns for enabling good usability, and creating intuitive affordances, that can be further extended in the future.

Keywords: Composability, Python, shell, workflow systems, data flow, graphs

Contents

Figures

Listings

Chapter 1

Introduction

Every piece of knowledge must have
a single, unambiguous, authoritative
representation within a system

— *DRY principle*, stated by
Hunt and Thomas [2000]

This thesis is about *composability*.

Composability is a broad term that covers important considerations in the human construction of software projects. It goes beyond simple software reuse. Acknowledging the benefits of software reuse is easy: who wants to do the same work twice or more at no additional gain?

But composability is about the overall complexity of large software projects and how to decompose that complexity into human-manageable pieces. It is about the human experience of building software, and it focuses on affordances provided to the programmer to make decomposition and recomposition practical and feasible.

Fundamentally, good composability in a software system leads to lower costs of maintenance, greater potential and likelihood for innovation, and a counteracting force to the exponential growth of complexity that is an inevitability of large software projects.

1.1 Composability

The field of software development is an industrial, applied field: it generally concerns itself with the creation of tools that implement some business process or scientific computation.

Software development is built upon computer science, using theoretical results to drive practical results. However, just as bridge-building relies on mechanical engineering, which relies on physics, a competent bridge-builder worries about things beyond just force equations. They worry about non-theoretical costs such as actual monetary cost, organizational cost, cost of time and effort, cost of understanding, or general cost of complexity.

Similarly, the field of software design and architecture is split between computer-facing concerns, like performance, optimization, compilers, parsing, and human-facing concerns, like reducing mental overhead, maximizing productivity, quality control, and overall correctness.

A common refrain among industrial software developers is that the job is often no more theoretically rich than plugging A into B. Their work is about composing existing entities and managing the compositions themselves. In these cases, concerns related to composability often dominate the software development process. Similarly, a common theme in industrial software programming is the construction and management of workflow systems.

Composability means many things, but fundamentally it means mechanisms, approaches, tools, and theories behind the creation of reusable components. Every programming language and paradigm offers its own techniques and mechanisms for composing individual units of work.

These can be judged on the basis of how they map to organizational structure – in other words, on how large efforts can be broken up into small efforts so that they fit into given job roles of an organization. They can also be judged on the basis of the degree of complexity they introduce or remove, but composing larger entities from smaller entities almost always involves some overlap in functionality.

Coming from another perspective, composability may also be judged on the basis of convenience of implementation and mental overhead. Encapsulating logical units in a meaningful way involving a sensical hierarchy goes a long way toward enabling the developer to keep the right things in mind at the same time when implementing new functionality or refactoring existing code.

It is not always the case that the monolith is less efficient in terms of cost of execution or cost of construction than a more composable approach. It is also not always convenient to factor out functionality where this would take significant effort or where it would add mental overhead from not having all pieces necessary for understanding the system in one place. For example, having to look at a scattered collection of tiny functions spread across numerous files in order to reconstruct the definition of a simple task may be inferior to just having the code all in one place.

Clearly, composability is not simply about reuse or elimination of duplicated work. Decomposing monolithic systems into composable parts often introduces additional work and additional duplication. In a monolithic system, less attention needs to be paid to internal boundaries, as it is assumed that changes to the system can freely change all internal details in a coordinated fashion. In a decomposed system special attention has to be paid at the boundaries of every component and their interfaces may be harder to change in a coordinated fashion. For example, in a service oriented architecture, information passing between components cannot pass directly, it must be serialized, transmitted, and unserialized, which requires additional work and additional code.

The advantages of decomposition in this case are better understandability of the system, since components can be viewed and tested in isolation, and better distribution of effort, allowing multiple programmers or groups of programmers to build very large systems in parallel. Unlike in a monolithic system, the dependencies in their work can be clearly encapsulated in the structure of the components.

Such a system can also be easier to maintain in the long term and more adaptable to changing requirements and changes in third-party dependencies. Components of the system can be better isolated and consequences of changes can be prevented from propagating through to the entire system.

Administration of the system can also be easier, as individual components can be monitored or updated in isolation. This could lead to greater robustness as individual components could fail without that failure necessarily cascading throughout the system.

Of course decomposition of a monolithic system into individual components is highly context dependent. There may exist many decompositions of a system at many levels of granularity, and these decompositions may not offer every advantage or solve every one of the above problems.

Because there are many systems in which decomposition increases overall effort and because there may be numerous competing decompositions of a monolith, the topic of composability is often measured in subjective terms, relative to the environment in which the software is developed.

Workflow systems focus on the human-facing effects of decomposing systems. Their composability is judged on their ability to recompose individual entities into new systems, and the difficulty of this process is tied to the human aspects of the software development process.

Often, workflow systems are the result of a software development process starting with a monolithic system. Programmers decompose this system into smaller pieces to improve their ability to understand the system and to improve component reuse. They reduce the monolith into small orthogonal, independent components that can now be recombined to build new tools and solve new tasks and implement new workflows.

In other cases, decomposition occurs in the design process. A programmer given a business task will decompose that task into logical units to make use of previous work. The programmer will try to predict how these units might be applicable to future tasks. The programmer will then write, verify, and test these units in isolation, then compose them into a singular system that accomplishes the original task. They may develop or employ a workflow system to ease in future recompositions of these components.

If the resulting units of a decomposition include the functionality of existing command line utilities, then the composition of these units may involve the writing of a shell script.

Thus, the programming paradigms introduced in the shell are closely tied to the idea of workflow-style composition and focus on composability to address human-facing concerns in the software development process.

1.2 What Is a Workflow?

This thesis considers a certain class of workflow problems. It introduces a tool for modeling this class of workflows. The basic mechanisms that comprise this tool are derived from shell programming. Shell programming has historically been used for stream-data processing and has proven useful in the construction of a certain class of workflows, those which could be called scientific workflows.

The fundamental grammar of the shell includes mechanisms for workflow-style composition, and include piping, file redirection, and process control. These specific mechanisms are a unique feature of shell environments and are rarely found in traditional programming languages and environments. Often, where complex triggering and control decisions are required as part of the workflow, these mechanisms are supplemented by control flow constructs. The control flow constructs are often derived from traditional programming languages – if-else clauses, while and for loops, etc. – and do not resemble *native* shell syntax.

The tool introduced in this thesis, called `tini.flow`, extends the fundamental mechanisms of the shell to address complex control flow and triggering. Rather than clumsily patching in traditional control flow syntax from non-shell languages, `tini.flow` tries to repurpose these existing mechanisms to focus on complex composition of processes.

The term process or system-level process refers to any string which can be passed to the POSIX `exec` system call to spawn a new OS-level process. This also includes any string which can be passed to `$SHELL -c`.

When discussing processes, we want to unify ELF-style binaries, `#!` shebang driven scripts, shell built-ins, and shell functions. To most users, these various constructs are considered equivalent, and many users may not even know what commands at their prompts are actually scripts, built-ins or functions.

Similarly, the term command line more broadly refers to any string that can be typed at a shell command prompt to perform some computation.

We use this terminology to distinguish single processes from more complex shell expressions which may involve multiple processes. Typically, we will distinguish between a process and a command line, in that a command line may involve calling multiple processes and may do its own piping and redirection internally.

`tini.flow` focuses on exploring workflow-style composition. While `tini.flow` might be useable for scientific data flows, we will mostly concern ourselves with compositional concerns that one might find in a so-called business workflow.

Tools for scientific workflows often concern themselves with optimization[Barker and Van Hemert, 2007], e.g. representing calculation dependencies as a graph to eliminate redundant computations or facilitate fast real-time recomputation, or memory use. For example, they may focus on making large computations possible through stream and map-reduce methodologies or they concern themselves with operational details of running these computations, such as abstracted execution systems that allow for quick local prototyping and easy production deployment to GPUs or supercomputer clusters.

These concerns are outside of the scope of this thesis. We will focus primarily on compositional concerns and motivate these with examples from the class of non-scientific workflows. We will denote these workflows as business workflows to distinguish them from the scientific variant. That these are called business workflows does not mean to imply that these workflows only have an economic meaning. Instead, we borrow this contrast between scientific workflows and business workflows from the literature:

> As the validation of scientific hypotheses depend on experimental data, scientific workflow tends to have an execution model that is data-flow-oriented, where as business workflow places an emphasis on control-flow patterns and events.

The Workflow Management Coalition (WMC) [Hollingsworth and Hampshire, 1995] characterized a *workflow* as the full or partial facilitation or automation of a business process. This is accomplished by passing documents, information, or tasks between processing entities for action, following a set of strict procedural rules. These automation workflows are often highly non-linear.

The WMC further elaborates on *workflow management systems* as encompassing the definition, management, and execution of such workflows by software that is driven by the computational representations of the actual workflow logic. `tini.flow` is designed to be such a workflow management system.

Fundamentally, in a business workflow we compose computational entities, automation entities, and reporting entities in order to accomplish some business process or task. These workflows will include data flows for the information they process and control flows for the conditional logic necessary to drive the processing.

When composing these subtasks in a human-meaningful goal, we call that a business process and imbue these otherwise very mechanical actions with some meaning. In this sense, the term business process is used as a term parallel to the common phrase business logic.

A business process is a process that lives in the world of human context, is written with regards to often non-mathematical or pseudo-mathematical human rules and guidelines, and accomplishes some task whose task is not wholly theoretical in nature.

A tool like grep is built with the purpose of searching through text files for specified patterns. At its core, its functionality is algorithmic in nature and concerns itself with the most space and time efficient way to perform these searches.

In the BSD-style, tools like grep often end shortly after this algorithmic core, and tools like BSD grep have very few command line options because their primary focus is accomplishing this direct mechanical goal.

In the GNU-style, tools like grep have much larger implementations because they surround this algorithmic core with a variety of flags and options, to adapt this mechanical process to a human context of meaning and intention.

At a higher level, a tool like grep becomes integrated into some process that accomplishes a goal that lives within this human context such as *search through my thesis for references* in order to build a bibliography so that the thesis can be submitted and accepted by the committee.

This latter adaptation of the algorithmic core to the human context and the composition of this tool into a broader goal highlights the distinction between the algorithm and business parts of grep.

The Unix style encourages the creation of these tools and composition of them into business processes. Commonly, these tools are created when decomposing a single business process into discrete units that have a singular and clear meaning, then implementing those units according to a specification derived from this meaning. These tools are written with ease of recomposition in mind, and are the fundamental units of the business workflows we will consider.

1.3 Workflows as Graphs

When discussing workflows, we will conceptualize them in the form of graphs.

We will discover that the modelling of workflows as graphs is a fundamental property of workflows.

A brief introduction to the terminology we will use: *Graphs* are collections of nodes and edges. *Nodes* can represent any discrete entity. *Edges* are tuples of two nodes. Edges are unique and can have accompanying edge data. Nodes are unique and can have accompanying node data. Edge data may represent how edges are connected – e.g., they may map to structures like pipes – and node data may represent the entities that are being connected – e.g., they may may to structures like processes.

Directed graphs are graphs where the edge (a, b) is distinct from the edge (b, a). *Directed acyclic graphs* (DAGs) are a class of directed graph in which there are no cycles: that is, there are no paths which involve the revisiting of a node. *Trees* refer to DAGs with one parent, in which each descendant node has only one in-edge.

1.4 Data Flow vs. Control Flow

The following command line is a simple data flow problem:

```
$ seq 1 15 | tee file | grep 3 | xargs -n1 echo 3s
```

It begins with one data source produced by `seq` and filters and modifies it to produce a final output. The last element of the pipeline is just a modification of the output, but the `xargs` could use its input to drive any arbitrary command. This could be viewed as a rudimentary form of data-driven automation.

This data pipeline is not that far from what we might call a workflow – it contains nodes and edges, arranged by the user to encode the solution to some goal. This use of the shell for building workflows has clear limits, which will be discussed in later chapters.

One major limit is that it primarily encodes data flow connectivity and does not encode control flow connectivity.

Figure 1.1 shows three graphs that intuitively suggest the idea of a workflow. However, without labels on the edges, the two branching graphs could have wildly different meaning.

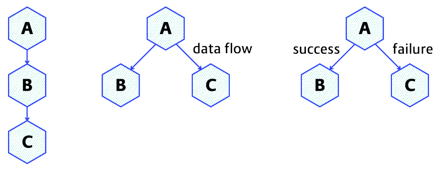


Figure 1.1. illustrating a pipeline, data flow, and workflow – difference basically just the labelling of the edges and nodes

In one variant, the edges mean data connectivity, and this could be called a data flow. In the other variant, the edges mean control connectivity, and this could be called a control flow. In the data flow, the edges indicate the flow of data from process to process. In the control flow, the edges indicate the next action to perform.

Typically, our workflows will have a combination of data flow and control flow, and our graphs must be able to accommodate both kinds of edges.

When we talk about workflow systems, we talk about systems that practically implement both of these tasks.

While shell scripting does implement both data and control flow, it does not unify their syntax. Still, it is a good starting point for considering workflow systems.

Why *the shell*? The shell is ubiquitous.

For better or worse, it is one of the world's most popular and most successful programming environments. It is also relatively simple from a theoretical perspective, though this simplicity sometimes creates accidental complexity. The shell is also probably the simplest concurrent programming environment, and is one of the few concurrent programming environments wherein users don't even have to know about concurrency to take advantage of it.

Shell piping is a mechanism that leads to high discoverability as it lends itself to interactive use and iterative development. Piping syntax is concise, visual, and directly encodes connectivity information. Writing to text-based stdout and reading from stdin forms a simple API that allows for easy collaboration between programs.

The shell enables simple, easy, and correct prototyping, which then also often even works well in production.

1.4.1 Levels of Granularity in Composition

A workflow is a compositional system.

We may use the term processes to refer to the units of composition in a workflow system. This term can then be interpreted narrowly or broadly. Broadly, processes could be the primary entity of composition of the system, implying nothing more. Narrowly, processes could be Unix processes, as opposed to functions in a programming languages, or services in a service-oriented architecture.

The latter distinction implies that a workflow does not necessarily have to be a composition of entities resembling Unix processes. A workflow can compose entities at any level of granularity and these entities, though they may casually be described as processes, may be taken from a variety of different computational constructs.

Regardless of the constructs chosen, a workflow will often define a basemost atomic unit, which cannot be further decomposed. It may also define larger groupings that serve to encapsulate or provide structure.

The choice of these units does not determine whether a compositional system built upon them is or is not a workflow. What makes these compositions workflows is more tightly tied to the modeling of the composition as a graph and first-class representation of composition.

In other words, a workflow is a composition, wherein the units of composition are clearly demarcated, the connections of these units are clearly demarcated, and both the units and the connections (i.e. the nodes and edges of the graph) have some first-class representation.

This representation allows the encoding of structures necessary to answer questions such as what stage of the workflow is currently executing, what task is currently executing, whether the execution of these tasks can be automatically parallelized, or what are the dependencies of a task. These questions are often primarily meaningful to a human being trying to manage the human goal and human process encoded by this computational structure.

Therefore, a workflow systems could be developed, which composes functions from a programming language. In fact, numerous such systems exist, e.g., `gulp` or `celery`. Another workflow system could be developed which composes services. This might look very similar to container orchestration platforms like Kubernetes.

Not only is the choice of compositional unit flexible, a workflow system may span many levels of granularity in common operation. What is considered to be the basemost unit and what is considered a grouping of units can be highly context-driven.

For example, consider containers such as Docker. Containers and containerization are frequently discussed in relation to workflow style composition. But their meaning to a workflow system is not fixed. In some cases, a container might be a large circle drawn around the entire workflow and may represent the largest possible grouping.

This would be the case if the container were used for deployment purposes. The workflow system itself could be a scripting environment like Python and the units of composition could be Python functions and classes. The container would serve only to isolate software dependencies.

In other cases, the container might be a grouping above or below the most common unit of composition. This would be the case in a service-oriented architecture, where a single service might be built out of multiple containers or a single container may encapsulate an entire service.

Despite the container representing a grouping at different levels of granularity, it is still meaningful to discuss it in the context of the workflow system.

That the container can have dramatically different OS-level characteristics and underlying mechanics on different platforms but still be a common element when describing a compositional system, suggests that this consideration of granularity for a workflow system has meaning primarily to the human beings developing the system. It doesn't serve the computer executing the workflow; it helps the human being who is writing the workflow to manage the complexity of the software development process.

Considering alternatives to process-style composition, one advantage of composing at the granularity of functions is that the programming language environment carries the weight of almost all the overhead, such as memory management, allowing the compositional units to focus only on the core task they are completing.

Disadvantages include restricting the workflow system to a single version of a single programming language environment, and a counter-point to the lower-overhead communication between units is that these connections do not have to be as rigidly defined, resulting in a higher likelihood of component port mismatch.

Alternatively, an advantage of composing at the higher level of individual services is that it is a better match of the compositional boundaries of the human organization developing the system. Service-oriented architectures are popular in the case of large distributed organizations, where subteams map to services and where teams minimize n^2 communication and coordination, by defining common interfaces between their services, e.g., REST, and rigid contracts for each individual service.

A disadvantage in this case is a higher degree of potential duplication, as services may inevitably overlap in functionality. Also, the communication and coordination problem may be more fundamental to an organization and cannot be solved with just an software change.

Somewhere between these two levels of granularity, functions and services, sits composition of Unix processes. This form of composition is a well established approach with deep history, and this thesis will focus on extending its use.

In later chapters, the disadvantages of shell-based composition of Unix processes will be detailed, many of which arise from the fundamentally text-based nature of this style of composition. The chapters will also cover the many advantages of this compositional approach. Simply, we can rely on hundreds of processes, written by thousands of developers, over decades of time, in dozens of languages. (This enormous library of compatible software almost achieves the ideal of universal interoperability!)

1.5 Command Interfaces in General

The shell is one of the primary ways to interact with a Unix-style operating system. It natively includes affordances for encoding workflow-style composability.

Kampe [2012] states that ninety-nine percent of the literature on human interface engineering is focused on graphical user interfaces, opposed to the literature that discusses purely text-based interfaces. He hypothesizes that this is due to the information density that graphics can convey – akin to the bon mot "a picture means more than a thousand words".

This makes sense, because throughout most of evolutionary history, processing visual information did not include any symbolic characters whatsoever. The human brain and that of its predecessors has been highly optimized by evolution to process and act on visual information, mostly regarding patterns that can be found in nature and that were meaningful for survival at the earlier stages of development, before the genesis of civilization [Gibson, 2014].

Command line interfaces (CLIs) refer to the general practice of interacting with the computer via commands. This term is most commonly used in reference to the use of a shell within a terminal application. Within the shell, one can interact with the computer in a text-based manner via commands. The way with which the commands are structured, the syntax around their specific usage, and generally the possibilities of composition define the way the human may interface with the computer based on text – the command line interface.

However, as noted by Wilson [2015] in his post on the return of CLIs, any form of interaction with a computer in the style of a verbal command grammar, even if it is not text-based, fits the more general definition of a command line user interface.

One example is the use of *bangs* within the DuckDuckGo search engine: `!songfacts disco duck` for example will give the computer system behind DuckDuckGo the command to forward you directly to the songfacts.com website's entry for *Disco Duck* by *Rick Dees & his Cast of Idiots*, giving you access to background information, lyrics, and a YouTube video of the song. In the same way, `!gi command line interface` will take you to their competition Google's image search and show you screenshots of command line interfaces.

Another great example for a command syntax in an everyday product is the use of the `@` (direct at this user) and # (tag with this keyword) syntax in social media. Twitter is a good example for this. The command syntax is a way for the human to command the computer system behind Twitter on how to construct the graphs that lay behind the functionality of their product. Other examples include voice assistants like Alexa and chatbots.

Even though it is interesting to look at CLIs from this viewpoint, this thesis will from here on focus exclusively on CLIs in the shell, specifically those within the Unix realm.

1.6 The Unix Shell

The shell has long history going back to 1977 with the introduction of sh. Early environments involving the shell did not resemble current environments and some did not even have interactive graphical environments, instead requiring users to interact via teletype machines.

This history is still visible in the modern shell, where the term `tty` has come to refer to any text-terminal but originally referenced these teletype devices. While there are very few of these historical interfaces in current use today, the legacy of the shell has been hard to shake loose [Stephenson, 1999]. One cannot dive too deeply into the technology behind the modern shell, without having to understand tools and technologies that may have been phased out decades ago.

It is a testament to the power of the shell that as interfaces to it have changed so dramatically, it's core functionality still resembles what it looked like in 1977 [Jones, 2011].

When we talk about the shell, we are talking about a number of distinct entities. In some cases, we should take care not to conflate these.

The shell can refer to a compositional style of programming where the units of composition are system-level processes. Communication between these compositional units falls under the broad banner of inter-process communication (IPC) – i.e., commonly pipes, shared memory, etc. – and the compositional operators, which will be described in further detail in the next chapter and involve how processes are started (i.e., background/foreground) and how processes are connected with regards to IPC (e.g., connecting of stdout to stdin).

The shell can refer to specifically the `sh`, `bash`, `fish`, `zsh`, etc. programs and the interactive computing environment they provides. The shell can also refer to an attended interactive programming environment provided by these tool or an unattended batch-oriented style of programming they support.

The former case is typified by interaction at the command prompt. The latter is typified by the writing of shell scripts. For users of Linux and even many users of macOS, this interactive style of computing at the command prompt makes up a significant portion of their day-to-day computer use.

Many Linux users even attempt to perform all of their computer tasks from their console. Even those which might fit a different paradigm better – e.g., using `lynx` for web browsing or using `cmus` for listening to music.

This style of computer interaction can be thought of as requiring the user to continuously write very short, throwaway, interactive programs. In this sense, the shell is one of the most successful programming environments. It mates iterative construction of complex programs in an intuitive fashion with more traditional approaches to program construction (e.g. writing scripts). The former style works extraordinarily well.

It is the extension of this style to scripting, where most complaints arise. Because of the unwieldiness of shell syntax and all the little quirks that come from backward compatibility, it is extra-ordinarily hard to write correct shell scripts. This becomes harder with length, which is why some even suggest to not write scripts that are longer than dozens of lines.

This thesis introduces a tool which does not address interactive computing but instead focuses on improves the batch computing use of shell. It tries to extend the viability of shell style programming to business workflow domains that currently often require full-fledged programming environments and platforms.

1.6.1 Resurgence of the CLI in the Era of GUIs

This thesis notes the CLI's continued relevance and uses it to justify a focus on textual interfaces. For many computer users, their lives are dominated by graphical UIs, but it is still worthwhile to talk about CLIs.

Generally, graphical composition is possible. However, one consideration might be the Deutsch limit [Whitley, 1997], which states that it's difficult to put more than 50 visual programming primitives on the screen at once. In the composition of processes by means of visual programming, Pautasso and Alonso [2005] have shown, however, that management of visual complexity is possible by use of nesting and intelligent layout algorithms.

Generally, graphical composition of course has many benefits, as humans are very visually driven and as we are used to manipulate a world of meaning founded in physical and visual objects. Most GUI-applications, however, don't have an accompanying visual programming environment within which to freely recompose their underlying computational mechanisms.

Visual interfaces of programs usually have UI elements that enable only specific, prior envisioned, commands. Only in the rarest of cases are these commands actually composable with each other (e.g., Photoshop – but even tools like Adobe Bridge, which are meant to compose actions within the Creative Suite ecosystem, still use a traditional textual programming interface, namely JavaScript).

As a result, graphical UIs are generally geared towards a specific use-case, predefined by the development team of the software. Unlike in non-graphical interfaces, users don't have direct access to the underlying units of composition, like commands, functions, and variables.

Also graphical programs themselves usually don't lend themselves to be composed with each other. In the command line world, the standard affordance for composition between commands and applications is the affordance of the text-stream, which makes interoperability possible.

No such generally accepted affordance for composition between graphical programs exists, other than importing and exporting file formats. Even that usually depends on whether somebody else decided to support that functionality.

It may be said that visual interfaces are more intuitive to cognitively process. Yet, the more abstract nature of text also has its advantages. For example, representing changes of program structure based on a parameter space becomes difficult to represent visually outside of text, especially when referencing arbitrarily large parameter spaces. There is a paucity of good visual metaphors for these types of higher level abstractions.

According to the *law of leaky abstractions* [Spolsky, 2002], speaking about abstractions through the use of metaphors may aid understanding only up to a certain point. The ease of only being aware of the metaphors that abstractions enable comes at a cost. Every metaphor, by definition of the word, is a kind of modeling that involves a usually lossy representation of the underlying information.

Sticking with the actual textual information has the downside of being more abstract and harder to understand, but the direct and less lossy representation that is given with text can be seen as essential for a programmer who has to have a full grasp of all the details of the deterministic – one could say mechanical – execution within the computation environment.

In a metaphor, there is a certain compression ratio, how much each unit of the metaphor represents of the underlying structure it is modeling. There is a tension between the physical size of a metaphor and the information it can convey. A common saying is that a picture contains more than a thousand words, but doesn't $e = mc^2$ that contain more information and insight on the world than any image could convey?

Similarly, in the study of the visual display of quantitative information, it is often argued by authors such as Tufte et al. [1998] that textual representations such as tables of values can convey information with a higher density than more graphical approaches such as bar-charts. In other words, many visual encodings are much less dense than common textual encodings. Of course this is not universally true. Tufte's sparklines are touted as being a denser way to display numerical information than if you were to write out numbers in a long list, and they are a primarily visual and graphical device.

Even in an era dominated by GUIs, traditional shells like bash and zsh are still under active development and both projects saw stable code releases in 2017. However, given the importance of backwards compatibility, these projects have not been able to introduce dramatic new functionality.

Given the relatively slow pace of developments for mature projects such as these, one might incorrectly assume that shells and shell programming is no longer a popular field of research and that the use of the shell is a holdover from an older era.

In fact, there are numerous attempts currently, to address various shortcomings of the shell and to modernize it in one way or the other. Each of these attempts acknowledge that the shell and shell programming provide a uniquely valuable set of mechanisms and functionality and merely try to address some shortcomings.

Projects like `xonsh`, `ngs`, oil shell, and PowerShell are all parallel attempts to improve shell programming in one way or another. Reversing or improving upon literally decades of history is no small task, but these projects have captured the attention of programmers and made significant headway.

1.6.2 Types of CLI-apps

This thesis goes into depth regarding the features of the shell and programs that can be called from the command line. Applications that run from the terminal have a variety of interfaces that might be called command line interfaces (CLIs).

The following discussion focuses on non-interactive batch programs that operate on data read from the stdin and produce output at the stdout.

`texttop`[1] is an absurd example of a something that could strictly be classified as

[1] https://github.com/tombh/texttop

a CLI-tool because it is invoked from a terminal but is clearly outside the scope of what will be discussed in this thesis.

Our focus in the further discussion are composable stdin-stdout programs, whose only other user inputs are in the form of environmental variables or command line flags. Given an environment and a set of command line flags, these programs run to completions without further user input and once launched can be considered modeless and stateless from the perspective of the user.

(N)Curses programs (like Calcurses, `top`, etc.) and other pseudographical modal programs are outside of the scope of this discussion. Similarly, interactive interpreters, like Python, `qalc`, and interactive use of the Shell are outside of the scope of the units of composition that we refer to.

1.7 `tini.flow` – A Tool for Composing Workflows

This thesis investigates the shell as a tool for composition, it's limitations, potential mechanisms for addressing these limitations, and introduces a new software tool called `tini.flow` that extends shell programming to support complex workflow programming, based on the mechanisms underlying composition in the shell.

These underlying primitives are reused in such a fashion as to enable much more complex compositions than was originally intended with the piping affordance supplied as part of the unaltered shell syntax. In order to make this functionality approachable and discoverable intuitively, the `tini.flow` DSL is introduced as an interface. Furthermore, this thesis investigates topics in software composability, by discussing considerations behind the design of `tini.flow`.

Chapter 2

Rethinking the Shell

Every woodworker needs a good,
solid, reliable workbench,
somewhere to hold work pieces at a
convenient height while he or she
works them. The workbench
becomes the center of the wood
shop, the craftsman returning to it
time and time again as a piece takes
shape Hunt and Thomas [2000].

This thesis focuses on composition where the units of composition are Unix processes. Composing Unix processes has a long history and has been a common and popular programming paradigm for the last few decades.

This approach is not without its faults and if this is the desired unit of composition, its limitations must be understood and addressed.

`tini.flow` is an attempt to build a workflow tool by extending existing tools. In this sense, `tini.flow` itself attempts to compose existing mechanisms to build something more powerful. Therefore, it is critical to precisely identify what is meant by the shell, what the shell offers, and what its limitations are.

But first, what to we mean by *the shell*?

2.1 The Language of the Shell: Operators

When we talk about the shell,

we talk about a language which provides certain compositional operators:

Sequencing operator ; `a; b` means execute a and upon termination of a execute b.

Backgrounding operator – `&` `a & b` means execute a and allow a to run in the background. b will be executed, while a is still running.

Control-flow operator, logical sequencing on conjunction – `&&` `a && b` means execute a and upon termination of a execute b only if a's return code is 0.

Control-flow operator, logical sequencing on disjunction – `||` `a || b` means execute a and upon termination of a execute b only if a's return code is not 0.

Concurrent data flow operator, *the pipe* – `|` `a | b` means concurrently execute a and b and direct output of a to the input of b.

Input and output (file) redirection – `>` or `<` `a <in >out` means execute a, feeding the contents of the file named in to the input of a and feeding the output of a to the file named out.

Input and output process redirection – `>(...)` or `<(...)` (available in most modern shells) `a <(b) >(c)` means concurrently execute a, b, and c, directing the output of b to an anonymous FIFO and feeding the contents of an anonymous FIFO to c. These FIFOs have read-ends and write-ends and are composed of a read-only file descriptor and a write-only file descriptor. Provide the read-only file descriptor of b's FIFO and the write-only file descriptor of c's FIFO to a as positional arguments, typically file paths under `/proc/self/fd`.

There are many other operators not listed here. They will be omitted in this discussion, as they don't add anything within the context of this thesis. Note, in this section we will be talking about bash, zsh, and other modern shells and ignore historical shells, which were significantly more limited by legacy considerations.

2.1.1 Shell Data-channels: Stdin, Stdout, Stderr

When we talk about the shell,

we talk about a certain mechanism that enables the possibility of universal interoperability of processes through text streams. Data-channels, namely stdin, stdout, and stderr are automatically setup at the spawn of each process.

On Unix systems I/O is handled through system calls, which are evaluated in the kernel and are triggered in userland via interrupts. For file I/O, the relevant system calls use file descriptors to reference open files as their arguments for these system calls. These file descriptors are integer values, defined on a per process basis and visible only to the process.

By convention, the stdin, stdout, and stderr are assigned file descriptors 0, 1, and 2. These file descriptors can refer to any Unix file: an actual file on disk, a /dev/-device, a /proc/-device, or a number of other structures, represented by the kernel as a file. By default, an interactive console will connect the stdin to its own input source and the stdout and stderr to an output source that is drawn on the screen.

2.2 Shell: Units of Computation

When we talk about the shell,

we talk about the composition of certain units of computation. When we talk about units of computation, we can talk about functions within the context of a programming language, about processes within the context of an operating system, and even about containers or services within the context of a service-oriented architecture.

In each case, we are describing a system that composes discrete – i.e., indivisible and opaque – units of computation. These units are indivisible, in the sense that they are logical units and are used as black boxes.

In the context of the shell, the individual process is the smallest level of composition and processes are typically in the form of individual binaries that are executed within an isolated environment (e.g., virtual memory, etc).

2.2.1 Basic Vocabulary of Simple Shell Commands

When we talk about the shell,

we will be frequently referring to specific computational units, i.e., processes, in order to motivate examples. Here is our basic vocabulary of commands. For detailed information about specific flags and tools which are not mentioned here, refer to explainshell.com – a wonderful interactive resource for looking up all the details of shell commands, their flags, as well as full chains of commands, by pasting in whole shell expressions.

`seq x y` output the sequence of numbers on the interval [x, y]

`sort` sort lines, flags control the comparator

`uniq` filter input line-by-line, suppressing the output of repeated lines

`ts` prepend a time stamp to every input line

`nl` prepend ascending line numbers to every line

`grep` output only lines matching a given pattern

`xargs -n1 echo x` for each line in stdin(for -n1; n2 would take two lines, space-separate them and then) prepend x for every line of output

`wc -w` print word (-w) or (-l) line counts

`cut` remove sections from each line of files

`find . -type f -print0` find all files in the current folder and print their names NULL-separated instead of newline-separated

2.3 The Advantages of the Shell

As Hunt and Thomas [2000] states, every programmer needs a good workbench and the shell supplies just the kinds of affordances that a programmer needs. While GUI-lovers disagree that the shell alone is the way to go, it is the completely free composability of commands that allows the shell to have a strict superset of functionality any GUI-based application could provide for similar use-cases.

Most of all, the shell offers the programmer, through the scripting of shell commands, automation of everyday tasks almost for free.

The shell is a very popular programming environment that excels at building complex macro tools for commonly performed tasks. It is known for its ease of composition, ubiquity of its mechanism, and a clear set of principles for composition and design.

Examples of the power of the shell are abundant: Just google for *bash oneliners* and take your pick! Here is a personal favorite that illustrates well the convenience and the kind of very complex tasks which can be built from simple composition of common tools:

`htop | nc seashells.io 1337` — this is a service oriented architecture, implemented with really fundamental built-ins: takes stdout, uses nc to send it over the network and you get pretty printed output back

The power of the shell is illustrated by an interaction between Donald Knuth and Douglas McIlroy. Tasked with writing a program for computing word counts, Knuth applied his literate programming paradigm and wrote a detailed multi-page exquisitely documented solution.

The solution showed the advantages of literate programming, by interleaving this documentation with the code that implemented its functionality. McIlroy responded to this with a six-line shell script that did the exact same thing. For documentation, he added a comment on each line that precisely detailed what each component did Drang [2011].

The terseness and high readability of McIlroy's version shows the power of the shell. In fact, that it is so terse and reuses so much pre-existing functionality, makes its relatively meager documentation clearer to the reader than Knuth's pages upon pages of explanation.

2.3.1 Data-flow in the Shell

With all the criticism of shell for its awkward syntax, it could be argued that the piping mechanism is peculiarly elegant in its simplicity and its efficiency of allowing program composition.

There is much to be said about structured versus unstructured passing of program information between parts of a modular program design.

One thing is clear however: that *a bag of bytes* must be seen as the lowest common denominator of inter-process communication. A bag of bytes, other than having a beginning and an end, is a fundamentally unstructured data format. The power of not imposing any structure assumptions can be seen in a simple but classical combination of command line apps for pushing and pulling files over the network, using compression and end-to-end encryption tools.

```
$ cat file | gzip -c | ssh user@host 'pv | gunzip > file'
$ ssh user@host 'cat file | gzip -c' | pv | gunzip > file
```

Listing 2.1. Pushing or pulling a bag of bytes over the network

The utility `cat` appends the file's contents to stdout, which is then piped to gzip, which compresses it, and with the `-c` flag is instructed to pass its output over stdout instead of writing to a file. This is then piped to ssh for transfer to a remote machine. On the remote side, it executes the command pv to show the progress of the transfer and `gunzip > file`, to unzip the contents of the stream to a file. This works as all the tools that are part of this pipeline are able to process streams of bytes in a way that is oblivious to the content of such a stream: A bag of bytes.

The convention of passing data inside the shell is usually the following however: the bytes are being text-encoded in something like UTF-8 or ASCII and individual data records are by convention separated through newlines. The advantage of such a format is that anyone can simply create such a program. As an example for the ease of writing programs to this convention, we can see two ad hoc reimplementations of the classical Unix utilities `ts` and `nl` being depicted below:

```python
from datetime import datetime

for line in stdin:
  print(f'{datetime.now()} {line}')
```

Listing 2.2. Ease of writing to the stdin/stdout API – `ts`

```python
for lineno, line in enumerate(stdin):
  print(f'{lineno} {line}')
```

Listing 2.3. Ease of writing to the stdin/stdout API – `nl`

2.3.2 Ubiquity of this Mechanism

The mechanism of the pipeline concept was devised by Douglas McIlroy, who had noticed that command shell workflows were often characterized by passing output files to the next application for processing. This repetitive workflow begged for automation within a more elegant mechanism. According to the *Annotated Excerpts from the Programmer's Manual, 1971-1986* [McIlroy, 1987], McIlroy had to persuade Ken Thompson to implement the concept within Unix.

> In one feverish night Ken wrote and installed the pipe system call, added pipes to the shell, and modified several utilities, such as pr and ov (see 5.1 below), to be usable as filters. The next day saw an unforgettable

> orgy of one-liners as everybody joined in the excitement of plumbing. Pipes ultimately affected our outlook on program design far more profoundly than had the original idea of redirectable standard input and output.

This feverish night is said to have taken place in 1973 and with the convenience and power of the pipe has forever changed and shaped the way we compose programs within the operating systems of the Unix tradition.

The ease of writing a program to read from stdin and write to stdout and the ubiquity — one could say domination — of Unix-based operating systems has led to an incredible wealth of openly and often freely available command line utilities for all kinds of purposes, even such unlikely ones as sending stdout to a Google spreadsheet[1].

The piping mechanism can be critiqued for passing program output in an unwieldy and unstructured way, meaning that it has to undergo various types of ad-hoc processing across various steps of the pipeline.

However, this also means there are no constraints on the programs themselves. It means that any applications can be incorporated without any other restriction than reading from stdin and writing to stdout.

The concept of the pipeline as a way to enabling composability and the processing of streams eventually became abstracted into the *pipes and filters* software architectural design pattern. This design pattern is formulated as describing the flow of data streams through a pipeline with filters, each of which processes the input stream in some way in order to produce an output stream.

Composability and reuse of functionality across a software system are fundamental pillars of a good software architecture. There are many ways to achieve this end, many of which might be superior to piping text streams. The simplicity of the mechanism, however, and its lack of restrictions explain the persistence of this concept over time.

2.3.3 The Unix Philosophy

Choosing Unix processes as a unit of composition for workflows, allows workflow authors to adopt the Unix philosophy. The Unix philosophy makes the development of large systems easier. When creating a new process for inclusion in a workflow, there is significant flexibility in the level of granularity embodied by the process.

A new process can have a large scope of responsibility or it can be decomposed into smaller processes with smaller scopes of responsibility. If the authors of a system need to compose just the functionalities of the smaller scope, the choice may be clear, but lower levels of granularity tend to introduce overhead in the development process. So, choosing the lowest scope of responsibility is inadvisable in the absence of known requirements.

[1] https://github.com/kren1/tosheets

Since the authors of these tools cannot perfectly predict how they will be used in larger systems, The Unix philosophy gives them a nice rule of thumb to guide their work. While this philosophy is not followed in all cases, it is sufficiently widely referenced that the shell has a remarkably consistent set of compositional interfaces, given how disparate of the authors of these units have been in space and time.

The Unix philosophy circulates around the idea of having many small programs with clearly defined tasks each and enabling software reuse through compositions of these small applications into larger use cases [Salus, 1994]. One of the main strengths of Unix environment is that it is designed with the idea in mind of enabling programs to easily cooperate with each other.

In *The Art of Unix Programming* [Raymond, 2003] the philosophy is summarized as follows:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

It is generally agreed that modularity is a good thing and having a universal interface sounds great, right? This leads to a situation, however, where the reusable components can't be reused without always having to tack on ad-hoc text-formatting adaptors on both sides of a compositional boundary. This issue will be elaborated in section 2.4.1 about line and string orientation.

2.4 Limitations of the Shell

Many problems arise from the fact that the shell has two masters: humans and computers.

Without the human master, the shell might never have become popular, but this directly influences and adds difficulty in order for it to serve the machine master. It introduces a tension between easy machine-level composition and the human beings ability to actually operate with this tool.

Command line apps often introduce inconsistencies in their interface in the name of brevity or terseness, where this benefits the human being, by allowing them to type something shorter or more naturally.

This however makes scripting more difficult and yet if the shell abandoned its human conveniences, it might end up a tool that nobody wants to use.

2.4.1 Line & String Orientation

In the shell, the only limitation of interprocess communication is that the medium needs to be a stream of bytes. Convention has it however that usually this string of bytes – at least if the processes need to understand the stream of bytes that they are processing (as opposed to let's say an encryption tool, which usually acts indifferent to the meaning of the bytes it encrypts) – are encoded as character strings.

One character string usually encodes one record that is delineated to the next one by a newline `\n`-character.

However, outside of simple streams of bytes, text is not always line-oriented. Some applications, especially human-facing ones, use phony, ad hoc object encodings.

Pseudo-object formats & Tabular Output

Take the following output of the `ifconfig` command:

```
$ ifconfig

lo  Link encap:Local Loopback
    inet addr:127.0.0.1  Mask:255.0.0.0
    UP LOOPBACK RUNNING  MTU:16436  Metric:1
    RX packets:8 errors:0 dropped:0 overruns:0 frame:0
    TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:480 (480.0 b)  TX bytes:480 (480.0 b)

p2p1  Link encap:Ethernet  HWaddr
    inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
    UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
    RX packets:41620 errors:0 dropped:0 overruns:0 frame:0
    TX packets:40231 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:21601203 (20.6 MiB)  TX bytes:6145876 (5.8 MiB)
    Interrupt:21 Base address:0xe000
```

The output is not one record per line. In fact, no single line can be interpreted without the context of surrounding lines.

This is a result of the shell having to be human-usable. The above output is human-readable, but this makes it cumbersome to process automatically, because its output is a pseudo-object in a text encoding. (Consider in the above that each network interface is an object and the network statistics on each interface are attributes of those objects.)

This textual encoding introduces a huge compositional barrier in contrast with an object-encoding like PowerShell, which intends to eliminate this component port mismatch. Encodings which textually represent an object in a human-facing manner need to be reparsed at every compositional boundary to create and recreate the actual object used in each stage of processing.

Even very simple tasks suffer from this component port mismatch.

Take `ls`, a commonly used, human-facing tool. It is enormously useful in interactive use, but immediately presents problems in compositional use.

Even when run without flags, the output of `ls` is hard to parse with standard ad hoc string manipulation approaches, because its output has varying amounts of spaces used for orienting columns.

`ls` tries to display its output in a space-saving manner that is easily human-perceptible. For machines however, this requires tricky parsing, as there isn't a specific record-separation character like the newline or NULL to delimit fields.

```
$ ls
anaconda3  Documents  Dropbox  Pictures  Templates
Desktop    Downloads  Music    Public
```

`ls -lh` can provide this same information a tabular format, but that hardly improves things. Parsing this output is still far from simple. Additionally, flags like `-h` are useful for human beings, giving human-readable file size information (displayed as K, M, G, ...), but useless for a machine parsing.

```
$ ls -lh

total 2,1M
-rw-r--r-- 1 cornelius cornelius 1,8M Nov 27 03:34 1925551162_n.gif
-rw-r--r-- 1 cornelius cornelius 192K Nov 27 19:35 mcillroy.png
-rw-r--r-- 1 cornelius cornelius  12K Dez  1 22:43 PyData NYC17.ipynb
-rw-r--r-- 1 cornelius cornelius  98K Nov 28 05:58 tiniflow.jpeg
-rw-r--r-- 1 cornelius cornelius  609 Nov 24 18:18 Untitled.ipynb
```

Trying to parse this tabular output is a nightmare and the trickiness of employing ad hoc, string-manipulation based parsing to handle it detracts from the beauty and simplicity of the underlying piping mechanism of the shell.

\n and Other Evils

However, the problems don't stop with pseudo-object formats and tabular layouts. As detailed by Wheeler [2016], filenames on Unix can include characters that thwart simple, ad hoc parsing, requiring peculiar solutions to correctly solve.

To show the constant problems with record separation in detail, especially regarding the newline character `\n`, let's go over the following example of wanting to delete the files in the current directory.

We will go over the different possible options step-by-step, looking at a code snippet at a time.\n

```
$ rm *
```

This doesn't work: a filename of the type '`-i`' would introduce wrong behavior.

In Unix filenames are allowed to include and even begin with dashes! Therefore, '`-i`' would be a legal filename — but it will confuse `rm`, because it will not be able to disambiguate between the flags passed and the names of files meant to be deleted.

This kind of error can sneak up on users, as it is a rare case to have a file with this name. Some shell script for automation may appear to work until the day it encounters a filename like this. Then, suddenly, everything will break.

```
$ ls -1 | xargs rm
```

This approach tries to be clever by listing the filenames with `ls -1` and then prepending rm to each line. But this also doesn't work dependably: it will choke on a `\n` newline character, which is a legal filename in Linux.

As mentioned in the previous example, things might work in most cases, but that one unlucky case of not deleting a file – because the shell tries to `$ rm abc` and `$ rm def`, instead of `$ rm 'abc\ndef'` as intended – might be critical.

```
$ ls -1 | while read file; do rm $file; done
```

This variation suffers from the same fate, it will choke on `\n` newline characters in filenames. The while loop iterates over the lines (\n-separated) of the stdin, thereby ending with the same fate as the previous approach.

Unfortunately, the shell contains many ways to do the same task in equally wrong ways!

```
$ rm -- *
```

This actually works, but it needs shell support. Most shells support *globbing* – the process of expanding filenames specified with wildcards – but some do not!

Also, globbing may trigger re-expansion in other cases, meaning that this is not a general fix to the delimiter problem.

```
$ find -exec rm -f -- {} \;
```

This works. The `find` command, combined with the `-exec command` flag, executes a command on every found filename, fitting the specified pattern and according to the set flags.

But this means something subtly different. In this case, no pattern or flag regarding file type has been specified. This means that find will search the current directory and all its subdirectories.

In the previous attempts, we used non-recursive `rm`: meaning that we only intend to delete files in the current directory and not the subdirectories as well as their contents. To get the same functionality with find, one would need to add another parameter: `-maxdepth 1`.

The shell offers many overlapping and subtly different ways to do the same task (often in an incorrect fashion)!

For `-exec`, there needs to be some escaping mechanism to delineate its arguments – the commands, options, et cetera – in such a way that the shell doesn't misunderstands and processes them in some other way or even expands parts that are not supposed to be expanded. This is done by ending the construction with an escaped semicolon `\;` to the end. The beginning of the construction is the `-exec` itself. The placeholder for the to-be-filled-in filenames is `{}`, which might also need to be escaped by having single quotes around it, in order not to be escaped based on the rules of the shell in use.

From the previous paragraphs, it becomes clear how unwieldy this approach is, one could even say that it is ugly and unintuitive. Also, almost more importantly, it doesn't allow for adding extra intermediate filtering steps, as instead of using the piping mechanism, it just directly executes the command `rm -f` directly on the results of the `find` operation.

The above example works, but it abandons all composability!

```
$ find -print0 | xargs -0 rm -f --
```

The above works in all cases and is generally the solution that is advocated. It works because the `-print0` flag instructs `find` to give its output NULL-separated instead of newline-separated. On the other side of the pipe, `xargs` is instructed to take NULL-separated input with the `-0` flag. The `-f` flag stands for 'force' and makes sure that non-existent files are ignored, never prompting the user for input. The latter is especially important for use within scripts, which are generally written to run without human oversight and which would just halt in the case of a prompt.

This advocated solution works perfectly and in any possible scenario, but who can remember it?

It also relies on every tool in the pipeline having support for the null-delimiter, and it relies on the data set not allowing null-values in data entries. (Not every tool supports this and not every data set has such restrictions.)

It is a clumsy workaround and doesn't adhere to the simplicity and elegance one might expect when first hearing about the benefits of the pipeline mechanism of the shell.

Let's consider another example. Try to parse the tabular format of `ls -l` as part of a pipeline to count the number of files owner by a user.

```
$ ls -l | tail -n +2
        | sed -r 's/[ ]+/ /g'
        | cut -d ' ' -f 3
        | uniq -c
```

Listing 2.4. Ad hoc parsing needed for parsing tabular `ls -l` output in order to determine number of files per owner

It is undoubtedly amazing that we can build such a pipeline incrementally, going step-by-step through the intermediate presentations, until we reach a pipeline presenting us with the wanted output.

This even works in practice when building these pipelines interactively. Presumably, any of the previous issues would be caught in one iteration and addressed in the next.

But this solution fails in a batch-processing context, because it is not safe to generalize ad hoc, interactive work at the shell.

We could again return to find, and, digging into the man page, discover the following solution:

```
$ find -printf %g\n | sort | uniq -c
```

Considering these deficiencies, one might say, *'hey, some tools were designed with humans in mind, and some were designed with machines in mind. Use the right tool for the job!'*

But this is actually besides the point. First, because some tools might only exist as human-facing versions and, second, it suggests that the shell should have two versions of every tool. Why can't the human-facing and machine-facing purposes be integrated?

```
$ list-files | filter --user | uniq -c
```

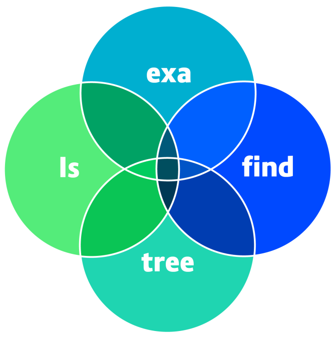


Figure 2.1. these tools are *not* orthogonal to each other

This limitation means that a simple set of orthogonal tools for humans as well as machines is **impossible with** all the issues of field and line separation of **text-based streams**.

> Orthogonality is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are *independent*. Move along one of the lines, and your position projected onto the other doesn't change.
>
> In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface. [...]
>
> An orthogonal approach also promotes reuse. If components have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned by their original implementers. The more loosely coupled your systems, the easier they are to reconfigure and reengineer. [Hunt and Thomas, 2000]

These deficiencies result in duplicated effort, because they often necessitate multiple tools with overlapping functionality. This is in direct violation of the Unix principle of having each tool do one thing well and having complex functionality emerge from composition.

Problem or Solution?

The difficulties introduced by the line and string orientation of the shell are undeniable. In some cases, such as the processing of filenames, using `find`, `sort`, and other tools, we can rely on specific known details of these data formats to work around problems like `\n`-delimiting cause.

`find`, `sort`, and other tools have flags to switch their record separator to the `\0`, knowing that Unix filenames are not allowed to contain null bytes. This is a special case, as other data sets may not have such restrictions and there is a fundamental risk of component port mismatch between these tools.

Shell programming environments such as PowerShell try to address this problem by transmitting objects rather than string-data over their data channels. This approach works when every process is written upon a common runtime and can share a common concept of what constitutes an object and a common implementation.

But this approach is universal only within the limited universe of the runtime that defines this object model. The Unix shell has no common runtime. It has only binary data strings as its most fundamental data type. It is possible to consider subdomains of the universe of shell processes and to create the same sort of common data encoding standard that PowerShell has.

For example, there are many tools that can read and write binary-encoded JSON or XML data and within pipelines using these tools, the aforementioned problems are not present. In some cases, standard tools like grep and sed have been rewritten as variants that work within these domains, e.g. jq[2].

The line and string orientation is undoubtedly a source of errors and frustration when dealing with the shell. However, the string orientation of the shell allows it to accommodate numerous subdomains which can choose their own data encoding and avoid the aforementioned problems.

The line orientation of the shell makes simple interactive shell scripting easy and straight-forward in the many common use-cases. The problems of line-orientation can be addressed in a similar fashion with custom domains, which can employ object-encodings, columnar data-encodings, n-matrix encodings, relational encoding, or whatever type system and encoding is best suited to the problem.

2.4.2 Only One Data Channel (stdin & stdout)

The operator for data composition in the shell, the pipe, implicitly works on the stdin and stdout. This is the only data channel that the shell can be guaranteed

[2] `https://stedolan.github.io/jq/`

will exist for all programs. (Of course programs can choose to ignore their stdin or stdout or even close or reassign them.)

The shell and common programs written for the shell typically assume there is only one guaranteed data channel. This assumptions causes problems for programs that need to read from multiple distinct data sources or write multiple distinct data outputs. This should be the case in a program that wants to emit both control information for control flow decisions and data that the program is actually operating on.

It is always possible to take one input and output channel and multiplex multiple distinct types of information on this channel. However, this leads to its own problems. It is also possible to have programs read or write to FIFOs that are not specified as part of the compositional grammar, but are specified as command line arguments or are part of some implicit state.

Having one data channel has the benefit of being simple, which is where we should start – following the *KISS design principle*, originally stated by, lead engineer of Lockheed Skunk Works at the time, Clarence Johnson Rich [1995]. According to this principle, we should "**k**eep **i**t **s**imple [and] **s**tupid" – an acronym reportedly stemming from the U.S. Navy in the 60s. More complexity means more surface area for failure to creep in. Therefore, complexity in engineering needs strong justification.

The shell was originally designed for pure data flow composition, where many control flow decisions would be made interactively by the user. An additional scripting layer is included in that processing context, but the scripting layer does not integrate into the data flow compositional layer.

Before investigating extending the shell, consider the current state of managing multiple data channels.

In a text-based stream system that uses some kind of separator – in the shell usually the newline-character \n – often there is no good way to output different types of data at the same out. The only way would be to tag the data, but then the applications handling the data on the other side of the pipe would have to have this logic implemented as well. This goes against the wish for universality in the composition of programs.

```
$ parse-graph |edges| edge-printer \
              |nodes| node-printer
```

The above hypothetical syntax would send two independent and distinct output streams to two independent and distinct programming pipelines. However, this syntax does not exist, even though it is a logical extension of syntax that does exist – the syntax for redirecting stdout and stderr to distinct files:

```
$ program 2>error.log >output.dat
```

An object-based system, like PowerShell, where objects are the data format passed between processes and where one object would represent one record, this record could just have different fields encoding different kinds of information at the same time. To give an example of the kind of problem that would arise in a text-based system with only one data-channel would be the following: How do we encode both edges and nodes as output to stdout of a single application? With the use of stdout, the only way would be some kind of ugly interleaving. Let's look into the options in detail below:

Option 1 – interleave on stdout: ⊖ requires parser ⊕ less redundant work

```
$ parse-graph --print-nodes --print-edges < graph.dat
node a
node b
node c
edge a - b
edge a - c
```

Option 2 – use stderr as a data channel ⊖ where do you put actual error messages?

```
$ parse-graph --print-nodes=stdout --print-edges=stderr < graph.dat
node a
node b
node c
edge a - b
edge a - c
```

Option 3 – two processes ⊖ lots of redundant work ⊕ "fits" line-orientation better

```
$ parse-graph --print-nodes < graph.dat
a
b
c
$ parse-graph --print-edges < graph.dat
a - b
a - c
```

Option 4 – write to a file ⊖ no longer stream processing

```
$ parse-graph --print-nodes=node-file --print-edges=edge-file < graph
    ↪ .dat
```

Option 5 – write to a FIFO ⊖ OK, let's move on

```
$ parse-graph --print-nodes=node-fifo --print-edges=edge-fifo < graph
    ↪ .dat
```

Each of these options has limitations, but none of these limitations should be intrinsic to the shell.

The stdin and stdout presented to a process are file descriptors like any other. A process could be provided numerous additional file descriptors, to allow to write to or read from multiple locations seamlessly.

This functionality is most useful in the following cases:

1. custom processes, especially those for handling higher dimensional datasets, where output for each dimension may be independent
2. tools that sit at the top or bottom of pipelines, to split or join data streams
3. workflows that need to convey both control information and data between nodes

Typically, in the second case, we need custom tools to extend common operations, for example wtail is a version of tail that can operate on multiple simultaneous input files. In this case and the first, before spawning these processes we could create additional file descriptors and pass the values of these file descriptors in environmental variables. The processes would look at these environmental variables to know that they have access to data streams beyond the 0, 1, and 2 – the stdin, stdout, and stderr.

Simple wrapper scripts could be written to adapt any of the above multiplexing syntaxes to this approach in order to allow us to avoid having to change some existing processes, without forcing us to use intermediaries, such as files on disk, for capturing new data sources or sinks.

In the last case, we see the most interesting variant of this problem. Fundamentally, in a workflow graph, the nodes are processes with connectivity represented by edges. The connectivity between nodes of this graph could be connectivity of data, i.e., stdin and stdout, or could be a logical connectivity, i.e., triggering conditions and sequencing.

In the latter case, if we control the spawning of these processes, we could model these as separate sets of edges, where the first set of edges create FIFOs that are bound to the stdin and stdout and the second set of edges create FIFOs whose existence is signaled through specially named environmental variables.

The controlling mechanisms of a workflow are likely to be a combinations of set shell scripts and ad-hoc shell snippets. It should be straight-forward, to have these snippets operate on only one set of file descriptors and have the processes they launch operate on the other.

2.4.3 Linearity of Pipes

The core syntax of the shell has no branching for data flows. Data flows are always linear. Because pipes are read once, branching requires some intermediary that can allow out-edges to read more that once.

Pipes can only support linear data flow connections. Linear means one process feeds into one process, feeds into one process, feeds into another process without branching. Obviously, shell syntax for control flow can lead to non-linear call-graphs, but this mechanism is separate from data flow and is not a first-class mechanism, i.e., you cannot immediate visualize, inspect or report on the call graph of a shell script. You can also not do this in a C program or Python script, but in graph execution frameworks, this is considered to be core functionality.

Via the pipe()-function, we can create anonymous pipes. These pipes can be read and written to and are otherwise identical to the stdin and stdout. The stdin and stdout are privileged, in that they are given fixed file descriptors 0 and 1 and in that most programming environments operate on them by default – e.g., the print function prints to stdout by default. And most programming environments provide conveniences to work with these, such as `sys.stdin` in Python.

However, our own anonymous pipes are indistinguishable from the stdin and stdout. If we tell a process via an environmental variable, which file descriptors to operate on in order to interact with our pipes and add a couple of similar conveniences, we can add an arbitrary number of completely separate channels to a data flow.

These extra-channels can convey controlling information, such as indication when a process has terminated and indicating its return code.

Via wrapper processes, multiple input channels can be interleaved into a single channel and fed into stdin or they can be fed into auxiliary programs, which can perform control operations such as launching of processes.

Most traditional commands may not themselves support multiple I/O or a control channel and it is not feasible to rewrite all of coreutils. However, through clever use of wrapper programs, it may be possible to create logical units that wire up multiple channels or operate on a control channel to enhance these existing programs.

pee & tee

Coreutils[3] and moreutils[4] are two packages containing common core shell utilities. Coreutils contains a utility called tee, which is commonly used for taking a single data input stream and copying it to multiple output streams.

Tee could be combined with process redirection, to be used in the construction of a non-linear data flow as follows:

```
$ seq 1 10 | tee >(nl) >(ts)
```

Pee could be used in the same fashion, without the need for explicit process redirection syntax:

```
$ seq 1 10 | pee 'nl' 'ts'
```

The above examples show a single branch and these branches could be nested to form larger trees.

```
$ seq 1 10 | tee >(grep 3 | tee >(nl) >(ts)) >(grep 4 | tee >(nl) >(
    ↪ ts))
```

One additional layer of nesting is required for each level of the tree. Clearly, this will quickly grow out of hand.

[3] https://www.gnu.org/software/coreutils/coreutils.html
[4] https://joeyh.name/code/moreutils/

dgsh

There is a need for better syntax for specifying large non-linear data flows. In extending *Unix pipelines to DAGs*, Spinellis and Fragkoulis [2017] introduce a tool called dgsh, for creating non-linear pipelines.

The above example could be rewritten in dgsh as follows:

```
seq 1 10 |
tee |
{{
    grep 3 |
    tee |
    {{
      nl

      ts
    }}

    grep 4 |
    tee |
    {{
      nl

      ts
    }}
}}
```

This syntax is a clear improvement on the use of tee and pee for larger graphs. However, it does not introduce additional functionality, missing from tee and pee. The functionality in tee, pee, and dgsh can be recreated in simple fashion with the use of anonymous FIFOs and simple code blocks that read from one source and copy that data to multiple sinks.

Actually, the functionality of dgsh can be reproduced within structure of this simple shell script:

```
#!/bin/bash
# dag.sh

dag="$0"
node=$1; [[ -z "$node" ]] && node=root
case $node in
  root)    seq 1 10 | pee "$dag left" "$dag right" | sort ;;
  left)    grep 3   | $dag ts                            ;;
  right)   grep 4   | $dag nl                            ;;
  ts)      ts                                            ;;
  nl)      nl                                            ;;
  *)       echo 'Bad node' >&2; exit                     ;;
esac
```

The fundamental problem is finding a good syntax for describing these non-linear structures and to automatically create the plumbing to implement them.

2.4.4 Bizarre and Premodern Scripting Syntax

Shell syntax is beholden to backwards compatibility. New features like arithmetic or data structure support must not break existing code, which could be decades old.

As a consequence, shell syntax feels cobbled together and seems to lack a coherent design. Full-fledged programming languages have the advantage of being designed from the ground up. Many modern shell replacements, e.g. Fish, are abandoning compatibility with old shells, so that they do not get trapped with supporting this mess of syntax.

Shell attempts to include the functionality necessary to do many tasks that otherwise require a full-fledged programming language. When writing shell scripts, there is always the question of when to abandon the shell and to rewrite it in another language, like Python, Perl, Ruby, Lua, or even C or C++.

The developers of popular shells have pushed their programming environments as far as possible to allow users to defer rewriting their shell scripts as long as possible. When used for composing other programs, shell syntax is extremely lightweight but its syntax outside of piping is nowhere near as convenient or coherent as the equivalents in modern scripting languages.

To call the shell syntax bizarre, is something of a subjective value judgement. However, it's irrefutable that much of the shell syntax does not match other imperative languages, that common operations on common types, like find-and-replace on strings, requires either shell-environment's specific syntax, like bash's `${var/` ↪ `Patter/Replacement}` or in the case of uppercasing a string, completely lacks any convenient syntax and instead requires contortions such as `$(echo $var | tr` ↪ `a-z A-Z)`. Note, some systems support `typeset -u`.

Shell syntax is not coherently designed, leading to many special cases and inconsistencies that have clearly been added after the fact, such as `typeset -A`. Very common tasks, like performing arithmetic operations, require specific syntax like `$(())`. In a modern language, numeric types are on equal footing with string types, so special syntax for denoting arithmetic expressions is unnecessary.

The shell, which historically handled only string types, requires this special syntax to remain backwards-compatible with shell scripts written 20 years ago.

Furthermore, on account for the need for backwards-compatibility, the shell features syntax with overlapping functionality with subtle and obscure semantic differences – for example static vs. dynamic parsing differences between `[ ... ]`-style and `[[` ↪ `... ]]`-style comparisons.

Finally, quoting and escaping will lead you into an endless jungle of madness. It requires enormous discipline and perseverance to make work and readily punishes the user for very slight missteps.

This is shell syntax:

```
xs=( 1 2 3 4 )
xs=($xs $((${xs[4]} +1 )))  # append 5
xs=($xs $((${xs[5]} +1 )))  # append 6
ys=( $xs $xs )              # concatenates; no sublists!
for y in ${ys[@]}; do
    echo $y
done
```

2.5 Why Not Abandon the Shell?

Seeing all the criticism of the shell, why not just write a custom script implementing your workflow, for example in Python? We would have to find all the libraries for all the same things the shell utilities needed for this workflow are already doing – at no extra cost other than piping them together and maybe some ad hoc text processing. Integrating these libraries, accounting for their individual APIs would be next. And what if there is no such (esoteric) library? Would we then write it from scratch? What about correctness and performance of this custom functionality we came up with in an ad hoc fashion? Do we really want to reinvent the wheel?

Within the realm of the shell we have a wealth of tools that already exist and that are more often than not seasoned to such a degree that bugs will either have been fixed already or are well-documented. For popular tools, performance will have been optimized where there is need for optimization.

For old tools, the optimization generally stems from the years of scrutiny, but also computational efficiency that used to be much more relevant decades ago. This will still pay off even today with data-intensive workloads, as shown in the PyData talk *Replace Hadoop with Your Laptop* by Boykis [2017]. Even though the talk only refers to workloads that are of a size that fits a single machine, it shows the that raw speed is not necessarily a problem with shell piping, when it outperforms Hadoop by two orders of magnitude.

The shell, with its low demands on the interface between programs merely being a stream of bytes, is the ultimate glue between programs. This makes it possible to create program compositions between arbitrary programs, written in different times, languages, by programmers that knew nothing of each other – as long as the programs read from stdin and write to stdout.

Returning to the previous point of why to use processes as the units of composition, the limitations of the shell again suggest that composing Python functions might be more robust and less problematic.

Indeed, this is not merely a theoretical suggestion but in fact many real-world software systems abandon the Unix approach as they grow and evolve to become their own platforms.

However, we aim to allow composition of as many existing entities as possible and so, if `tini.flow` were its own enclosed platform, we would not be able to use the rich libraries of command line tools that have already been developed – often in other languages and other eras.

It is hard to justify abandoning the shell and its enormous library of existing programs, because, in spite of the limitations of its interfaces, these interfaces are sufficiently universal that users could create units that could be used within `tini.flow`, without those users even knowing that `tini.flow` exists.

The construction of the workflow graph that `tini.flow` relies upon definitely needs more functionality and better ease-of-use than what Shell syntax can provide.

Rather than writing a workflow system, a similar question would be: Why not write an ad-hoc Python script that uses subprocess.run() to directly invoke units? This script would need additional structure to be able to perform additional meta-operations, namely operations trying to manipulate or interpret components of the Python script as conceptual parts of a workflow.

`tini.flow` is a conservative attempt to add this structuring, in an elegant, flexible, and robust way.

Chapter 3

`tini.flow` in Depth

`tini.flow` is a workflow system consisting of the following pieces:

- an eDSL (Python module) for programmatically specifying the graph compositions
- an execution engine for executing workflow programs
- a library of control wrappers for modelling complex triggers
- a shim for connecting the DSL to the construction of a graph

Together, these solve the following problems:

- specify a workflow as a decomposed set of nodes and edges
- specify edge information to indicate data flow and control flow triggers
- build a graph from the above specification
- execute the graph as a set of system processes
- wrap certain processes with helpers to process control flow information

Its design focuses on ease of implementation, user affordances, extensibility, simplicity, code-reuse, and quality of the compositional framework it provides.

As a practical matter, it was developed in the limited resources available during latter phases of developing this thesis. In this sense, it represents a software project subject to the same limitations as many commercial projects.

It is provided as a proof of concept, with the minimum viable set of functionality necessary to demonstrate a set of ideas developed in this thesis about workflow systems. It has been written with care to avoid design errors. Acknowledging that design errors are an inevitability, certain aspect of the design process provide *escape hatches* for later revisiting design decisions and correcting them without necessitating a total rewrite.

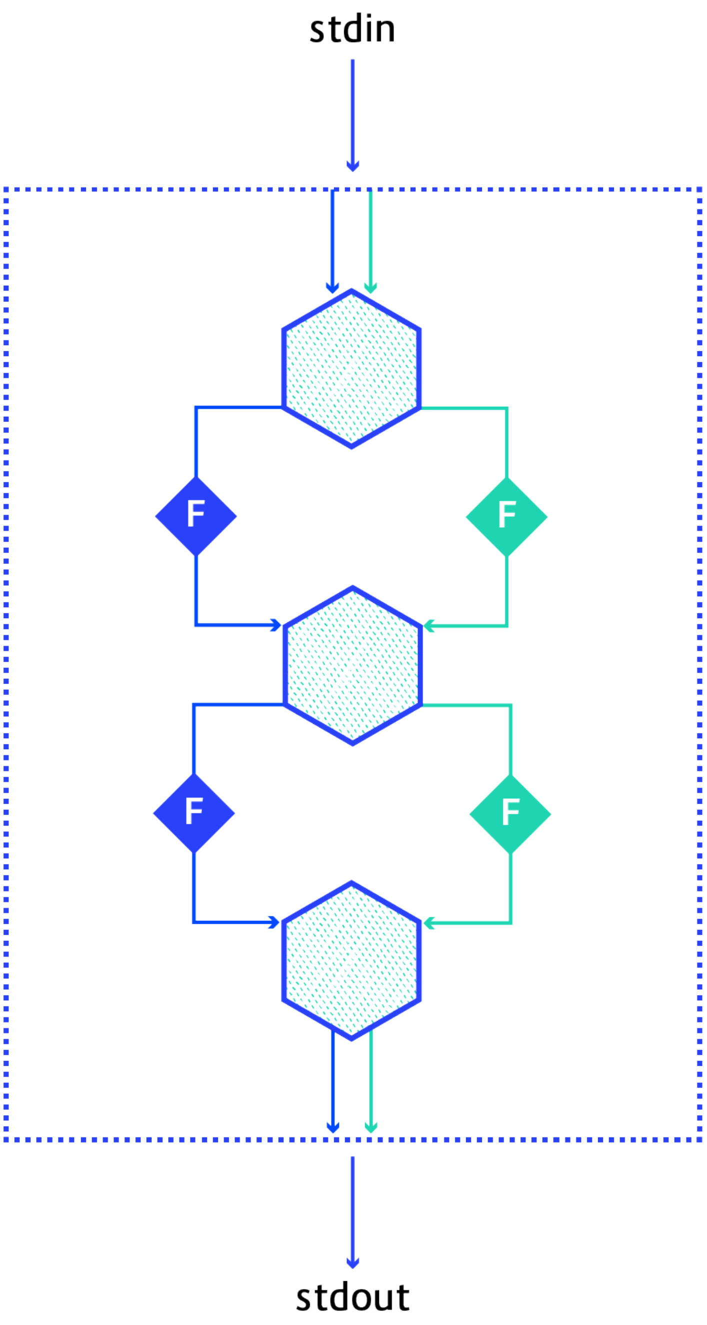


Figure 3.1. a `tini.flow` control and data flow graph

3.1 Software Architectural Design Goals

tini.flow was developed under the following set of architectural design guidelines.

It derives inspiration from many existing tools, such as communication files[1], the shell, or the use of Make for non-build workflows. It takes a practical perspective and is intended to be a tool that will see actual production use (unlike communication files, but like Make and shell).

tini.flow revolves around having an accessible user-interface that is visually pleasing with affordances which are either easily visually graspable or with which the user is likely already acquainted by common use, namely shell piping and Python syntax.

Visual appeal is not only about aesthetics, but also about ease of visual processing, as sight is the main interface for intake of computing information. Visual clarity thereby helps in getting work done, only one aspect being the goal of visual debugging of code with relative ease.

Furthermore, tini.flow aims to have sensible defaults, understanding that the ideal use-cases for this system overlap between scenarios typically seen as fitting for shell scripts and single file Python scripts: namely, tasks that take no more than a week of programmer time to implement.

An ideal user story for tini.flow is: the user prototypes either the entire workflow or critical paths in the workflow at the command line, using their shell. They do this interactively. They paste these components into a tini.flow flow file as nodes. Finally, they arrange the nodes into a workflow by defining the necessary data flow edges and control flow edges and logic.

The user may optionally improve upon their work by overloading some of the built-in tini.flow functionality. They may extend this functionality for the purpose of for example adding better logging, debugging, error handling, or reporting. This last step is wholly optional as tini.flow's defaults and built-in node/edge constructs should be designed to be sufficient for most tasks fitting the aforementioned profile. In general, tini.flow follows the principle as outlined by Larry Wall, creator of Perl, that "Easy things should be easy, and hard things should be possible" [Wall].

tini.flow is designed to create a user experience wherein easy, common types of workflows are direct, straight-forward, and simple to implement. While it should be the case that very few users will spend more than a month or more than a year continuously implementing a single workflow within tini.flow, tini.flow should not prohibit or directly discourage this kind of use.

tini.flow aims not only for a simple user-interface, but for simple implementation. In the terms of Richard Gabriel's *Worse is Better*, tini.flow implements the *worse* approach, also known as the New Jersey style. In the New Jersey style, "simplicity is the most important consideration in the design". "The design must be simple both in implementation and interface [. . .]. It is more important for the implementation to be simple than the interface".

[1] http://www.cs.dartmouth.edu/~doug/DTSS/

This *worse* approach is noted to be the approach behind extremely popular software and languages, like Unix and C, and could be argued to be descriptive of even Python.

One direct consequence is a reliance on existing tools and functionality. `tini.flow` tries to rely on the Python stdlib and third-party packages as much as possible, even where a more complex implementation might give better results.

For example, since `tini.flow` involves graph-manipulation, it relies on the NetworkX package, which is widely considered a convenient and simple toolkit for modeling graphs.

NetworkX is not without its faults, and better alternatives exist for modelling large graphs. Internally, NetworkX models graphs as dictionaries of dictionaries and suffers from serious performance issues with large, highly connected graphs. There are other packages that address this limitation, but they tend to be harder to use, are less commonly used, and add software dependencies that might be hard to meet on all user systems. Most of these have C dependencies which are not necessarily included in the package managers of common Linux distributions like Ubuntu.

NetworkX, on the other hand, is available on all platforms, is written in pure Python, and does not require compiling anything from source. In the case of NetworkX, the additional efficiency of *better* tools is not sufficient to account for their additional complexity.

Beyond the choice of libraries, the design of `tini.flow` itself shows a desire for implementation simplicity.

For example, workflow graphs containing both control and data edges could be modelled as `networkx.MultiDiGraphs` – directed graphs which support non-unique edges. In these graphs, a pair of nodes (u, v) may be connected by two edges: one which represents a control flow and one which represents data flow. Normally, the edges of a directed graph are considered unique, therefore the control edge and data edge cannot coexist. Unfortunately, NetworkX's interface for these kinds of direct graphs is very poor: there is no way to easily traverse just the control edges or just the data edges, without lots of ugly if-else statements.

As a consequence and because `tini.flow` aims for implementation simplicity, `tini.flow` models one workflow as multiple regular `networkx.DiGraphs`: one graph contains all the control edges, the other controls all the data edges. The two graphs share the same set of nodes. This formulation adapts the internal design of `tini.flow` to the implementation complexity of its dependencies in order to achieve the lowest overall implementation complexity.

One might assume that, given its focus on implementation simplicity and its eagerness to use the of standard library and third-party packages, `tini.flow` would use the `subprocess` module from the Python standard library.

This module makes many tasks involving child processes and pipes very easy to do and it subsumes multiple prior attempts to provide this functionality to users.

However, in its attempt to solve many common problems in an easy interface without loosing generality to solve uncommon and rare problems, the `subprocess` module's Popen structure has evolved an incredibly convoluted and overcrowded interface. The Popen-constructor takes over 15 arguments, which overlap in complex ways:

```
class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None,
    ↪  stdout=None, stderr=None, preexec_fn=None, close_fds=True,
    ↪ shell=False, cwd=None, env=None, universal_newlines=False,
    ↪ startupinfo=None, creationflags=0, restore_signals=True,
    ↪ start_new_session=False, pass_fds=(), *, encoding=None,
    ↪ errors=None)
```

Instead, the simplest implementation uses system calls directly. While we might miss some of the subtleties that subprocess.Popen has invariably had to consider over its many years of development, the result is still a simpler implementation. If these subtleties prove to be show stoppers, user-feedback can drive additional complexity on an as needed basis.

In this case, a side-effect of avoiding a library for this part of the code is the affordance of familiarity. This code can borrow techniques for using system calls like fork, exec, pipe, or dup2 such as those in the coreutils and moreutils packages. If the abstractions these system calls provide prove to be leaky, they will be leaky in a fashion that is familiar to users. Users can employ existing techniques to address them.

3.1.1 What Is an Affordance?

According to the psychologist James J. Gibson [2014], the word *affordance* describes the property of complementarity between object and actor in such a way that the object affords an action to an actor: i.e., an objects physical properties hints at a way to use it in certain ways and contexts. More universally formulated, the concept of affordance includes all possible transactions between an actor and its environment.

The design of everyday things, usability engineer and cognitive scientist Donald Norman [2013] universalizes this notion of the affordance from the purely physical to the abstract, by including *perceived* affordances in his definition. According to Norman, affordances may be the things – or better put, hints – which symbolically suggest how to interact with an (abstract) object, therein opening the door for the inclusion of the concept of affordance into the terminology of human-computer interaction design.

Furthermore, the specific perceptions of an affordance may vary significantly from individual to individual: A staircase may afford an average human to walk up to the next floor, but for someone constrained to a wheel-chair the same cannot be said.

On the other hand, an affordance might be created in such a way to directly map to the scope and limits of a given functionality: The steering wheel of a car cannot be turned beyond the maximum steering angle. The overcoming of a lack of functionality may be afforded as well: Imagine the cap of a water bottle, which is rimmed with a dotted, rough surface for friction in order to overcome the difficulty of opening it, if it were slippery.

In the digital realm, when we design APIs, think of how to take input from stdin, or have to supply a developer facing CLI, create schemata for the naming and organization of function calls across a large library, we have to keep the same ideas in mind.

The concept of the affordance applies here just as well: Even if or even more so in a purely text-based domain, developers have to think thoroughly on how to suggest the right actions, give the right context, and give cues as to whether a program or a function is used in the correct way.

This might not only decide about the happiness of the developer, but through time-savings – by affording developers the intended use, instead of sending them on wild goose chases on StackOverflow – will be translated directly into monetary gain and through future avoidance of bugs – that might meet the eye quickly within the right interface design – will avoid (potentially disastrous) outcomes of unforeseen and possibly public failing.

Within the context of the command line, the power of affordances is somewhat limited – the only available medium for communication being purely text-based. Yet, in this somehow scarce environment, it is even more useful when things behave in certain expected or at least coherent ways.

For example, that commands have a command name, some flags with optional arguments, possibly input and output files, and that all these follow a certain consistent ordering. Additionally, commands that read and write from stdin and stdout may be concatenated through piping.

Over time, this has created an environment in which among the group of users of a system there is an established vocabulary of usage patterns: in this case – the command line interface – handed down by the traditions of the Unix and Linux operating systems.

The more coherent these abstract text-based affordances are within a computer system, the more logically accessible and discoverable they are to the novice. And, as mentioned before, even an expert user will profit from the lower cognitive load of logically coherent affordances and the consequent higher efficiency as well as lower likelihood of human error.

3.2 What Kinds of Nails Need a Domain-Specific Hammer?

The purpose of a *DSL* (domain-specific language) is the creation of a human-computer interface that mediates between a human's ability of abstract reasoning and the computer's ability to perform complex combinatorial processing. It uses the grammar of visual text-based word symbols, with the goal of reducing the mental load of such a grammar by only including aspects and affordances that are relevant to a specific problem domain or necessary as part of given computational constraints.

In the sense of the metaphor of the domain-specific hammer mentioned in this section's title, in fact, even the DSL itself could be considered a kind of meta-affordance.

It is a concise set of affordances or handles, with which to steer and manipulate the computer into swiftly achieving goals within a specific problem domain.

By introduction of a grammar for accomplishing specific tasks, DSLs reduce cognitive overhead. This is accomplished by providing a more limited and directed grammar within an encapsulated context. Being formulated around such a more reduced logical subcontext of only a specific domain, these grammars can then be terser, more direct, and simpler than any general-purpose programming language could ever be.

In his studies on command use and interface design, Kraut et al. [1983] liken interface design to an art, rather than a science. People's natural use of computer systems is determined by patterns of use and affordances that users already know from previous use of systems.

Determining a globally optimal syntax for a DSL therefore becomes an impossible and circular task, akin to the Greek parable of Sisyphus. This doesn't mean that such work isn't necessary and valuable however. Instead, this merely mentioned to show that determining the syntax and affordances of a DSL is a very work-intensive circular task that never comes to a *finished* solution.

However, it doesn't keep us from finding a *good* solution. The artistic component mentioned by Kraut is described as being based around study of social behavior instead of the following of rigorous norms. In the same way that art can only be meaningful in the tension between metaphors of the real world, audience, and artist; the creation of a DSL is only possible in the tension of use-case, users, and creator of the DSL.

This ambiguity towards what constitutes a good DSL, combined with the inherent usefulness of the approach, has led to a wealth of DSLs throughout the history of computing, most of which will rightfully be lost to obscurity. In fact, this is well summarized by Sprinkle et al. [2009]:

> The landscape of software engineering is littered with languages that were supposedly the next great thing but failed to take the development world by storm. In contrast, many tools actually have changed the technical space in which certain domains expect to operate.

They continue to mention a wealth of both graphical and text-based DSLs that dominate their respective fields in making computing resources easily accessible for the end user, who often may not even be fluent in general programming language concepts. These include Simulink and LabView for signal processing and control, SolidWorks for building physical devices, or VHDL and Verilog for the modeling and verification of embedded hardware.

The authors go on to define the central elements that constitute the domain-specific approach and result in its superior functionality:

> In domain-specific approaches, developers construct solutions from concepts representing things in the problem domain, not concepts of a

given general-purpose programming language. Ideally, a DSL follows the domain abstractions and semantics as closely as possible, letting developers perceive themselves as working directly with domain concepts. The created specifications might then represent simultaneously the design, implementation, and documentation of the system, which can be generated directly from them. The mapping from the high-level domain concepts to implementation is possible because of the domain specificity: the language and code generators fit the requirements of a narrowly defined domain.

3.3 Design of the `tini.flow` DSL

In order to provide maximum functionality in the smallest package of grammar, `tini.flow`'s approach is to implement an embedded DSL (eDSL). In other words, it composes a specific grammar for constructing graphs into the general purpose grammar for constructing Python programs.

3.3.1 The Evolution of the `tini.flow` Graph-format

`.flow` is not purely a document format; it is a computational format because it is embedded into a Python script.

A consideration in the design of the `.flow` DSL is that it intends to intermingle direct definitions of edges and nodes with code-defined ones. The format is actually a variant of a Python script, and this feature can also use this to the user's advantage in the definition of nodes and edges.

For example, the following is a 1001 node graph that is built up using computational means:

```
# concise description of 1001 node graph
* tail = seq 1 10
for _ in range(1000):
    * node = ts
    % tail = tail | node
```

`tail` is a node that creates the sequence from 1 to 10. Create a new node with the command `ts` in a loop and attach it to the end of the result by capturing the result as the new tail.

Clearly, writing this graph manually would be tedious and difficult for a user.

A `tini.flow` `.flow` file appears to be a document format, but actually contains executable code, which may make some users wary. However, this is not a such a rare case. Many file formats that are generally perceived as document formats are actually Turing complete, like PostScript and LaTeX.

For example, this thesis is written in LaTeX. It can execute code, including running arbitrary Shell commands, with the `--shell-escape` flag and the `\immediate\` ↪ `write18`.

Of course, it could be argued that the need for a command line flag to add arbitrary execution to LaTeXindicates the presence of a restricted computation environment. TeX itself is Turing-complete. However, under normal circumstances, it is limited in its ability to do stateful or destructive system actions.

In recent days, the Microsoft Excel team has publicly requested comment on adding Python as a scripting language to Excel [Sharma, 2017]. As it stands, Excel documents can contain arbitrarily complex VBA macros which have presented a similar security consideration to users, expecting document viewing to be completely benign.

A Python script has access to built-in system functionality and a standard library that allows for many stateful and destructive actions without explicitly signalling this capability. That said, there is no theoretical barrier to creating a restricted Python runtime, which could provide a similarly restricted – *safe* – computation environment.

Python, the language, does not require arbitrary system level access to operate. That such a mode does not exist is merely a result of the lack of economic incentives and unsure requirements for support of third-party code. Additionally, in practice, more practical alternatives exit (i.e., Docker-style containerization).

Even though history is littered with Python sandboxing attempts and Python 2 even included an unsuccessful and since abandoned restricted mode – there is simply no theoretical barrier to making this work.

In fact, *PEP 551 – Security transparency in the Python runtime* Dower [2017] has begun some of the work necessary to making this possible. And, of course, were Python to be included in Excel as a scripting language like VBA, it is almost certain that the economic incentives would flip and Microsoft would develop exactly the kind of restricted Python execution mode that would make Excel documents safe to view and `tini.flow` graphs safe to evaluate.

The design of `tini.flow`'s DSL followed an iterative thought process, considering many different options before deciding on a Python-based eDSL.

The thought process started by considering two varieties of formats: static and dynamic.

Graphviz/Dot

An obvious choice for the use of graph format within `tini.flow` would be to borrow from an existing format. One graph format that is very commonly used is the Graphviz/Dot plain text graph description language. The format comes with a rich toolchain that for example enables easy visualization.

NetworkX even supports reading and writing graphs directly to this format, and this functionality is even used by `tini.flow` for visualization purposes.

A Graphviz/Dot-based format might look like:

```
graph {
    a [seq 1 10]
    b [grep 3]
    c [ts]

    a -> b -> c
}
```

This approach has the following advantages:

- no additional design is needed
- no additional learning-curve for users who already know this format
- no responsibility for `tini.flow` to teach users about a new format
- an array of tooling already exists for this format

This approach has the following disadvantages:

- if this format requires any amendments, a new parser for it would have to be written from scratch
- if this format requires any amendments, these could lead to serious incompatibilities with tools and different dialects of the format
- any incompatibilities would immediately reintroduce a learning curve for users and the responsibility of `tini.flow` to teach users about these differences
- the format is simply not “programmatic”
- the format might require escaping, since node contents should be arbitrary shell snippets
- escaping offers very poor affordances for users (e.g., the backtick-character ` in shell has been a constant problem for users)

Completely Custom Format

A completely custom format might be another option, This format could take many forms.

Some questions arise when devising a brand-new format. For example, how similar should a completely custom format be to something that already exists? Too similar and you have a false promise problem, creating cognitive dissonance for users relative to the similar language. Too different and you are potentially unnecessarily redoing design work that somebody has already done.

For the purpose of evaluating this option, consider a moderately different syntax to Graphviz/Dot.

```
graph {
    var root
    for command in "seq 1 10" "grep 3" "ts"; do
        var node = [command]
        root -> node
        root = node
    done
}
```

This approach has the following advantages:

- total flexibility in the general design and specifically the chosen affordances

This approach has the following disadvantages:

- requires writing a parser from scratch
- no tooling readily exists
- the learning curve for users will be high for a completely fresh approach
- there is a risk of design errors (coding oneself into a corner)
- large amounts of time required to validate and iterate on a design
- as the format introduces programming functionality, have to design not only a data format but also a programming language
- designing a programming language introduces enormous design considerations from the start (e.g., if it includes for-loops, does it include while-loops?)

JSON/XML

Consider the problem of writing a parser from scratch.

Using XML or JSON document formats would be another option, and it might remove the need for writing a parser:

```
{
    "nodes": {
        "a": "seq 1 10",
        "b": "grep 3",
        "c": "ts"
    },
    "edges": [
        ["a", "b"],
        ["b", "c"],
    ]
}
```

This approach has the following advantages:

- parsers and tooling already exist (including validators)
- there is an easy learning curve for learning the syntax

This approach has the following disadvantages:

- tedious to write by hand
- XML and JSON, per se, are not specialized to our use-case
- have to design the JSON or XML schema
- users still have to learn a schema
- code still has to validate to the schema

In fact, these formats actually do little more than what could be considered tokenizing. The parser complaint from previous approaches is more fundamental than the superficial validation offered by JSON and XML. Therefore, many previously mentioned problems persist.

Using Code for Data

Programmability is an important feature for concisely defining graphs. Business workflows have a lot of repetition, as they are often driven by fairly simple user input. For example, performing the same action on various input files, with only slight differences based on which file. This repetition puts a burden on users writing data files by hand. Just as programmers cannot abide by the tedium of writing assembly and instead rely on higher-level programming constructs, so can a data format allow programmatic representation of information that would be too tedious to write out.

In other words, *programmability* describes mechanisms by which a user can dynamically encode graphs in shorter form than their static or unrolled equivalents. This is especially useful for allowing data definitions that rely on some dynamic context: e.g., create a node for processing each file that sits in some directory. By adding dynamic mechanisms to the data format, it can rely upon runtime state for more concisely encoding.

It is possible to separate the static and dynamic sides of the data format into two parts. It is possible to leave the format itself static and provide an external tool to perform templating or processing. In fact, this is a popular approach historically, and tools like m4 have been developed explicitly for this use-case.

Unfortunately, this approach requires users be intimately familiar with both the data format and the templating language.

These templating languages are often very limited; they can be text or macro-based and do not provide the programming language functionality necessary for handling complexity.

Additionally, they introduce operational complexity, such as the need for a *build* process. In the case of LaTeX, (which is already a dynamic language) these build processes can be very clumsy and error prone.

A data format can also support dynamic behavior directly. A good deal of functionality in TeX is implemented in this fashion and requires no external build process. Of course, while this has the advantage of being operationally simpler, TeX files are known for their inscrutability.

`tini.flow` tries to take the TeX model of dynamic programmability of data and improve upon it. Criticisms of the complexity of this approach have been considered, but the complexity seen in TeX may not be a consequence of this choice. After all, large legacy Python code bases face the same complexity concerns. It may be that the complexity is inevitable in any large dynamic system with the same scope, popularity, and long history.

(When discussing some aspects of `tini.flow`'s design with colleagues an unfavorable comparison with TeX was made. The author's response: If only `tini.flow` could be so successful!)

Before reviewing `tini.flow`'s approach, consider the question of: why not just write an actual Python script in the first place?

Writing raw Python has the following advantages:

- an abundance of tooling already exists
- Python is already popular and wide-spread
- `tini.flow` assumes relatively little responsibility for teaching syntax
- the approach is very flexible, because Python is a general purpose language

Writing raw Python has the following disadvantages:

- this choice might tie `tini.flow` to a specific set of dependent libraries (like NetworkX)
- this is relatively tedious
- there is still an API-design problem and API learning curve

The above considerations lead to the consideration of a DSL format. This format would give the user the most bang for the buck for this domain-specific use-case.

When creating a DSL, two options exist: creating a full-fledged *external* DSL with its own interpreter or an *embedded* DSL, that integrates into Python in some way and is fully grammatically compatible with Python.

The first approach, a full-fledged external DSL (not necessarily grammatically compatible with Python) with its own interpreter, would have significant benefits. It is a completely flexible approach with 100% control over the format. The DSL could be constructed to restrict or prohibit insecure code.

Unfortunately, this approach requires even more design and implementation work than any of the previous alternatives! `tini.flow` would have to have a completely custom parser and interpreter. This creates a huge risk for mistakes in both design and implementation. Additionally, a full-fledged DSL would mean a steep learning curve, since users have to learn a completely new tool from scratch.

`tini.flow` adopts a hybrid approach: it creates a DSL embedded into Python. This approach allows `tini.flow` to have full access to Python internals, and it allows users to *drop down* into Python where the DSL functionality alone is insufficient. This provides an *escape hatch*: errors in the design of the DSL have immediate workarounds by dropping into raw Python.

This approach also enables a relatively smoothly progressing learning curve: users only have to learn the additions to the host language – the rest is standard python.

Since Python does not supply us with an extensible parser, `tini.flow` has two main options for implementing an eDSL.

In the first, it would implement a full parser. The full parser approach would give more control over how the eDSL extends the Python language. It would mean that the only theoretical limitations restricting eDSL design aspirations from a language standpoint would be interoperability with the rest of the Python syntax (and that the DSL syntax has to be parsable and unambiguous).

It would also mean implementing a full parser from scratch. This is not prohibitively difficult, but it can be time-consuming to do correctly, even with the help of parser libraries. More notably, the Python language changes from version to version. With a completely custom parser, the burden of maintaining the parser is on `tini.flow`. Furthermore, Python maintains parallel release versions (2 vs <3.6 vs >3.6), and a custom parser would have to maintain interoperability with all supported versions.

Alternatively, a much simpler approach is line-level regex approach. This has limited flexibility, since it can only integrate with the Python statement grammar. The Python grammar is a context-sensitive grammar and cannot be parsed by regular expressions. However, lines in the statement grammar can be transformed by simple regular expressions with high accuracy, because this syntax is generally restricted to a single line.

Strictly, applying regular expressions to Python on a line-by-line basis will lead to incorrect behavior, because a Python-scripts can contain a multi-line string and the beginning and end of the string may have appeared in previous or later lines.

If we ignore this possibility or provide a simple escaping mechanism, we can parse a Python script on a line-by-line basis by looking for lines that begin with a character that is an invalid start-of-line in Python and then transforming that line into a Python statement.

A Markdown for Graphs

Graph structures are surprisingly common in use. They are one of a set of critical structures for computing.

Recent computing research has focused on other structures, columnar data structures, as needed by *machine-scale* deep learning and data science problems. These data structures are *machine-scale* because their data is often not written by a human being. In contrast, a *human-scale* structure would be one that a human would typically enter manually.

In computer administration tasks and even in many data-driven computing tasks, human-scale graph structures are still very common. These include systems like payroll/accounting software, build processes, and business flowcharts/workflows.

The `.flow` format is strictly a human-scale format. For machine-scale uses, other formats may be superior.

The deep focus on programmability when designing the DSL and the `.flow` format is driven by the desire to ease human authorship of these files.

In this sense, the DSL could be viewed as a *Markdown* for graphs. In other words, graph definition formats (like Graphviz/Dot) already exist.

But these are hard to write from memory and are tedious for repetitive graphs. Just as XML and other markup formats exist in wide-use, they are poorly suited for describing quick, ad hoc data structures in a user-friendly fashion.

The structure behind the `.flow` format may be critiqued as *imperfect* and *informally specified* but is obvious and approachable for describing ad hoc graphs. This mirrors Markdown's reputation as an imperfect, informally specified document markup language.

In fact, Markdown was originally specified in a blog post, driven by its authors frustration with the clumsiness of alternative tools. This is the spec that people are still following today. Despite being an informal definition, it works.

3.3.2 Line-level Statement-grammar Composition

`tini.flow` transforms Python scripts on a line-by-line basis, replacing special DSL lines with Python statements.

`tini.flow` identifies a DSL line by looking for a special character, – henceforth called a *sigil* – that cannot at the beginning of a line in a valid Python script. (This is true only outside of rare cases, such as in multi-line strings or in lines with inappropriately placed line-continuation characters.)

Python statements include `def`, `class`, `for`, `while`, `if`, and `with` and must appear on a line by themselves with only whitespace before them. Note: it is possible to put the body of a block statement on the same line as the introducing statement, but only if the body is a single line. This appears relatively rarely in practice and is discouraged by most style-guides.

Having identified a line as being invalid Python, the DSL-parser can then parse this line in whatever fashion it desires, ultimately transforming the line from a custom grammar into a Python statement.

The `tini.flow` DSL uses the following four characters as start of line markers. None of these characters can appear at the start of a line in a Python script, outside of a multi-line string or an expression with awkwardly placed line continuation. These sigils can be escaped with a single `\` in the rare case that they appear in the line of a multi-line string.

The characters:

A `*` at the beginning of a line indicates that the line encodes a graph node. The `*`-character resembles the representation of a node in the drawing of a graph. Following the `*` is any sequence of characters, except for an equal sign, then followed by an equal sign, and finally any sequence of characters.

This line is transformed into a Python assignment statement. On the left-hand side of the assignment statement is whatever characters were on left-hand side of the equal sign in the original text. The character on the right-hand side of the equal sign in the original text are surrounded by quote characters and are passed as the first argument to a function called `__node__()`.

For example, `* a = ls` is transformed into `a = __node__('ls')`. The argument to the node function must be a string as it can represent any arbitrary shell expression. It is the responsibility of the node function to parse this string, if it needs to perform further transformations.

A `%`-sign at the beginning of the line indicates that the line encodes an edge. The `%`-character could resemble two nodes with a line between them, implying the meaning of edge. Following the `%`-sign is a sequence of characters, then an `=`, then an expression, delimited by a special node separator pattern. This line is also transformed into an assignment statement.

The left-hand side of the assignment is again whatever was on the left-hand side of the equal sign in the original text. The right-hand side of the equal sign is a function call to a function called `__edge__()` with the following arguments. The contents of the original line are split on a special node separator pattern and then comma-separated and placed within parentheses.

The original text must encode a set of valid Python expressions, delimited by the node separator. The node separator pattern is usually `['-’]|`. The actual separator text found in the original text is collected into a list, which is passed as a keyword argument `seps`.

For example, `% e = a - b - c` would translate into `e = __edge__((a, b, c),` ↪ `seps=['-','-'])`. The edge function is responsible for processing its arguments. In this example, the edge function might end up creating two actual graph edges (a, b) and (b, c). The choice of node separator allows the edge function to easily distinguish between different types of edges, such as data vs. control edges.

Because control flow requires the encoding of triggering information, an amendment to the above grammar was later made. In addition to the node separators, node data and edge data separators were defined. These were defined as a doubling of the node separator, i.e. `--` or `||`.

Before the above processing is performed on the right-hand side of the original text, a splitting is performed on this node or edge data separator. The first element is processed as described above and all later elements are passed as additional positional arguments, without modification.

For example,

```
* a = ls    -- wrapper
% e = a - b -- success
```

would be transformed into:

```
a = __node__('ls', wrapper)
e = __edge__((a, b), success, seps=['-'])
```

The node and edge functions are responsible for processing the additional data passed. To avoid conflicts with the use of the node separator characters and the contents of the edge and node arguments, the `$`-sigil allows changing the node separator on an as needed basis.

A flow file may choose to contain multiple workflows. The `>`-character indicates that the following indented block represents a new named graph. This line is translated into a `with`-statement, where the original text contents are passed as a string to a function called `__workflow__()`.

For example,

```
> wf
  * a = seq 1 10
  * b = grep 2
  % e = a | b
```

would translate to:

```
with __workflow__('wf'):
  a = __node__('seq 1 10')
  b = __node__('grep 2')
  e = __edge__((a, b), seps=['|'])
```

Care is taken to perform the line-level transformations without adding or removing lines. Other code generation techniques use source maps to map the resulting transpiled code to the original input. These source maps are important so that errors can be tracked to their source.

By keeping a one-to-one correspondence of original to transformed line, we can avoid the need to create a source map. When encountering a bug in the transpiled code, the user can refer to the same line number in their original input.

This line-by-line approach is easy in Python, given Python's use of significant whitespace. It would be harder to do in a language like C++ or Java.

Like in other scripts, the first line must be a shebang line. This is because this way any flow file can be run as a script and also, much more importantly, we have one line that we can replace with the imports to run the script.

3.3.3 Worse is Better

Criticisms of Unix and the C language are numerous: From the *Unix Haters Handbook* [Garfinkel et al., 1994] to Richard Gabriel's *Worse is Better*. But the widespread popularity of these tools is undeniable and their criticisms have been reinterpreted as a modern parable embodying the aphorism *perfect is the enemy of good*.

Tools written in the so-called *New Jersey style* have won. They have captured the interest of programmers in spite of their flaws. Indeed, it appears that worse is in fact better. Obviously in the design of `tini.flow`, choices had to be made between different potential approaches, and it is tautological that a truly worse choice would not be taken over a truly better choice.

However, in the design of the DSL a less strict and more limited approach was taken, and it deserves explanation why these kinds of decisions would be made. Understand that this proof of concept approach can be easily revisited in later versions.

In the presence of unlimited time and unlimited money, a division of programmers could address every downside of the custom parser approach. This would be strictly more powerful and flexible than the `tini.flow` approach. As a practical matter, few projects start with a blank check. `tini.flow`'s approach is intended to be a quick and easy stop-gap that is well within the capability of a single programmer that could eventually be extended into the traditional approach as additional resources can be allocated to the project.

All too often when discussing software design, we discuss the design of the platonic ideal of a finished system that was the work of an expert team, presuming no limits or unrealistic limits on the resources needed to build this system.

But real world projects often start with small, imperfect prototypes that developers imagine will be rewritten when the system goes into production. It is not uncommon for prototypes to not be rewritten but merely to be extended, polished, hacked upon until they are deemed production ready.

When discussing software architecture, it is critical to understand these realities and the `tini.flow` DSL approach is one which tries to coexist with these imperfections in organizational process.

It is a technique that is not expected to live forever, but is expected to be good enough that it can survive through the initial phases of a project and can be easily extracted and replaced, when developers are tasked with coercing their initial codebase into something their managers can call ready for deployment.

Again, note: a full fledged DSL with its own custom parser is not at all fundamentally difficult to write. With a full-fledged parser `tini.flow` could create its own error-checkers or source mapping mechanism, so that errors in the resulting python could be tied to errors in the given input file. However, under time constraint this could be error-prone and could involve a lot of code.

That said, the technique used for .flow is a very general technique: It can be implemented quickly, in an ad hoc fashion that requires significantly less effort. This follows the KISS principle and a similar one: *if its stupid and it works, it's not stupid*. In other words, given a team of graduate students and a year, one could easily develop a full tool chain for an arbitrarily complex DSL and users would be able to use this as conveniently as they can use any other similarly mature tool.

However, the technique used by tini.flow is a generalizable technique for the realistic case, that a single programmer on a deadline will need to be able to create a tool that is robust, convenient, and transparent on a strict deadline.

This opens the possibility of doing a kind of *micro-DSL*, that can be quickly and reliably developed. It is a software engineering technique to get powerful results quickly. It is more of an engineering approach than purely scientific.

Of course, this approach is not without limitations. For example, it cannot disambiguate certain structures without the need for escaping (such as multi-line strings) which would not be the case in a full-parser approach. This is because the tini.flow parser cannot *understand* (i.e., parse) Python expressions: it can only look for very specific patterns.

To illustrate, without escaping:

```
helptext= '''
    To define a node, write
    * a = 10
    '''
```

would be incorrectly translated to:

```
helptext = '''
    To define a node, write
    a = __node__("10")
    '''
```

3.3.4 How Much DSL is Enough?

As part of the design process, a natural limit must be formulated on how much grammar the DSL should contain.

As a practical matter, too much DSL grammar means too much time developing the DSL, too much complexity, and too much of a learning curve for users.

However, too little DSL grammar means that users will be forced to drop out of the DSL grammar. If users drop into Python too readily, then it reduces the usefulness of the DSL and might lead users to abandon it altogether.

Therefore, there must be a sweet spot of just enough grammar to keep users or some percentage of their work in the DSL, but not too much that the DSL becomes a full-fledged language.

As a guideline, tini.flow implements just enough DSL to make the examples listed in this thesis convenient to read and write.

However, for a production release of this tool, it would make sense to set a more explicit design goal. This might be formulated as follows. Avoid dropping the user into Python for 80% of common flow files under 200 lines of code.

With a design guideline like this, `tini.flow` could aim for just enough DSL to meet whatever the immediate business use case might be.

Since the DSL approach is intended to be a prototyping of a more general engineering technique, it is strongly suggested that others attempting to employ this technique make up-front design decision of this form.

Otherwise, the risk may be too great that this imperfect technique, designed to save programmer time, may not only waste as much time as learning and using a full-fledged DSL design may take, but also result in an overall worse product.

3.4 Design of the Execution Engine

A `tini.flow` workflow consists of a set of nodes representing actual Unix processes and edges representing data flow and control flow connections.

`tini.flow` represents these workflows as multiple NetworkX directed graphs, where every graph contains the exact same set of nodes and only differs in whether it contains the data flow edges or the control flow edges. Each graph object can contain only edges with the same semantic meaning (control or data flow).

In software engineering, there is a *zero-one-infinity rule* [MacLennan, 1999]. This principle suggests to programmers to pay attention to software constructions in which the count of some artifact is not zero, one, or infinity (infinity meaning any value greater than one, but without distinguishing between these values). These software constructions may be unnecessarily over-specified.

`tini.flow`'s design is intended to be practical and therefore generalizations and abstractions are not considered to be a priori desirable. While this is the case, `tini.flow` does want to avoid design errors from code that is unnecessarily over-specific, such as code that employs *magic numbers* – meaning specific constants instead of simple parameterization with sensible defaults – to maintain the same simplicity of interface. When considering the case of these different graphs, the zero, one, or infinity principle appears to be violated, because we describe a workflow as having two graphs: one for control and one for data.

`tini.flow` simplifies its design and implementation as follows.

The input to the execution engine must be a collection of graphs, representing the workflow. There must be at least one graph in this collection. This first graph is not optional and it represents the data flow information. This graph is not optional because `tini.flow` cannot operate without knowing which processes are connected to what stdin/stdout, even in the case where every process is just connected to the console stdin/stdout.

In the Unix model, standard-in and standard-out have special privileged meaning. `tini.flow` acknowledges this special meaning in its design. Following the zero, one, or infinity principle, the other graphs in this collections are not given any specific, privileged meaning. Any other graph merely specifies a collection of edges that connect some of the processes. Note, for simplicity's sake, `tini.flow` mandates that every graph contains exactly the same set of nodes, even if some of those nodes are completely disconnected.

These additional graphs may represent control flow, they may represent additional data flow, such as for logging, or they may represent some other message passing mechanism.

The execution engine merely creates pipes and tees, in order to connect nodes on one of these graphs, injects environmental variables, so that node's processes have information regarding which information pathways they appear in, but it leaves the actual handling of these pipes to the process itself.

The following examples mostly show workflows with data and control edges, since that is the most common case, but `tini.flow`'s design allows us to effortlessly generalize to any number of information pathways in addition to the mandatory data flow graph.

3.4.1 Creation & Annotation of the Process, Pipe, & Execution Graphs

The `tini.flow` execution engine has two responsibilities. The first is to take a workflow description in the form of a set of graphs and identify what processes would need to be created and what additional pipes would need to be created in order for it to be executed. The second responsibility is simply to execute the workflow.

In this sense, the `tini.flow` execution engine is relatively simple and its implementation reflects this simplicity. `tini.flow` is designed in the *Worse is Better* style in which simplicity of implementation is paramount.

Executing a workflow is not as simple as creating one child process for each node in the original workflow description. First, because the execution engine is not written in shell, connecting the stdin and stdout of piped processes requires explicit action: in `tini.flow`'s case, this means the use of anonymous pipes and the `pipe` system call.

In the case of a non-linear data graph, one output end cannot be connected to two input ends without loss of data. The only assumption for the inter-process communication (IPC) mechanism, that represent edges in a `tini.flow` graph, is that these constructs be read-once FIFOs.

As is the case with anonymous pipes, once data has been read from them, that data can never be read again. Therefore, at every branching point within the graph, `tini.flow` must introduce a new node that performs a tee operation. This operation does the same work as the coreutils `tee` utility. (In the `tini.flow` proof of concept it is actually implemented with coreutils/tee.)

The execution engine identifies all the locations where these tees must be inserted by transforming the input graph into a new graph that includes the inserted nodes. Any node that has more than one ancestor nodes must have a tee immediately after.

In the case of a node with two predecessors, no tee is necessary. Instead, both predecessor nodes write to the same FIFO read-end for their ancestor process. Because `tini.flow` workflows can include communication channels in addition to those used for connecting stdin and stdout, it must repeat this process for each set of edges – data, control, and so on – and make sure to create FIFOs for every edge.

Some effort is put into making sure to create the minimum number of FIFOs in order to be able to execute workflow. Additional unused FIFOs each require two file descriptors. File descriptors have high, but non-infinite, resource limits.

For debugging and display purposes, `tini.flow` creates supplementary graphs that allow users to visualize all of the extra tee nodes and extra FIFOs that are necessary in the execution of the workflow.

Note that when connecting pipelines, `tini.flow` only ever connects the stdin and stdout of two programs connected on a data edge. The stderr is untouched. Typically, this means that the stderr will remain connected to the console. If there are errors in any nodes and these are correctly reported on the stderr, `tini.flow` wants to make sure that it doesn't accidentally hide these errors.

(Future releases of `tini.flow` may also provide functionality to tee the standard-error and write it to its original file descriptor as well as some additional location. For example, it could send the error messages for every node to a special logging facility, like journald or Zabbix, such that error messages for individual nodes are correctly collated and tagged for easy systematic debugging.)

3.4.2 Executing the Workflow

The `tini.flow` execution engine takes input representing a workflow and executes it on the current machine. Workflows are encoded as a set of graphs, in which the nodes are the system-level processes to start and the edges indicate how the processes communicate with each other.

A system-level process is any command line invocation that is valid at the shell. This may be a shell script, as in the case with a wrapper script, a binary-like `seq` or `grep`, a string of shell-builtin commands, or any other command that the shell can run.

The execution engine merely determines how to intelligently connect these processes, so that data flows as specified and so that these processes, when started, will implement the desired workflow. The execution engine targets the Linux platform for evaluation of these workflows, but future additions to the execution engine could easily support any platform that offers the following facilities:

1. the creation of child processes
2. the creation of uni-directional, read-once, binary data communication channels
3. the ability to specify these channels for reading and writing for launched processes

These facilities map cleanly to the POSIX model, as implemented on Linux, in the following fashion:

The *fork system-call* allows a process to fork it's execution. Upon calling fork, the operating system copies the current process into a new process, hereby denoted the *child* process. The original process, hereby denoted the *parent* process, continues execution from the point of the fork call. The parent process is provided the process id of the child. And this relationship is maintained by the operating system and the child can determine its parent via the getppid system-call.

Typically, all user processes will have PID 1 as their ultimate ancestor. In the child, `tini.flow` will execute an actual system-level process corresponding to the node in the workflow graph. Thus, a linear workflow that contains n nodes should appear on the system's process tree as one parent process, the flow engine, and n child processes, the actual commands required by the workflow.

These processes are started simultaneously, just as they would be in a typical shell pipeline operation, such as `seq 1 10 | grep 3 | ts`. As a consequence, a workflow in which nodes represent entities which must only be invoked upon some control trigger – most likely, because they perform stateful or unsafe changes, as opposed to mere data transformation – must not be run directly. Instead, `tini.flow` makes use of wrapper processes, which are simple shell processes that are safe to start as they defer the ultimate invocation of the underlying system process.

In the case of a non-linear workflow of n nodes, there should be $n + m$ child processes, where the m child processes would be processes created in order to tee one output file descriptor to two or more input file descriptors.

In order to launch the child process, `tini.flow` simply uses the execve system call. This system call replaces the currently running process image with another image in memory, retaining certain details of the original process, such as its open file descriptors. `tini.flow` uses the execvpe interface to the system call for the following reasons:

The argv of the new process must be carefully and precisely controlled to avoid mistakes arising from reprocessing/resplitting of arguments. This requires the execv* family of interfaces.

`tini.flow` must be able to run any process that is in the user's path, without requiring that the user needs to specify the full path. This is a convenience for `tini.flow`'s users and requires use of the exec*p* of interfaces.

Finally, `tini.flow` needs to set environmental variables for the child process. These variables tell the child process what additional file descriptors it has been given. These file descriptors map to non-standard – i.e., non-standard-in and non-standard-out data channels or control channels. For control channels, the child processes would discover them in environmental variables `TF_CTRL_IN` and `TF_CTRL_OUT`.

This requires the use of the exec*e family of interfaces. Our only choice for the interface to the execve system call is execvpe.

Once all of the child processes have been launched, the parent process has one final responsibility: the parent process uses the waitpid() function call to wait for one of its child processes to terminate.

Upon termination of a child process, the parent closes the write-end of any file descriptors used by that child process. This is necessary to trigger termination of the pipeline.

In a data flow pipeline, each node will typically continue to process data or it will block on reading from its stdin, until the stdin reports that it has reached EOF.

It is necessary to close the corresponding writing end of a pipe, so that its reading end will indicate EOF, otherwise processes may not terminate.

Many processes employ buffered I/O – by default, Python buffers all I/O, when using its high-level interfaces. This may result in no output appearing at the console. The processes may only flush their output once they have finished reading from their input end.

In order to create uni-directional read-once communication channels, `tini.flow` uses the pipe system call. This system call creates a communication channel with a singular read-end and a singular write-end. The communication channel can transmit any arbitrary binary data. It may block if too much data is written to it without that data being read out.

The degree to which this channel can buffer data is determined by the system, but for simplicity, `tini.flow` will expect that any processes that require buffering will take that responsibility onto themselves. These communication channels are referred to as anonymous pipes, and they are the standard mechanism by which inter-process communication occurs between the pieces of any shell pipeline, such as `seq 1 10 | grep 3 | ts`.

Because these channels are read-once, if one process is outputting information for more than one process to read, `tini.flow` must insert a process to tee this data: Typically, the tee process provided by the coreutils package. coreutils' tee is not platform specific. Therefore, it accomplishes the above by copying data, which has an associated performance penalty.

On Linux, there is a Linux-specific system call, called tee that `tini.flow` could use to create a higher performance version of the coreutils tee utility, because it does not perform any data-copying. The `tini.flow` proof of concept has opted to leave optimizations such as this for later releases, as these may introduce additional implementation complexities. Additionally, the coreutils tee utility is in broad use and is widely considered good enough for most production use cases.

The responsibility of the execution is two-fold. Its job is to determine what processes would need to be created to implement a workflow and how to wire up communication channels between these process, so that data and control messages are routed correctly.

The execution engine is also responsible for actually starting these processes, creating the channels, and wiring them up. The final system call that is necessary to compile the execution engine is the dup2 system call.

The dup2 system call operates on any file descriptor and allows one to create a copy of that file descriptor as an alias. This is typically, how the Unix shell implements input and output redirection: Simply, for a given process `tini.flow` uses dup2 on the end of the pipe that is connected to the previous process, according to the wiring setup generated by the execution engine, to either file descriptor 0 or file descriptor 1, in order to use these as the new stdin or stdout.

`tini.flow` uses the `set_inheritable()` function to mark new file descriptors as inheritable by child processes. Note that since Python 3.4, file descriptors created within Python are not inheritable by default [Stinner, 2013].

For the purposes of the proof of concept, the executor directly executes the workflow using the mechanism described above. However, these mechanisms are commonplace on different platforms, with a variety of interfaces for them. Later versions of `tini.flow` may choose to split the responsibility of the execution engine, such that the execution engine can either directly execute the workflow or simply emit code that when evaluated would execute the workflow.

In this latter mode, the execution engine would be responsible for compiling a `tini.flow` workflow into code that executes the workflow. The most probable compilation target would be an sh shell script – as opposed to a bash or zsh script.

This script would likely be a block of lines, whose responsibility would be the creation of a named or anonymous pipe or other unidirectional read-once FIFO-channel, for each edge in each control or data flow subgraph and a block of lines for executing each process, corresponding to each node with the appropriate input/output redirection and environmental variables.

By compiling a `tini.flow` workflow to a shell script, users would be able to create complex workflows that they could run in environments that would be very sensitive to dependencies like Python. This includes security-restricted environments, containers, or low-overhead environments, such as those on micro-controllers or even the Linux initramfs environment.

This is a very useful niche for `tini.flow`. These are environments in which current practice is the use of shell scripts, because these environment cannot support high-overhead interpreters, like Python or even Make, but in which developers cannot tolerate the slow speed of development and relatively unforgiving nature of compiled bare-metal language, like C and C++.

`tini.flow` would offer an alternative that is richer than shell scripting for construction of complex workflows without additional dependencies or overheads. Note that a compiled `tini.flow` workflow would not be incompatible with existing shell scripts. It would be entirely possible in these environments, to use `tini.flow` in a surgical fashion to extend existing shell scripts.

3.4.3 Wrapper & Helper Scripts

When `tini.flow` executes a workflow, it launches every node process simultaneously. If these processes perform some streaming data transformation, as in the case of a pure data flow program, this behavior is correct and matches exactly how the linear pipe at the shell behaves.

However, in the case of a workflow with some control flow, the nodes may represent non-data-transforming programs that potentially perform some stateful action.

According to the workflow, this process may be run only upon meeting some condition and unlike a data transformation process it may not run for the lifetime of the workflow, but instead may run once per trigger only when the right conditions are met.

Imagine a node that deletes files or creates directories. Obviously, `tini.flow` can't execute this node immediately upon starting the workflow, as these processes have stateful consequences. To resolve this, `tini.flow` includes simple wrapper scripts.

At their simplest, these scripts are simple shell or Python scripts that take the node's command line as an argument. These scripts may listen to the control channel in a loop and execute that command line only upon the correct conditions being met.

These scripts are safe to start, as the underlying functionality of the node is executed only when the wrapper script determines that the right conditions have been met.

Simple wrapper scripts include always-once, success-once, and failure-once. These scripts listen to a specially formatted message transmitted on the control channel. This message is a single line that gives the return code, PID, and command line string upon termination of predecessor processes.

Upon termination of processes wrapped by these wrappers, they generate a status message of this form to send to their ancestors. Inside these wrappers, they wait for a message of this form and trigger the underlying command line only once for the lifetime of the workflow and in the case of always when any status line is received or in the case of success and failure when the status line indicates successful or unsuccessful termination.

In addition to these wrappers, there should also be wrappers for always-success-failure-loop and always-success-failure-ntimes to behave in a similar fashion, but allow the underlying node to respond to control messages an unlimited or specific number of times before terminating for good.

In the case of a node with multiple predecessors, these wrapper scripts will have to indicate whether the predecessor nodes must all terminate and send a specific conjunction, disjunction, or other logical combination of success or failure codes.

For example, a node which starts when either one of its predecessors signals success, behaves differently than one, which starts when one predecessor signals success and the other signals failure. The exact structure of these status line messages depends on how the wrapper programs are written and in the proof of concept, `tini.flow` implements the simplest encoding of this information.

In the proof of concept, `tini.flow` implements a subset of all wrapper programs to motivate some simple demos. However, later releases may improve the status line encoding and may implement a more complete collection of wrapper processes.

The exact structure of these wrapper scripts, for example whether always-once or always-loop are distinct script files or a single script files that just takes a command line flag (-n1 and -nloop), is considered an implementation detail that can be refined in subsequent releases without meaningfully affecting the overall design of this system.

Note that it is not uncommon in these cases to write one single master script, then symlink it to various filenames and have the master script inspect it's $0 to determine the mode it is operating in, without having to specify this information via command line flag.

A production-ready `tini.flow` should come with wrapper scripts that cover the majority of common triggering scenarios. This likely includes triggering on always-success-failure, based on return code, triggering on found-not_found, based on stdout or stderr contents, triggering on other observable stateful changes, triggering once, n-times or every time for the lifetime of the workflow.

These scripts must implement the Cartesian product of these conditions in order to be considered reasonably complete. By keeping the control messages simple and this initial library of scripts simple, `tini.flow` should encourage users who need even more sophisticated triggering actions to write their own wrapper scripts.

One design difficulty is how to specify the appropriate wrapping behavior via the `tini.flow` DSL. Recall that in the `tini.flow` DSL, users are able to specify additional arguments to the node and edge functions and these additional arguments typically encode some concept of metadata.

In the proof of concept `tini.flow` nodes contain all information relevant to the workflow, including trigger conditions and `tini.flow` edges only contain information relating to connectivity of IPC. In the DSL, users may then specify the trigger conditions as node metadata or they may specify them as additional arguments to the edge function, which the prologue can then transform into node metadata.

Given the richness of triggering conditions, the prologue may need to be responsible for intelligently determining the appropriate wrapper scripts, given a combination of specified node and edge metadata.

3.4.4 Prologue & Epilogue Code

The design of the `tini.flow` DSL asserts that function calls, classes, and context managers are pretty much all that is needed to implement most DSL functionality.

This is not necessarily true for all DSLs. However, many DSLs are about the construction of objects of a custom type. Therefore, they can be decomposed into method calls, OOP-style statement management, and contexts. In Python, this means classes, context-managers and function-calls.

The DSL is basically an interface for creating a graph, its edges, and nodes. Therefore, the DSL-grammar maps directly to function-calls for node creation and for edge creation. This makes it easy to drop down into Python for more sophisticated manipulations of the graph.

There is a paradoxical advantage to this DSL approach: it narrows options really quickly. It takes a very open-ended problem of language design and reduces many degrees of freedom to force a specific design as quickly as possible. This is useful because the goal of `tini.flow` is not to investigate language design but to implement enough of a language to be useful. Time spent designing the DSL is time stolen away from implementing workflow functionality.

`tini.flow` takes the description of a workflow in the form of a graph and executes it. This functionality is split into three separate pieces. Broadly, the DSL handles the encoding of the workflow and the construction of the corresponding graph. The execution engine handles the execution of the workflow and special snippets of code, called the prologue and epilogue, connect these components.

The `tini.flow` DSL merely translates a set of line-level statements into Python function calls. The `tini.flow` execution engine takes a graph, where the nodes are command lines and the edges are communication channels. It is the prologue and epilogue code that determines what these node and edge function calls do and that launches the execution engine with a data structure created with these calls.

The `tini.flow` proof of concept shows one version of this prologue and epilogue code. The prologue code is named this way, because it is code that appears at the top of a `tini.flow` script. The epilogue is code that appears at the bottom. The standard `tini.flow` prologue imports code from a similarly named module and the standard epilogue calls a run method on a structure representing the workflow.

This run method is responsible for launching the execution engine with the workflow graphs. Note, that the prologue and epilogue code can be specified as arguments to the DSL. Where the standard prologue and epilogue snippets or prologue module are insufficient, it is encouraged that users replace these with their own version.

This allows the execution engine and the DSL to have very specific and limited responsibilities. The prologue and epilogue adapt the DSL to the execution engine and provide an opportunity for the user to address design problems in `tini.flow`. In other words, since the DSL and execution engine do *one thing right*, the prologue provides space for handling functionality necessary for turning these simple tools into an actual workflow tool

Users are not expected to have to change the code for the DSL or the execution engine, because the prologue and epilogue provide a space where the user can freely change the code to add functionality or address design limitations.

The standard prologue and epilogue that ship with the `tini.flow` proof of concept implement only the functionality necessary to complete the demos in the `tini.flow` in action chapter. The prologue and epilogue are structured in the model of a plugin architecture, to allow users to add functionality without having to change core parts of `tini.flow`.

Currently, the standard `tini.flow` prologue and epilogue work as follows: The prologue defines classes for nodes, edges, workflows, and groups of workflows. The WorkflowGroup is a collection of Workflow classes that allows you to call add_node() and add_edge() functions on the most recently created element of the collection.

It provides a run function that then calls run() on each workflow in contains in order. Note, that with the standard prologue and epilogue a flow file that defines multiple workflows will not run those workflows simultaneously. Instead, it will run them sequentially. Also note, that a workflow graph does not have to be fully connected. In other words, a workflow graph can contain subgraphs that are executed independently and simultaneously.

The Workflow class represents a single workflow. Its add_node() and add_edge() methods create Node and Edge objects and store them. The standard Node class just stores node contents and metadata and provides no additional methods or functionality. The standard Edge class stores standard edge content and metadata and provides only a method for traversing this edge to turn the edge into a sequence of node pairs.

The standard Edge class is misnamed because it does not represent a single u,v edge but instead represents a path, whose elements can be nodes or other Edge-class paths. The traverse() function turns the Edge object into a sequence of concrete actual Node objects. In the Workflow class, the run() function takes every registered Node and Edge and constructs NetworkX graphs from them.

These graphs are the input to the `tini.flow` execution engine. The `tini.flow` execution engine requires at minimum a data graph that specifies through its edges how the various stdins and stdouts of the child processes are connected. The `tini.flow` execution engine also accepts any additional graphs representing other inter-process communication networks, such as edges that define the movement of control information.

The `tini.flow` prototype focuses on data flow and control flow as the two most important networks for communicating information between processes. The `tini.flow` design and future releases will make it clear that users can define any number of additional IPC networks. These networks might be useful for transmitting control, status, or reporting information through a `tini.flow` workflow and the user has complete control over how these networks can be structured.

For example, a workflow could describe an $n * n$ communication network for control information that would allow every processes to see status information for every other process and make intelligent error-handling, reporting, or optimization decisions, based off of this information. In the Workflow class's run() function, the Node and Edge objects are turned into nodes and edges on a NetworkX graph.

In order to do this, Node and Edge metadata must be processed. As a convenience to users, Node metadata can be inferred from Edge metadata and the run function performs this inference task.

Though a small part of the overall design of `tini.flow`, the design of the prologue and epilogue is critical to the design of the overall system.

As a thought experiment, imagine how a `tini.flow` workflow might be written by a sample user.

Typically, a user might sketch out small units interactively at their shell as simple shell pipelines. The user would verify that each unit correctly performs its task, for example that the various programs the users uses have the correct command line flag set, and any post-processing, such as grep and sed commands, work correctly.

The user may then cut and paste these command lines into a `tini.flow` script and turn them into named nodes. The DSL allows a node's content to be any command line string. Therefore, even a complex pipeline can be represented as a single logical `tini.flow` node and cut and paste directly from the shell into a `tini.flow` script.

Having defined the logical units in the workflow, the user may then turn their attention to how these logical units are interconnected from the perspective of control flow and data flow. This latter process may require a number of iterations, as the user may add additional debugging or error handling units to this as necessary.

These interconnections will be represented as `tini.flow` edges. For control flow edges, the triggering mechanism is modeled as wrapper scripts on the nodes themselves. However, requiring the user to specify trigger conditions for control flow edges in the node definitions may prove inconvenient, given the above style of development.

It is only in the latter phase, where the user investigates the interconnections between units that the user will consider control flow trigger conditions. The standard prologue module accommodates this style of working, by allowing the user to specify trigger conditions and trigger logic on edge definitions. The Workflow's run() method can then infer Node metadata from this Edge metadata, when processing these into a graph.

Chapter 4

`tini.flow` in Action

`tini.flow` was created with an eye on the elegance of simplicity.

This section is not merely a gallery of demos. It is intended to be a collection of carefully curated proofs that the assertions in this thesis about how to extend the shell to support rich data and workflows are in fact correct. These are proofs via working example. Because these examples work and because they do what they are intended to do, they prove the feasibility of this approach and corroborate the claims.

As `tini.flow` was built, these examples were used to drive development and design of features.

Similar to test-driven development, this is an Agile process. Each demo is a user story that is then developed against in successive iterations, continuously rewriting `tini.flow` to be as small and simple as possible.

These successive rewrites are necessary in order to obtain the desired functionality, revisiting design choices at every step to reducing complexity and optimize for greater generality.

4.1 `linear.flow` and `branch.flow`

`tini.flow` is not intended to replace all forms of shell scripting. It is intended to supplement uses of shell scripting where the program represents a complex non-linear data flow or where the program is in the form of a workflow. Following a long-standing tradition in programming manuals, we start with a *hello world!* demo.

`linear.flow` and `branch.flow` are two of the simplest programs that can be encoded with `tini.flow`.

They serve as a first proof-of-concept.

By itself `linear.flow` does not present a dramatic improvement over a simple shell script where the data pipeline is written out all at once.

It offers minor reusability benefits, which are already possible in the limited functionality provided by standard shell scripting.

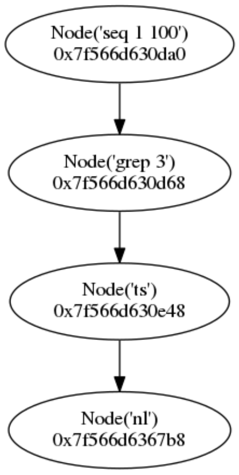


Figure 4.1. `linear.flow` in graph form

The source for `linear.flow` is presented below:

```
#!/usr/bin/env tf

** a = seq 1 70
* b = grep 3
* c = ts
* d = nl

% =  a | b | c | d
```

In this listing, the basic DSL format is visible, including node and edge definition syntax. Most simple `tini.flow` flow files will follow this structure of first defining nodes, then defining the edges.

Note that `linear.flow` and `branch.flow` do not include any control edges. They are data flow programs only. An equivalent shell program for `linear.flow` would be:

```
#!/bin/bash

seq 1 10 | grep 3 | ts | nl
```

(Note also that the `tini.flow` proof of concept does not currently address concerns such as `set -eo pipefail` which terminates the script on the first error and changes the scripts exit status to be the value of the rightmost command that exited with a non-zero status. This functionality can be easily included in later releases of `tini.flow`.)

The source for `branch.flow` is presented below:

```
#!/usr/bin/env tf

* a = seq 1 10
* b = grep 3
* c = grep 5
* d = ts
* e = nl
% = a | b | d
% = a | c | e
```

An equivalent shell script would be:

```
#!/bin/bash

seq 1 10 | pee 'grep 3 | ts' 'grep 5 | nl'
```

Even with just one branch, the `tini.flow` program is already en par or a slight improvement over the shell script. Additional branching would quickly become unmanageable in the shell, but would not present nested complexity for `tini.flow`.

The `tini.flow` DSL is a better match to the task of representing a graph than raw shell syntax.

4.2 `graph.flow` and `cycle.flow`

As the complexity of a data graph grows, the ability of a shell script to conveniently encode this graph becomes more and more limited. `graph.flow` is an example of a complex data flow graph encoded with `tini.flow`. Its source is presented below:

```
#!/usr/bin/env tf

* a = annotate a
* b = annotate b
* c = annotate c
* d = annotate d
* e = annotate e

* src1 = seq 0 1
* src2 = seq 2 3
* src3 = seq 4 5

% = src1 | a | b
% = src2 | a | b
% = src3 | a | b
% = b | c
% = b | d
% = b | e
```

This program corresponds to the following graph and implicitly includes extra processes for teeing data at each branch location.

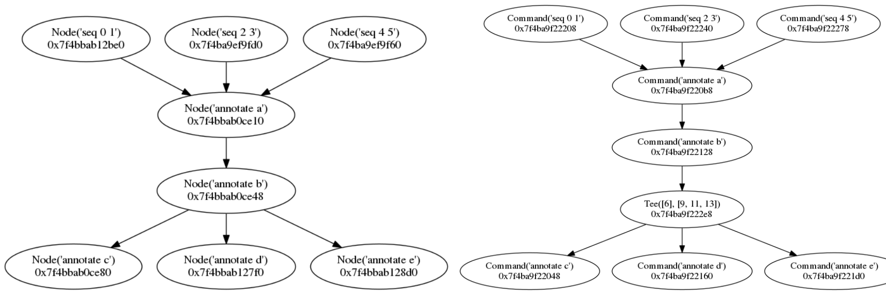


Figure 4.2. flow graph and process graph of `graph.flow`

(The annotate command is a simple Python script that reads lines in from stdin and writes them out to stdout, annotated with some fixed string. It is used in this example to help track the data flow in the program.)

This graph has only 8 nodes, but an equivalent shell script is too unwieldy to provide as a comparative example.

A tool like dgsh would encode this graph in the following fashion:

```
#!/usr/bin/env dgsh
```

```

(seq 0 1 & seq 2 3 & seq 4 5) |
annotate a |
annotate b |
tee |
{{
  annotate c

  annotate d

  annotate e
}} |
cat
```

For reference, the output of `graph.flow` is as follows:

```
$ ./graph.flow
2 a b c
3 a b c
0 a b c
1 a b c
4 a b c
5 a b c
2 a b e
3 a b e
0 a b e
1 a b e
4 a b e
5 a b e
2 a b d
3 a b d
0 a b d
1 a b d
4 a b d
5 a b d
```

`graph.flow` illustrates encoding complex DAGs in a pure data flow program. While we will not make a comparison regarding the ease of use or the advantages and disadvantages of the `tini.flow` vs. dgsh encoding, we will note, that dgsh is limited to acyclic graphs.

The listing for `cyclic.flow` is provided below. It encodes the following cyclic graph:

```
#!/usr/bin/env tf

* src = seq 0 0
* a = annotate a
* b = annotate b
* c = annotate c
```

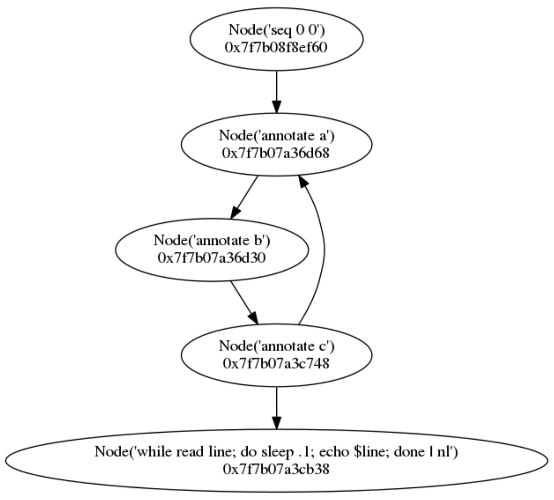


Figure 4.3. flow graph of `cyclic.flow`

```
* d = while read line; do sleep .1; echo $line; done | nl

% = src | a | b | c | a
% = c | d
```

There is no directly equivalent dgsh encoding.

This graph is non-terminating. Since none of the units use buffered I/O, cyclic.flow will stream output to the output as it runs.

Note, that by itself, a cyclic graph in a data flow program may be of limited use. However, as we introduce edges into our workflow graphs that carry control information, cyclic graphs will be important in their ability to represent non-terminating, long running programs.

4.3 `grandpa.flow`

`grandpa.flow` is an example of the DSL in action.

It shows the functionality of the DSL for describing arbitrary graphs as well as some advanced DSL features that have been included in the proof of concept.

In `grandpa.flow`, we follow the parody country song *I'm My Own Grandpa* with the intention to validate the genealogical assertions in its lyrics. The lyrics side-by-side with an interactive graph can be seen here: `http://gean.wwco.com/grandpa/`. However, the lyrics are also included in the flow file. It is probably best to watch the video on YouTube in order to be able to listen to the song with an included visual illustration of the graph: `https://youtu.be/4SnI963ZzgY`.

`grandpa.flow` starts with a block of helper functions that take two nodes and traverse the family trees to identify whether a set relationship exists between those two nodes. For example, the following function determines if one node is the grandfather of the other node.

```
def is_grandpa(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for gparent in parentage.predecessors(parent):
                for gparent in [gparent, *marriage.neighbors(gparent)
    ↪ ]:
                    if gparent == v:
                        return True
    return False
```

grandpa.flow does not use the standard prologue or epilogue, nor does it use the tini.flow execution engine. Instead, it defines its own set of __node__() and __edge__() functions that are used to build graphs for the custom purpose of encoding a family tree.

tini.flow flow files are full Python scripts and can contain inline Python functions, classes, and any other programmatic structures. If the family tree problem solved in grandpa.flow represented a common use case in our work, we could extract these helper functions to a Python module that we could import at the top of every relevant flow file.

grandpa.flow defines global state with the following __node__() and __edge__() ↪ functions, in order to construct two NetworkX graphs, representing marriage and parental relationships. These graphs are used as part of the helper function to determine relationships between nodes.

```
def __edge__(nodes, relationship, *args, **kwargs):
    if relationship == partner:
        u, v = nodes
        marriage.add_edge(u, v)
    elif relationship == child:
        u, v = nodes
        parentage.add_edge(u, v)

def __node__(description, gender, *args, **kwargs):
    marriage.add_node(description, gender=gender)
    parentage.add_node(description, gender=gender)
    return description
```

grandpa.flow is written by pasting the lyrics of the song as a comment into the file and defining relevant nodes and edges every time a new character or relationship is introduced.

```
# My father fell in love with her and soon they, too, were wed.
* dad = my father || male
```

```
% = dad | me || child
% = dad | redhead || partner
```

grandpa.flow shows that a flow file can serve as a data file, describing a graph structure, or even serve as a self-contained program operating on such a structure.

```
# This made my dad my son-in-law and changed my very life
assert is_soninlaw(me, dad)
```

```
# As husband of my grandmother, I am my own grandpaw.
if is_grandpa(me, me):
    print("OMG, I am actually my own grandpa!")
```

A valuable use-case for the functionality shown in grandpa.flow would be in describing a graph and including validation information inline in the document. This would allow for self-validating graph document files.

grandpa.flow could have been rewritten as a pure Python script and the data structure it processes as well as the helper functions it includes might even end up identical to the flow file.

Clearly, the DSL provides a nicely clearer and terser alternative to lines upon lines of NetworkX.DiGraph.add_node and NetworkX.DiGraph.add_edge calls.

The tini.flow version is in fact a Python script, but its advantage is not solely that it provides a slightly nicer DSL syntax. Conceptually, a flow file is not an arbitrary script that performs an arbitrary task. In the abstract, it is a piece of graph-data and some means by which to process that graph-data.

This difference restricts the contents of a flow file and this restriction provides structure that can be employed by additional tooling. For example, we could write a generic program that visualizes tini.flow graphs, knowing that the flow file contains a data structure that might have been produced dynamically but whose dynamic execution is conventionally a safe operation.

Static analysis tools in Python are severely limited by the language's dynamic nature. This dynamic nature can hide information from analysis tools such that the only guaranteed way to correctly process Python code is in its runtime context, i.e. by evaluating it.

Arbitrarily evaluating Python code is not possible, because not all code can be run at any time in any context, for example, it may rely on external state. And not all code is safe to run, i.e. it might affect external state. But dynamic behavior in a tini.flow flow file is restricted by convention to that which is needed to construct its graph data.

As a consequence, tini.flow flow files should be written so that they are safe to execute and capable of being executed at any time and in any context. Thus, a variety of dynamic analysis tools can be built to safely and robustly process a flow file, where they could not be guaranteed to be able to process the equivalent Python script.

4.4 make.flow

In the development of tini.flow, in order to identify bugs or other regressions, introduced by changes to core code. Originally, testing for tini.flow was done in a manual fashion by just calling the example scripts detailed in this section from the command line.

```
$ ./linear.flow
     1   J 19 06:20:10 3
     2   J 19 06:20:10 13
     3   J 19 06:20:10 23
     4   J 19 06:20:10 30
$ ./branch.flow
     1   5
J 19 06:20:18 3
```

The output for many of these test scripts is short enough that visual inspection was sufficient. As part of the development process, it is important to optimize the length of each iteration cycle, so that the time of trying something and testing it is as short as possible.

This is a case where the time between making a code change and running these scripts to see the result of that change is measured in seconds, typically totaling less than 1 minute.

If this 30 second to 1 minute iteration cycle could be reduced to 5 to 10 seconds, it would speed up development by allowing more iterations in the same period of time. In general, it would improve the development cycle to make testing *continuous*.

We could do that with the following snippet of code:

```
$ watch -n1 ./linear.flow
```

The above reruns the test every second, which means that we do not have to switch terminals or type anything to see the result of our work once we save a file that contains a code change. tini.flow was developed on a laptop and the use of watch, while convenient, had a marked effect on battery life and CPU temperature.

This lead to a more sophisticated version that ran on demand and addressed these issues:

```
$ while true; do clear; ./linear.flow; inotifywait -e create,modify
    ↪ tiniflow/*.py; done
```

This solution used inotify, an API for a file system monitoring mechanism in Linux, to trigger rerunning of the example upon detecting changes to code files. The result is a simple live testing environment. The above line can be run on a terminal on a separate screen to give feedback and shorten iteration loops.

The previous sequence of steps should be very familiar to any shell user.

Given some task, the user determines a way to solve the task by handwriting a command at the shell prompt. As the need for convenience or complexity grows, the user transforms direct hand-written, interactively input commands into commands that involve programmatic control flow that may no longer interactively input.

Before discussing how `tini.flow` fits into this style of working, let's consider the next step from here using two standard tools. The above command line, using `inotify`, is straightforward and clear, but is fairly difficult for the user to type from scratch every time that they start their work session.

Typically, a user will start by recalling the command from their shell history, perhaps using the readline default CTRL-r binding to perform history completion. One amazing facet of shell programming is that it acknowledges and embraces the inherent laziness of programmers.

The term laziness is not meant to be an insult. Instead, it conveys that programmers are strongly drawn to automation to eliminate redundant, manual, boring tasks. But they do so only after astute judgement as to whether this is actually saving time or not.

In the shell, there are many conveniences that allow the user to save time without having to pay the up-front cost of immediately having to reach for some programming language to build an automation tool. In the case of common repeated command line tasks, conveniences provided by the shell for manipulating the shell history is often enough to get minimum repeated effort with the minimum total expenditure of work.

However, the imprecision of shell history makes this technique relatively short-lived, as the history may contain many similar commands such that recalling them with CTRL-r or other conveniences is more time-consuming than rewriting them from scratch. It is at this point that a pragmatic programmer may reach for an automation tool in the form of a script of some sort.

In the development of `tini.flow`, we could improve upon the hand-written command line by moving that logic into a simple shell script that might look like the following:

```bash
#!/bin/bash

target=$1
while true; do
    clear
    ./$target
    inotifywait -e create,modify tiniflow/*.py
done
```

The above case could be called `watch.sh` and be stored in the root directory of the project. It could be invoked by calling `$ ./watch.sh linear.flow`. Note that in addition to being much shorter to write than the previous iteration, the `watch.sh` script is parameterized to allow the user to watch any example flow file.

Under some circumstances, the user may want to continuously watch the execution of an example flow file or may just want to execute the example one time. To facilitate both of these tasks with a minimum of repetition, this shell script could be expanded. The alternative approach of writing two shell scripts, one for each task, might introduce duplication.

The combined shell script might look like the following:

```
#!/bin/bash

target=${1:-testall}
case $target in
    watch)
      target=$2
      while true; do
          clear
          ./$target
          inotifywait -e create,modify tiniflow/*.py
      done
      ;;
    test)
      target=$2
      ./$target
      ;;
    testall)
      for file in *.flow; do
          $0 test $file
      end
      ;;
esac
```

This shell script is an improvement on the previous. It supports multiple *targets* for single testing or continuous testing of files. It includes a special meta-target, testall, which automatically triggers testing of every file.

It even uses bash's default variable syntax to give a default action to run. However, from a strictly functional perspective – even ignoring complaints about the baroqueness of shell syntax – this script has notable limitations.

(Before discussing these, note, for the purposes of this investigation, that it is irrelevant whether the shell script is a shell script or a script in some other language like Python. Definitely, the syntax and features of Python are richer and friendlier to use than those of the shell, but in both cases, the script is encoding the direct execution of some task and we'll soon see that we need to model the task itself as a first-class higher-order structure, in order to take advantage of certain properties.)

One such limitation is that it does not support tab-completion. Tab-completion is an affordance that users have come to rely upon. It is not a feature of a program or the shell paradigm. It is a feature of the shell environment, namely bash, zsh, ksh, et cetera. To implement tab-completion for a given executable in a given shell environment requires writing configuration code against that shell environment's completion API.

For ad-hoc shell scripts, it is possible to write completion logic for them on a script-by-script basis. However, this is tedious and the completion logic determines what arguments to provide, based primarily on the name of the executable.

Therefore, two scripts both named make.sh would use the same completion logic. Given this limitation, the natural next step would be to give the script some structured way to report its completion *words*. This could be accomplished through various means, such as accompanying an every `make.sh` script with a `.make.completion` script that encodes the completion logic or writing every make.sh script in a very strictly patterned style so that the completion logic could be determined via parsing of the `make.sh` file or rewriting the `make.sh` with some higher language functionality to expose its completions.

All of these approaches are either difficult to implement or difficult to unify around. A very likely response to this problem is to replace the use of a shell script with the use of a Makefile.

It may seem like rewriting the shell script, solely because it does not provide tab-completion, is a sign of how spoilt millennial programmers these days are. However, not having tab-completion is actually a symptom of a much broader class of problems.

Fundamentally, most shell scripts are direct encodings of automation steps. There is no higher-order, first-class, or meta-encoding of the purpose, structure, or behaviors of the shell script.

They lack the essential feature of a workflow: a first class (graph) encoding of the units of composition.

The lack of this consistent higher-order encoding robs us of the ability to automatically generate conveniences like tab-completion, dependency detection and management, systematic error-handling, or even automatic generation of documentation.

Each of these can be added to a shell script, but it must be added to every shell script by hand every time or the shell scripts have to be completely rewritten that no longer fits this iterative refinement and then cut-and-paste from the command prompt procedure.

The Make program is most popularly used as a build tool, usually in the case of building software written in C and sometimes C++. However, Make provides a regularized structuring of cut-and-pastable shell snippets such that we can process these structures to automatically generate conveniences like help-text, tab-completion, smart dependency management, and even rudimentary automatic error-handling, i.e. stop-on-first-error.

A Makefile version of `make.sh` might look like:

```
.SHELL := /bin/zsh

.PHONY: testall test watch

testall:
  for flow in *; do FLOW=$$flow make -s test; done

test:
  @[[ ! -e $$FLOW ]] && echo 'must specify $$FLOW' && exit
  ./$(FLOW)

.ONESHELL:
watch:
  @[[ ! -e $$FLOW ]] && echo 'must specify $$FLOW' && exit
  @while true; do
    clear
    ./$(FLOW)
    inotifywait -e create,modify tiniflow/*.py
  done
```

This Makefile shows some distinct improvements over our shell script. The Makefile contains the same set of targets, including the default target and the batch target. The rules for these targets are simple shell snippets and can be written in our preferred iterative interactive style.

The rigid structure of the Makefile means that we get tab-completion of all targets for free. Strictly, we can think of a Makefile as a graph of dependencies and allowing the user to evaluate or to efficiently execute shell snippets, upon detecting changes on this graph. This Makefile does not encode dependencies between targets but it can still be viewed as a graph consisting of three disconnected subgraphs, one for each target.

We can consider each subgraph a linear sequence of rules, one for each line of the rule, a control sequence of nodes, with control flow moving from one node to another upon success and terminating immediately upon failure.

However, new problems immediately appear. Previous complaints regarding the limitations of the shell are even more exaggerated in the case of a Makefile. The syntax is bizarre, programming functionality is limited and text/macro-based, is built around even flimsier assumptions, such as assuming that entities should be space-delimited by default, and this often without ways to work around these limitations.

Additionally, the Makefile is too rigid and too static. Many of our tools for analyzing this Makefile analyze it via static parsing. As a consequence, we cannot automatically discover the names of our targets from within the Makefile itself and are likely to either manage these by hand or with some supplementary tool, like a Makefile-generating script or heavy-weight tools like m4, autotools, or autoconf.

Note, Michael Klement [2014] claims on Stack Overflow that the following line-noise will list all of the targets in a Makefile, which would allow us to use programmatic functionality within Make to discover targets, which we would then need to feed to our completion engine.

```
list:
    @$(MAKE) -pRrq -f $(lastword $(MAKEFILE_LIST)) : 2>/dev/null |
    ↪ awk -v RS= -F: '/^# File/,/^# Finished Make data base/ {if (
    ↪ $$1 !~ "^[#.]") {print $$1}}' | sort | egrep -v -e '^[^[:
    ↪ alnum:]]' -e '^$@$$' | xargs
```

Where the shell script allows us to process command line arguments, allowing us to parameterize our tasks, the Makefile only allows us to parameterize by looking at environmental variables, which proves clumsier.

The arguments of the Makefile are the targets to build, typically representing filenames to create, which limits our ability to provide proper namespacing of actions. Commonly, the solution to this is to *fake* namespaces by hard-coding words like `test-` or `watch-` at the beginning of the target name. This would allow for targets like `test-linear.flow` and `watch-linear.flow` which would provide convenient tab completion for users.

The shell script allows us to specify arbitrary trigger conditions for actions, even though it does not expose the triggers and actions as a first class entity like a graph. The Makefile, on the other hand, only allows us to check for existence of a file, check the timestamp on a file, or run the rule always in the case of a phony rule.

Furthermore, in the body of a rule, the individual lines of the shell snippet give only rudimentary error handling by default, stopping on failure, but not providing a means for specifying alternate error-handling paths.

But fundamentally, unlike the shell script, Make provides a higher-order structuring that we need to build conveniences on top of the raw automation snippets. The Makefile does separate the tasks and a graph of their dependencies from the actual execution of these tasks where the shell script does not.

What is missing is a more script-like interface for the construction of the graph. This leads us directly to `tini.flow`. A `tini.flow` flow file is a Python script, written with the help of a convenient DSL, for describing a graph that can be fed into an execution engine to execute the nodes of the graph.

A `tini.flow` version of this Makefile would look like the following:

```
#!/usr/bin/env tf

from os import listdir

flows = [filename for filename in listdir('.')
                   if filename.endswith('.flow')]

> test/all
    * tail = true -- on.start
    for flow in flows:
        * run   := clear && ./{flow} -- on.success
        % tail   = tail - run

for flow in flows:
    > test/{flow}
```

```
        * run   := clear && ./{flow}

    > watch/{flow}
        * wait  := inotifywait -e create,modify tiniflow/*.py
        * run   := clear && ./{flow}
        * debug := python -m tiniflow.dsl ./{flow}

        % = wait - run - wait   -- on.always, on.start
        % =        run - debug  -- on.failure

if __name__ == '__main__':
    from sys import argv
    if not argv[1:]:
  print('\n'.join(__workflow__.workflows))
    for arg in argv[1:]:
      __workflow__[arg].run()
```

The above flow file dynamically constructs $1 + 2N$ separate workflow graphs for N flow files that it finds in the current directory. The general look of each of these graphs matches that of the Makefile and shell script, in which shell snippets that we were able to iteratively and interactively discover at the command line have been pasted in.

The flow file separates the definition of the executable shell snippets, i.e. nodes, from their connectivity and their triggering conditions, i.e. the edges. This gives us space to customize the control flow, such as in the case of the watch target. If the execution of the watched flow file fails, we follow a special branch that emits debugging information. In this case, we simply emit the transpiled Python code to help identify DSL errors, causing the script to fail.

A simple branching like this would be trivial to add to the shell script. After all, the shell script encodes the direct execution of some task and makes the inclusion of additional control flow very easy. This simple branch could even be added to the Makefile, but more complex branching or more complex debug operations will quickly overwhelm the Makefile, making it hard and maintain.

The program structuring we were missing in the shell script, would have still been missing if we had just translated it line-by-line to Python. We needed a first-class or higher order representation of the graph. We could have rewritten the shell script as a Python script, to avoid the pitfalls of shell syntax and then rewritten the Python script to add in this structuring.

If we followed this rewriting process enough times, we might yearn for some more convenient syntax, we might consider writing a DSL, and, thus, we might develop a tool much like `tini.flow`.

Chapter 5

Conclusions

5.1 Summary of Results

Through the examples in the previous chapter, we can see the beginnings of a useful workflow tool. Fundamentally, a workflow tool is about the higher-order modeling of components in a system. This modelling allows us to automatically answer and solve specific problems related to the program when viewed as a composition of subtasks, accomplishing some larger human directed task.

The most important characteristic of a workflow systems is that it allows for composition of units, retaining this higher-order information. Clearly, this composition is a graph-composition, and there is a natural mapping of graph nodes to tasks and graph edges to connections between those tasks.

The choice of computational mechanism for the nodes, whether processes, functions, services, or something else, is largely a secondary concern. However, choosing Unix processes as the most granular unit in a workflow system has clear advantages, stemming from the history, longevity, and popularity of the shell.

This long history has left the shell with serious baggage that cannot be ignored because loosing backwards-compatibility also throws away huge quantities of work that we do not want to duplicate.

The limitations of the shell are not all fundamental to the composition of Unix processes and, through a custom execution environment, many of these limitations can be addressed and dealt with. This leaves us with an approach for implementing a brand new workflow system that can be viewed as an extension of classical shell programming.

Such a workflow system necessarily must include some way to represent workflow graphs, some way to map those graphs to executable processes and mechanisms for connectivity and must include a way to execute the graph.

The core design of `tini.flow` revolves around these three pieces. `tini.flow` introduces a DSL for modeling graphs in a syntax that is reasonably terse, easy to implement, and easy to extend. The DSL is embedded into Python, which leads to these advantages.

Two small, but critical, pieces of shim code connect the DSL to the execution engine. The first is denoted as the prologue and epilogue. These are snippets of code that surround the workflow definition to map node and edge definition syntaxes to graph operations. The second is a user-extensible library of wrapper scripts that wrap process nodes, in order to manage triggering and control flow information.

The last piece is an execution engine, which, by virtue of the design of the above components, has a simple and general implementation. The result is a new workflow tool that fits nicely into the common and popular paradigm of shell scripting.

`tini.flow` is probably not the right tool if you have lots of data munging to do. As long as you can describe the tasks on a napkin, `tini.flow` might be the right tool for you to use.

In other words, `tini.flow` is the right tool for workflows of the form:

If this, then that, transform some data through filter, update some database, do some data analysis, and send some individualized reports ('you missed your deadline'/ prognoses for future developments of stocks in portfolio, etc.). If the generation of these tasks is itself dynamic, `tini.flow` may be the right tool – e.g. generate some custom scripts for every addition to a system or for new person joining the business.

Just as Python is used as an orchestration and glue language for connecting numerical sub-routines written in C, C++, or Fortran, `tini.flow` could be considered as an overarching orchestration tool, where the unit of computation may be a standalone process. `tini.flow`'s reliance on the shell is derived from the shell's reputation as a language highly optimized for data manipulation.

Just as `tini.flow` extends shell scripting, the design of `tini.flow` immediately leads to extensions of each of its pieces of core architecture. These aspects of `tini.flow` could each be extended, in order to add functionality common to other workflow systems, to bring `tini.flow` up to parity with existing production alternatives.

Additionally, extensions to `tini.flow` could enable novel functionality not commonly seen in existing systems but of potential value.

A sampling of future avenues of research is discussed below.

5.2 Benchmarks

`tini.flow` is not an HPC computing or scientific workflow tool. Its strength lays in it's ability to structure workflows in a human-understandable way, in order to allow for easy and correct implementation of such human-focused workflows. Long shell scripts can become very cumbersome. This is where `tini.flow` shines.

However, it is important to establish a baseline and see if all the extra functionality comes at a significant cost. It turns out that it does not and that could have been expected as `tini.flow` uses all the same system calls as the standard piping mechanism. All benchmarks were executed at a fixed clock speed of 2GHz. OS-functionality will, however, cause some unavoidable noise in the measurements through the way it interleaves and schedules the processes.

One consideration to take into account is that the piping mechanism leads to parallelism in the execution, as piping essentially gives you concurrency for free. This means that processes can be scheduled on multiple cores of the CPU by the OS.

The below flow file describes some simple but meaningful benchmarks to establish this argument through the measuring of throughput. `raw throughput` gives an upper bound on how fast the pipeline can be, by basically only involving the CPU. `raw sequence` shows how fast the seq command is, to give an understanding how fast an actual straight-forward command line application is and to give a starting point for the next example. `pipeline` shows how fast a very simple sample shell pipeline can be in `tini.flow`.

The exact benchmark code listings can be found in the appendix.

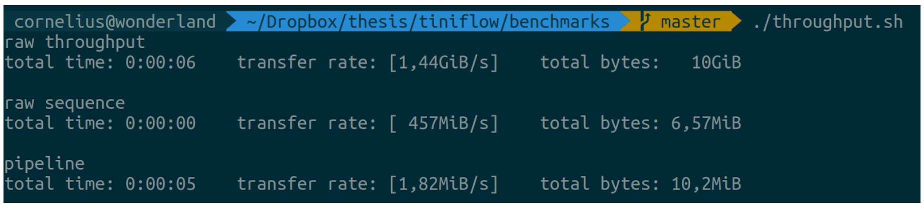
```
> raw throughput
    * a  = dd if=/dev/zero bs=1G count=10 2>/dev/null
    * b := pv -F '{pv_format}' > /dev/null
    % = a | b

> raw sequence
    * a  = seq 1 $((1e6))
    * b := pv -F '{pv_format}' > /dev/null
    % = a | b

> pipeline
    * a  = seq 1 $((1e6))
    * b  = grep 3
    * c  = ts
    * d := pv -F '{pv_format}' > /dev/null
    % = a | b | c | d
```

Figure 5.1. flow file with benchmarks for throughput

The execution of the flow file with the benchmarks as well as the equivalent shell script shows that measurements are not meaningfully different. As mentioned above, there is some noise in the management caused by the nature of how processes are scheduled by the OS.

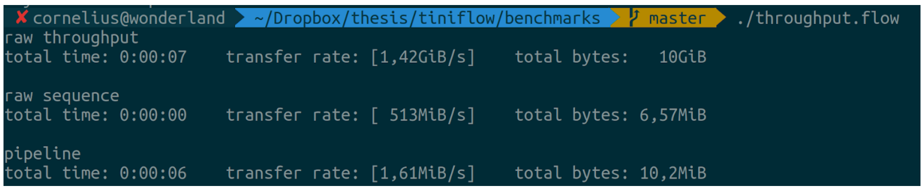
After having established the baselines for throughput in the previous examples, we now look at an exemplary shell workflow: the McIlroy example that was mentioned earlier in this thesis. This benchmark clearly shows the difference between the flow or pure shell executions of this particular pipeline.

There is a meaningful difference, but mainly in overhead. Further profiling led to the clear conclusion that about 90%, namely 0.7s, of the overhead were introduced by the import of the NetworkX graph library. About 10% of the overhead can be attributed to `tini.flow`'s construction of the graph.



```
 cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  ./throughput.sh
raw throughput
total time: 0:00:06    transfer rate: [1,44GiB/s]    total bytes:   10GiB

raw sequence
total time: 0:00:00    transfer rate: [ 457MiB/s]    total bytes: 6,57MiB

pipeline
total time: 0:00:05    transfer rate: [1,82MiB/s]    total bytes: 10,2MiB
```

Figure 5.2. throughput measure for pure sh



```
✗ cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  ./throughput.flow
raw throughput
total time: 0:00:07    transfer rate: [1,42GiB/s]    total bytes:   10GiB

raw sequence
total time: 0:00:00    transfer rate: [ 513MiB/s]    total bytes: 6,57MiB

pipeline
total time: 0:00:06    transfer rate: [1,61MiB/s]    total bytes: 10,2MiB
```

Figure 5.3. throughput for flow file with same functionality

From a theoretical standpoint,

1. `tini.flow` prevents whole program optimizations (because units of composition are opaque)
2. `tini.flow` allows the OS to schedule on multiple cores

If the units are shell processes, then there will be overhead in launching these processes and in managing the processes, for example context switches.

Context switches are necessary because the CPU has to treat OS processes as opaque. On the other hand, the CPU can distribute OS processes to multiple cores because it knows they are opaque.

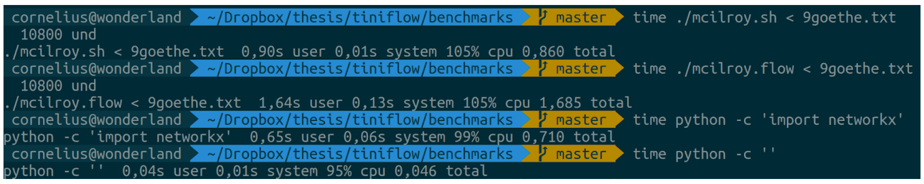
In a closed system, like one where the units are functions, you can optimize across these units to eliminate overhead and startup penalties.

Groups of units in `tini.flow` can be reformulated as a single unit on demand to address the above problem. Nevertheless for typical stream processing examples, as in not matrix multiplication or deep learning, the shell is competitive with other approaches. See the Hadoop vs Laptop talk mentioned in the thesis.

Because the `tini.flow` runtime merely orchestrates a more sophisticated wiring up of shell commands, but does essentially the same thing, `tini.flow` is going to be on par with the speed of a shell pipeline.

`tini.flow` will have additional overhead related to the construction of the graph, which could be dynamic, and in practice `tini.flow` will have noticeable overhead, on the order of $\frac{1}{4}$ second, because it relies on NetworkX and importing NetworkX is slow. Obviously, switching to a custom library instead of NetworkX will eliminate this penalty.

Fundamentally, `tini.flow` units should represent human-describable units and should not be so granular that process start time would be significant.



```
 cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  time ./mcilroy.sh < 9goethe.txt
  10800 und
./mcilroy.sh < 9goethe.txt  0,90s user 0,01s system 105% cpu 0,860 total
 cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  time ./mcilroy.flow < 9goethe.txt
  10800 und
./mcilroy.flow < 9goethe.txt  1,64s user 0,13s system 105% cpu 1,685 total
 cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  time python -c 'import networkx'
python -c 'import networkx'  0,65s user 0,06s system 99% cpu 0,710 total
 cornelius@wonderland  ~/Dropbox/thesis/tiniflow/benchmarks  master  time python -c ''
python -c ''  0,04s user 0,01s system 95% cpu 0,046 total
```

Figure 5.4. benchmarks of the McIlroy example implemented in sh and flow

5.3 Future Research

The `tini.flow` proof of concept includes just enough functionality to complete the examples discussed in this thesis. Given `tini.flow`'s design, extending the implementation is possible. Below are some additional avenues of research to extend the `tini.flow` model and its design.

5.3.1 `inline.flow`

Currently, `tini.flow` nodes are shell expressions: either the names of executable programs or simple inline shell syntax. The execution engine just spawns `$SHELL` and passes the contents of the node to the -c flag. Nodes could be wrapped with additional layers such that they could contain other programming language snippets, such as Ruby, Awk, or Sed snippets.

The execution engine could also be extended such that it does not spawn a process, if the node is marked as an *in-process* node. An in-process node could contain a snippet of Python code that would be evaluated within the execution engine or within a single spawned Python child process.

If this spawned child process looked like a Jupyter Notebook kernel, it could receive multiple snippets to run and these snippets could share state. This could allow a `tini.flow` flow to have nodes that are executable programs, nodes that are inline shell snippets, nodes that are inline Ruby, Awk, or Sed programs, and nodes that are inline Python code.

In the last case, the Python code could be required to be a Python generator. This Python generator could take one argument per type of input edge and yield a structure that multiplexes all of the data for its output edges. The requirements for the communication mechanism between nodes is that they should be read-once FIFOs that can transmit binary data.

A Python generator fits this very nicely. A sophisticated example of this could seamlessly mix data passed between Python generators, in a high-speed in-process fashion, with data passed via traditional stdin, stdout mechanisms.

5.3.2 `fifo.flow`

Just as a Python generator provides a sufficient interface between nodes to replace the passing of data on anonymous pipes, so do many other mechanisms. The execution engine only requires that edges become binary FIFOs whose data can only be read one time.

A simple extension to `tini.flow` would be to replace anonymous pipes with named FIFOs, using the mkfifo command. This may have few advantages over the current approach.

It may be difficult to replace anonymous pipes with named files, because `tini.flow` would have to coordinate reads and writes, to ensure that lines are not skipped or re-read. However, the `tini.flow` execution engine could be further generalized to insert additional nodes at every edge. These nodes would communicate over anonymous pipes at one end and at the other end, between inserted nodes, would be free to communicate in any fashion they desire.

With this approach, `tini.flow` could model edges with adapted netcat nodes, so that node data is transmitted via TCP or UDP sockets across a network. `tini.flow` could model edges with processes that transmit data via message broker systems, like Kafka. `tini.flow` could model edges with processes that record data to a database or other permanent store.

5.3.3 `distributed.flow`

Because `tini.flow` is based on Unix-pipelines, it follows a blocking/buffered pipeline execution semantic and it supports simple parallelism as do most data flow systems.

`tini.flow`'s primary goal is not parallel execution or to focus on the performance of computations, but rather improve the semantics with which to compose processes in human-driven workflows. These workflows manipulate tasks which have some human-determined semantics and are modeled as nodes.

`tini.flow` does not model graphs purely to perform graph-style mathematical computations. It is not a tool like Dask, which focuses on numerical and scientific computing problems.

Once `tini.flow` edges can be generalized beyond anonymous pipes, `tini.flow` workflows can become distributed programs. Currently, the execution engine spawns every process on the same computer. However, minor changes could allow the execution engine to ssh into specific machines and only execute the processes there.

These distributed `tini.flow` nodes could communicate via TCP or ssh or any other network communication standard. Thus, a large `tini.flow` workflow could be turned into a coarsely-grained distributed program.

In such a program, the units of computation would still be Unix processes. One limitation of Unix processes is that it is expensive to copy data without shared memory.

Additionally, there is a cost to context-switching between processes when performing a computation. These costs could increase with the distributed program of this form. Ultimately, `tini.flow` is probably unsuited for non-commodity computing environments such as supercomputers with custom interconnects for super fast data sharing.

This kind of distributed computing would be useful in cases where the distribution of work has some non-performance-driven requirement, such as a semantic requirement of some sort.

5.3.4 Custom Codec

In the proof of concept, flow files are transpiled to Python by invoking the DSL directly via a helper shell script. This has a disadvantage of obscuring certain details, like the original file name, in Python error-reporting.

Python supports custom codecs that can be installed alongside software libraries. These codecs are usually used for decoding text-data in a non-standard text-encoding.

However, the code for implementing a codec requires just a function that decodes a byte-string into an encoded Python-string. `tini.flow` could package the DSL as a custom Python codec to avoid the need for a helper-script to launch flow files.

Flow files could be run directly from a standard Python interpreter, with the codec handling transpilation. Additionally, flow files would become importable Python modules and users could employ this familiar form of complexity management.

Users could also create libraries of common nodes for common tasks and distribute them via the Python package index. These libraries could be imported and used in flow files, just as if they were any other Python module.

5.3.5 Additional Affordances: Plumbing vs. Porcelain

In practice, affordances are grown or evolved, rather than designed up front. There is an old story about a university campus designed by a master architect with beautiful walkways surrounding carefully manicured lawns, connecting building to building. Upon visiting the campus years later, the architect immediately realized the folly of his design when seeing the trampled grass cutting between the buildings in the most efficient path.

In the design of `tini.flow`, focus was placed on what affordances were necessary to implement core parts that were needed in order to create the example workflows. As a proof of concept, `tini.flow` is undeniably missing the fit-and-finish of an more mature, production-ready tool.

This is by design. `tini.flow` could have been further polished when created, only for us to realize that this polish would immediately rust in the hands of actual users.

Instead, `tini.flow` does its best to make the tool fit the user's hand with affordances like the DSL. It understands that as the tool is used the rough spots will be naturally worn away and polished in subsequent releases.

Critically then, we must ensure that it is possible to polish the interface in subsequent releases without the risk of needing an extensive redesign of `tini.flow`'s internals. The distinction between the interface and its internals has been likened with plumbing vs. porcelain [Perez De Rosso and Jackson, 2013]. The focus in the proof of concept for `tini.flow` is solid plumbing, upon which a smooth porcelain can iteratively be developed.

The plumbing vs. porcelain distinction is often brought up when discussing Git. Git's user interface is controversial, with many proponents and detractors. The risk in the approach we take with `tini.flow` is that in the face of good plumbing, but poor porcelain, it is easy to fall into a local maximum and find it difficult to ever climb out. This case can be worsened if if the tool sees explosive popularity like Git, and users quickly become set in their ways.

Perhaps, the Haskell developers are right and `tini.flow` should *avoid success at all costs* [Biancuzzi et al., 2009].

Bibliography

Adam Barker and Jano Van Hemert. Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics*, pages 746–753. Springer, 2007.

Federico Biancuzzi et al. *Masterminds of programming: Conversations with the creators of major programming languages*. " O'Reilly Media, Inc.", 2009.

Vicki Boykis. Replacing hadoop with your laptop: The case for multiprocessing. https://veekaybee.github.io/data-lake-talk/#/, 2017. Accessed: 21.01.2018.

Steve Dower. Pep 551 – security transparency in the python runtime. https://www.python.org/dev/peps/pep-0551/, 2017. Accessed: 21.01.2018.

Dr. Drang. More shell, less egg. http://www.leancrew.com/all-this/2011/12/more-shell-less-egg/, 2011. Accessed: 20.01.2018.

Richard Gabriel. The rise of "worse is better". https://www.jwz.org/doc/worse-is-better.html. Accessed: 21.01.2018.

S Garfinkel, D Weise, and S Strassmann. *The UNIX Haters Handbook: The Best of UNIX-Haters On-line Mailing Reveals Why UNIX Must Die*. IDG Books Worldwide, Inc., June, 1994.

James J Gibson. *The ecological approach to visual perception: classic edition*. Psychology Press, 2014.

David Hollingsworth and UK Hampshire. Workflow management coalition: The workflow reference model. *Document Number TC00-1003*, 19, 1995.

Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

M. Tim Jones. Evolution of shells in linux: From bourne to bash and beyond. https://www.ibm.com/developerworks/linux/library/l-linux-shells/index.html, 2011. Accessed: 21.01.2018.

Mark Kampe. Guidelines for command line interface design. http://www.cs.pomona.edu/classes/cs181f/supp/cli.html, 2012. Accessed: 18.01.2018.

Brian W Kernighan. The unix system and software reusability. *IEEE Transactions on Software Engineering*, (5):513–518, 1984.

Robert E Kraut, Stephen J Hanson, and James M Farber. Command use and interface design. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 120–124. ACM, 1983.

Bruce J MacLennan. *Principles of programming languages: design, evaluation, and implementation*. Oxford University Press, Inc., 1999.

M. D. McIlroy. A research UNIX reader: Annotated excerpts from the programmer's manual, 1971-1986. Technical Report CSTR 139, AT&T Bell Laboratories, 1987.

Daniel Goldfarb Michael Klement. How do you get the list of targets in a makefile? https://stackoverflow.com/questions/4219255/how-do-you-get-the-list-of-targets-in-a-makefile, 2014. Accessed: 21.01.2018.

Don Norman. *The design of everyday things: Revised and expanded edition*. Basic Books (AZ), 2013.

Cesare Pautasso and Gustavo Alonso. Jopera: a toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce*, 9(2):107–141, 2005.

Santiago Perez De Rosso and Daniel Jackson. What's wrong with git?: a conceptual design analysis. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 37–52. ACM, 2013.

Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

Ben R Rich. clarence leonard (kelly) johnson. *National Academy of Sciences Biographical Memoirs*, 1995.

Peter H Salus. *A quarter century of UNIX*. Addison-Wesley Reading, 1994.

Ashvini Sharma. Python as an excel scripting language. https://excel.uservoice.com/forums/304921-excel-for-windows-desktop-application/suggestions/10549005-python-as-an-excel-scripting-language, 2017. Accessed: 21.01.2018.

Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 2017.

Joel Spolsky. The law of leaky abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/, 2002. Accessed: 20.01.2018.

Jonathan Sprinkle, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis. Guest editors' introduction: What kinds of nails need a domain-specific hammer? *IEEE software*, 26(4), 2009.

Neal Stephenson. *In the beginning... was the command line*. Avon books New York, 1999.

Victor Stinner. Pep 446 – make newly created file descriptors non-inheritable. https://www.python.org/dev/peps/pep-0446/, 2013. Accessed: 21.01.2018.

Edward R Tufte, Susan R McKay, Wolfgang Christian, and James R Matey. Visual explanations: images and quantities, evidence and narrative. *Computers in Physics*, 12(2):146–148, 1998.

Larry Wall. Amazon.com interview: Larry wall. https://www.amazon.com/gp/feature.html?ie=UTF8&docId=7137. Accessed: 21.01.2018.

David A. Wheeler. Filenames and pathnames in shell: How to do it correctly. https://www.dwheeler.com/essays/filenames-in-shell.html, 2016. Accessed: 20.01.2018.

Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.

Fred Wilson. The return of the command line interface. http://avc.com/2015/09/the-return-of-the-command-line-interface/, 2015. Accessed: 20.01.2018.

Appendix A

Code Listings

This appendix contains a listing of the `tini.flow` proof of concept. `tini.flow` is a work in progress. The most up-to-date version of `tini.flow` can be found at

https://github.com/TitusCornelius/tiniflow

A.1 dsl.py

```python
#!/usr/bin/env python3

from re import compile, escape, DOTALL, UNICODE
from collections import namedtuple

Pattern = namedtuple('Pattern', 'block node edge workflow tempchange
    ↪ permchange nodesep edgedata nodedata workflowdata')

def generate_patterns(nodesep='[|-]'):
    node_sigil         = escape('*')
    edge_sigil         = escape('%')
    nodesep_sigil      = nodesep
    workflow_sigil     = escape('>')
    tempchange_sigil   = escape('$')
    permchange_sigil   = tempchange_sigil * 2
    edgedata_sigil     = nodesep_sigil * 2
    nodedata_sigil     = nodesep_sigil * 2
    workflowdata_sigil = nodesep_sigil * 2

    block  = r'([^\n]+)(\n)'
    indent = r'[ \t]*'
    name   = r'\w+'
    assign = escape(':') + r'?' + escape('=')
    space  = r'\s*'
    expr   = r'.+'

    _name   = f'(?P<name>{name})?'
    _assign = f'(?P<assign>{assign})'
    _expr   = f'(?P<expr>{expr})'
    _indent = f'(?P<indent>{indent})'

    node         = _indent + space.join([node_sigil, _name, _assign, _expr])
    edge         = _indent + space.join([edge_sigil, _name, _assign, _expr])
    workflow     = _indent + space.join([workflow_sigil, _expr])
    tempchange   = _indent + space.join([tempchange_sigil, _name, _assign, _expr])
    permchange   = _indent + space.join([permchange_sigil, _name, _assign, _expr])
    nodesep      = space + nodesep_sigil      + space
    edgedata     = space + edgedata_sigil     + space
    nodedata     = space + nodedata_sigil     + space
    workflowdata = space + workflowdata_sigil + space

    flags = DOTALL | UNICODE
    return Pattern(
        block        = compile(block,        flags=flags),
        node         = compile(node,         flags=flags),
        edge         = compile(edge,         flags=flags),
        workflow     = compile(workflow,     flags=flags),
        tempchange   = compile(tempchange,   flags=flags),
        permchange   = compile(permchange,   flags=flags),
        nodesep      = compile(nodesep,      flags=flags),
        edgedata     = compile(edgedata,     flags=flags),
        nodedata     = compile(nodedata,     flags=flags),
        workflowdata = compile(workflowdata, flags=flags),
    )

def parse(text):
    temp_changes = {}
    perm_changes = {}
    p = generate_patterns(**{**perm_changes, **temp_changes})
    for block in p.block.split(text):
        if block.strip():
            temp_changes.clear()
        mo = p.node.fullmatch(block)
        if mo:
            indent = mo.group('indent')
            name   = mo.group('name')
            expr   = mo.group('expr')
```

```python
            assign = mo.group('assign')
            args   = p.nodedata.split(expr)
            f      = 'f' if assign == ':=' else ''
            arg    = args[0].replace('\\','\\\\').replace('"', r'\"')
            args   = ', '.join([f'{f}"{arg}"', *args[1:]])
            yield f'{indent}{name} = __node__({args})'
            continue

        mo = p.workflow.fullmatch(block)
        if mo:
            indent = mo.group('indent')
            expr   = mo.group('expr')
            args   = p.workflowdata.split(expr)
            args   = ', '.join([f'f{args[0]!r}', *args[1:]])
            yield f'{indent}with __workflow__({args}):'
            continue

        mo = p.edge.fullmatch(block)
        if mo:
            indent = mo.group('indent')
            name   = mo.group('name')
            expr   = mo.group('expr')
            args   = p.edgedata.split(expr)
            seps   = [p.nodesep.findall(arg) for arg in args]
            args   = [p.nodesep.split(arg) for arg in args]
            args   = ', '.join(f'({", ".join(arg)})' for arg in args)
            yield f'{indent}{name or "_"} = __edge__({args}, seps={seps!r})'
            continue

        mo = p.tempchange.fullmatch(block)
        if mo:
            indent = mo.group('indent')
            name   = mo.group('name')
            expr   = mo.group('expr')
            temp_changes[name] = expr
            yield f'{indent}#$ {name} = {expr}'
            p = generate_patterns(**{**perm_changes, **temp_changes})
            continue

        mo = p.permchange.fullmatch(block)
        if mo:
            indent = mo.group('indent')
            name   = mo.group('name')
            expr   = mo.group('expr')
            perm_changes[name] = expr
            yield f'{indent}#$$ {name} = {expr}'
            p = generate_patterns(**{**perm_changes, **temp_changes})
            continue

        yield block

if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser()
    parser.add_argument('filename')
    parser.add_argument('--disable-prologue', action='store_true',
    ↪ default=False)
    parser.add_argument('--disable-epilogue', action='store_true',
    ↪ default=False)

    args = parser.parse_args()

    with open(args.filename) as f:
        text = f.read()
    first_line, rest = text.split('\n', 1)
    if not first_line.startswith('#!'):
        raise Exception('first line MUST be shebang!')
    prologue = 'from tiniflow.prologue import __node__, __edge__,
    ↪ __workflow__, on'
    epilogue = f'__workflow__.run({args.filename!r})'
    if (args.disable_prologue):
        prologue = ''
    if (args.disable_epilogue):
        epilogue = ''
    parsed = parse(rest)
    first_line = ''.join([prologue, first_line])
    print(first_line, ''.join(parsed), epilogue, end='', sep='\n')
```

A.2 flow.py

```
#!/usr/bin/env python3

from networkx import DiGraph
from os import getpid, getppid, execvpe, environ, fork, waitpid
from os import open as os_open, pipe, dup2, close, set_inheritable
from os import O_RDONLY, O_WRONLY
from sys import argv

noread  = os_open('/dev/null', O_RDONLY)
nowrite = os_open('/dev/null', O_WRONLY)
set_inheritable(noread,  True)
set_inheritable(nowrite,  True)

```

```
shell = environ.get('SHELL', '/bin/sh')

class Pipe:
    def __init__(self):
        self.read, self.write = pipe()
    def __repr__(self):
        return f'Pipe(read={self.read}, write={self.write})\n{hex(id(
    ↪ self))}'

class Command:
    def __init__(self, command, ifds=[], ofds=[], env={}):
        self.command = command
        self.ifds, self.ofds = list(ifds), list(ofds)
        self.pid = None
        self.env = {'TF_DATA_IN':'0',         'TF_DATA_OUT':'1',
                    'TF_CTRL_IN':f'{noread}', 'TF_CTRL_OUT':f'{
    ↪ nowrite}', **env, }

    def __repr__(self):
        return f'Command({self.command!r})\n{hex(id(self))}'

    def close_fds(self):
        for fd, _, _ in self.ofds:
            try: close(fd)
            except OSError: pass

    def __call__(self):
        self.pid = fork()
        if self.pid:
            return self.pid
        for ifd, isdata, name in self.ifds:
            set_inheritable(ifd, True)
            if isdata:
                dup2(ifd, 0)
            self.env[f'{name}_IN'] = str(ifd)
        for ofd, isdata, name in self.ofds:
            set_inheritable(ofd, True)
            if isdata:
                dup2(ofd, 1)
            self.env[f'{name}_OUT'] = str(ofd)
        env = {**environ, **self.env}
        execvpe(shell, [shell, '-c', self.command], env)

class Tee:
    def __init__(self, ifds=[], ofds=[]):
        self.ifds, self.ofds = list(ifds), list(ofds)
        self.pid = None
```

```python

    def __repr__(self):
        return f'Tee({self.ifds!r}, {self.ofds!r})\n{hex(id(self))}'

    def close_fds(self):
        for fd, _, _ in self.ofds:
            try: close(fd)
            except OSError: pass

    def __call__(self):
        self.pid = fork()
        if self.pid:
            return self.pid
        for ifd, _, _ in self.ifds:
            set_inheritable(ifd, True)
            dup2(ifd, 0)
        for ofd, _, _ in self.ofds:
            set_inheritable(ofd, True)
        args = [f'/proc/self/fd/{ofd}' for ofd, _, _ in self.ofds]
        command = f'tee {" ".join(args)} >/dev/null'
        execvpe(shell, [shell, '-c', command], environ)

def create_xgraph(graph, nodes, isdata, name):
    xgraph = DiGraph()
    for u in graph.nodes():
        xgraph.add_node(nodes[u])
        if graph.out_degree(u) > 1:
            t = Tee()
            xgraph.add_node(t)
            xgraph.add_edge(nodes[u], t)
            for v in graph.successors(u):
                xgraph.add_edge(t, nodes[v])
        else:
            for v in graph.successors(u):
                xgraph.add_edge(nodes[u], nodes[v])

    pipes  = {}
    pgraph = DiGraph()
    for u in xgraph.nodes():
        pgraph.add_node(u)
        for v in xgraph.successors(u):
            if (u, v) not in pipes:
                p = Pipe()
                for n in xgraph.predecessors(v):
                    pipes[n, v] = p
            p = pipes[u, v]
            pgraph.add_edge(u, p)
            pgraph.add_edge(p, v)

    for u, v in xgraph.edges():
        p = pipes[u, v]
        u.ofds.append((p.write, isdata, name))
        v.ifds.append((p.read, isdata, name))

    return xgraph

def run(data_graph, *extra_graphs, filename=''):
    nodes = {n: Command(n.contents) for n in data_graph.nodes()}
    data_xgraph = create_xgraph(data_graph, nodes, isdata=True, name=
    ↪ 'TF_DATA')
    extra_xgraphs = [create_xgraph(g, nodes, isdata=False, name='
    ↪ TF_CTRL')
                    for g in extra_graphs]
    all_nodes = set(data_xgraph.nodes())
    for xg in extra_xgraphs:
        all_nodes.update(xg.nodes())

    pids = {node(): node for node in all_nodes}

    # in the parent: wait for all children
    while pids:
        pid, rc = waitpid(-1, 0)
        if pid in pids:
            pids[pid].close_fds()
            del pids[pid]
```

A.3 prologue.py

```
from networkx import DiGraph
from .flow import run
from contextlib import contextmanager

class Node:
    def __init__(self, contents, metadata):
        self.contents = contents
        self.metadata = metadata
    def __repr__(self):
        return f'Node({self.contents!r}, {self.metadata!r})'

class Edge:
    def __init__(self, contents, metadata):
        self.contents = contents
        self.metadata = metadata
    def __repr__(self):
        return f'Edge({self.contents!r}, {self.metadata!r})'
    def __iter__(self):
        nodes = list(self.traverse())
        return zip(nodes, nodes[1:])
    def traverse(self):
        for x in self.contents:
            if isinstance(x, Node):
                yield x
            if isinstance(x, Edge):
                yield from x.traverse()

class WorkflowGroup:
    def __init__(self):
        self.workflows = {}
        self.current   = None
    def add_node(self, *args, **kwargs):
        return self.current.add_node(*args, **kwargs)
    def add_edge(self, *args, **kwargs):
        return self.current.add_edge(*args, **kwargs)
    def run(self, *args, **kwargs):
        for flow in self.workflows.values():
            flow.run(*args, **kwargs)
    def new(self, name=None):
        self.current = self.workflows[name] = Workflow()
        return self.workflows[name]
    @contextmanager
    def __call__(self, item):
        if isinstance(item, int):
            workflow = list(self.workflows.values())[item]
        else:
            if item not in self.workflows:
                workflow = self.new(item)
            else:
                workflow = self.workflows[item]
        previous = self.current
        self.current = workflow
        yield
        self.current = previous
    def __getitem__(self, item):
        if isinstance(item, int):
            return list(self.workflows.values())[item]
        else:
            return self.workflows[item]

class Workflow:
    def __init__(self):
        self.nodes, self.edges = [], []
    def add_node(self, contents, metadata=(), *args, **kwargs):
        node = Node(contents, metadata)
        self.nodes.append(node)
        return node
    def add_edge(self, contents, metadata=(), *args, seps=(), **
    ↪ kwargs):
        seps = {s.strip() for s in seps[0]}
        if len(seps) != 1:
            raise TypeError('cannot mix data & control edges in the
    ↪ same line')
        if not isinstance(metadata, tuple):
            metadata = (metadata, )
        if '-' in seps:
            metadata = (*metadata, on.control)
        elif '|' in seps:
            metadata = (*metadata, on.data)
        edge = Edge(contents, metadata)
        self.edges.append(edge)
        return edge
    def run(self, filename):
        data_graph    = DiGraph()
        control_graph = DiGraph()
        for edge in self.edges:
            if on.success in edge.metadata:
                for i, node in enumerate(edge.traverse()):
                    if i == 0: continue
                    node.metadata = ('tf-success',)
            elif on.failure in edge.metadata:
                for i, node in enumerate(edge.traverse()):
                    if i == 0: continue
                    node.metadata = ('tf-failure',)
            elif on.always in edge.metadata:
                for i, node in enumerate(edge.traverse()):
                    if i == 0: continue
                    node.metadata = ('tf-always',)

            if on.start in edge.metadata:
                for node in edge.traverse():
                    node.metadata = ('tf-start', *node.metadata)
                    break
        for node in self.nodes:
            if node.metadata:
                node.contents = f'{" ".join(node.metadata)} {node.
    ↪ contents!r}'
        for edge in self.edges:
            if on.data in edge.metadata:
                for u,v in edge:
                    data_graph.add_edge(u, v)
            elif on.control in edge.metadata:
                for u,v in edge:
                    control_graph.add_edge(u, v)
        for node in self.nodes:
            data_graph.add_node(node)
            control_graph.add_node(node)
        run(data_graph, control_graph, filename=filename)

class Tags:
    start   = 'start'
    always  = 'always'
    success = 'success'
    failure = 'failure'
    data    = 'data'
    control = 'control'

on = Tags()
__workflow__ = WorkflowGroup()
__workflow__.new()
__node__ = __workflow__.add_node
__edge__ = __workflow__.add_edge
```

A.4 .flow examples

Some of the later flow file examples in the thesis may rely on incomplete functionality. Please reference `tini.flow` on GitHub for the most complete working version.

A.4.1 `linear.flow`

```
#!/usr/bin/env tf

** a = seq 1 70
* b = grep 3
* c = ts
* d = nl

% =  a | b | c | d
```

A.4.2 `branch.flow`

```
#!/usr/bin/env tf

* a = seq 1 10
* b = grep 3
* c = grep 5
* d = ts
* e = nl
% = a | b | d
% = a | c | e
```

A.4.3 graph.flow

```
#!/usr/bin/env tf

* a = annotate a
* b = annotate b
* c = annotate c
* d = annotate d
* e = annotate e

* src1 = seq 0 1
* src2 = seq 2 3
* src3 = seq 4 5

% = src1 | a | b
% = src2 | a | b
% = src3 | a | b
% = b | c
% = b | d
% = b | e
```

A.4.4 cyclic.flow

```
#!/usr/bin/env tf

* src = seq 0 0
* a = annotate a
* b = annotate b
* c = annotate c
* d = while read line; do sleep .1; echo $line; done | nl

% = src | a | b | c | a
% = c | d
```

A.4.5 dsl.flow

```
#!/usr/bin/env tf

* a = seq 1 3
* b = nl

% = a | b
```

```

> another example
    * a = seq 1 3
    * b = ts

    % = a | b

> another workflow
    * a = seq 1 3
    * b = nl
    * c = ts

    % e = a | b
    % = e | c

> long linear workflow
    * tail = seq 1 3
    for x in range(10):
        * n = annotate x
        % tail = tail | n


def xyz():
    * x = annotate x
    * y = annotate y
    * z = annotate z
    % e = x | y | z
    return e

> longer linear workflow
    * tail = seq 1 3
    for x in range(3):
        % tail = tail | xyz()

> linear workflow with f-strings
    * tail = seq 1 3
    for x in range(3):
        * n := annotate {x}
        % tail = tail | n

> sigil-changing
    * a = seq 1 3
    * b = nl
    * c = ts
    * d = annotate d
    $ nodesep = -
    % = a - b
    $$ nodesep = ->
    % = b -> c
    % = c -> d
```

A.4.6 grandpa.flow

```
#!/usr/bin/env tf

from networkx import Graph, DiGraph

def is_soninlaw(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in [u, *marriage.neighbors(u)]:
        for child in parentage.successors(parent):
            for spouse in marriage.neighbors(child):
                if spouse == v:
                    return True
    return False

def is_mother(u, v):
    if parentage.nodes(data=True)[v]['gender'] != female:
        return False
    for parent in [v, *marriage.neighbors(v)]:
        for child in parentage.successors(parent):
            if child == u:
                return True
    return False

def is_uncle(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for gparent in parentage.predecessors(parent):
                for gparent in [gparent, *marriage.neighbors(gparent)
    ↪ ]:
                    for child in parentage.successors(gparent):
                        for child in [child, *marriage.neighbors(
    ↪ child)]:
                            if child == v:
                                return True
    return False

def is_brotherinlaw(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for child in parentage.successors(parent):
                for spouse in marriage.neighbors(child):
                    if spouse == v:
                        return True
    return False


def is_brother(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for child in parentage.successors(parent):
                if child == v:
                    return True
    return True

def is_stepmother(u, v):
    if parentage.nodes(data=True)[v]['gender'] != female:
        return False
    for parent in parentage.predecessors(u):
        for stepparent in marriage.neighbors(parent):
            if stepparent == v and v not in parentage.predecessors(u)
    ↪ :
                return True
    return False

def is_grandchild(u, v):
    for parent in [u, *marriage.neighbors(u)]:
        for child in parentage.successors(parent):
            for child in [child, *marriage.neighbors(child)]:
                for gchild in parentage.successors(child):
                    if gchild == v:
                        return True
    return False

def is_grandmother(u, v):
    if parentage.nodes(data=True)[v]['gender'] != female:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for gparent in parentage.predecessors(parent):
                for gparent in [gparent, *marriage.neighbors(gparent)
    ↪ ]:
                    if gparent == v:
                        return True
    return False


def is_grandpa(u, v):
    if parentage.nodes(data=True)[v]['gender'] != male:
        return False
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            for gparent in parentage.predecessors(parent):
                for gparent in [gparent, *marriage.neighbors(gparent)
    ↪ ]:
                    if gparent == v:
                        return True
    return False

def get_mothers(u):
    for parent in parentage.predecessors(u):
        for parent in [parent, *marriage.neighbors(parent)]:
            if parentage.nodes(data=True)[parent]['gender'] == female
    ↪ :
                yield parent

partner = 'partner'
child   = 'child'
male    = 'male'
female  = 'female'

marriage  = Graph()
parentage = DiGraph()

def __edge__(nodes, relationship, *args, **kwargs):
    if relationship == partner:
        u, v = nodes
        marriage.add_edge(u, v)
    elif relationship == child:
        u, v = nodes
        parentage.add_edge(u, v)

def __node__(description, gender, *args, **kwargs):
    marriage.add_node(description, gender=gender)
    parentage.add_node(description, gender=gender)
    return description

# http://gean.wwco.com/grandpa/
# Many, many years ago when I was twenty-three
* me = twenty-three year old || male
```

```

# I was married to a widow who was pretty as could be.
* widow = pretty widow married to me || female
% = me | widow || partner

# This widow had a grown-up daughter who had hair of red.
* redhead = red-haired grown-up daughter of widow || female
% = widow | redhead || child

# My father fell in love with her and soon they, too, were wed.
* dad = my father || male
% = dad | me || child
% = dad | redhead || partner

# This made my dad my son-in-law and changed my very life
assert is_soninlaw(me, dad)

# For my daughter was my mother, 'cause she was my father's wife.
assert is_mother(me, redhead)

# To complicate the matter, even though it brought me joy
# I soon became the father of a bouncing baby boy.
* baby = my son || male
% = me | baby || child

# My little baby then became a brother-in-law to dad
# And so became my uncle, though it made me very sad
assert is_brotherinlaw(baby, dad)
assert is_uncle(me, baby)

# For if he was my uncle, then that also made him brother
# To the widow's grown-up daughter, who, of course, was my step-
    ↪ mother.
assert is_brother(redhead, baby)
assert is_stepmother(me, redhead)

# My father's wife then had a son who kept them on the run
* fathers_son = son of father's widow || male
% = redhead | fathers_son || child

# And he became my grand-child, 'cause he was my daughter's son.
assert is_grandchild(me, fathers_son)

# My wife is now my mother's mother, and it makes me blue
assert any(is_mother(mother, widow) for mother in get_mothers(me))

# Because, although she is my wife, she's my grandmother too.
```

```
assert is_grandmother(me, widow)

# If my wife is my grandmother, then I am her grandchild
# And every time I think of it, it nearly drives me wild
# For now I have become the strangest case you ever saw
# (This has got to be the strangest thing I ever saw)
assert is_grandchild(widow, me)

# As husband of my grandmother, I am my own grandpaw.
if is_grandpa(me, me):
    print("OMG, I am actually my own grandpa!")
```

A.4.7 make.flow

```
#!/usr/bin/env tf

from os import listdir

flows = [filename for filename in listdir('.')
                  if filename.endswith('.flow')]

> test/all
    * tail = true -- on.start
    for flow in flows:
        * run   := clear && ./{flow} -- on.success
        % tail   = tail - run

for flow in flows:
    > test/{flow}
        * run   := clear && ./{flow}

    > watch/{flow}
        * wait  := inotifywait -e create,modify ../tiniflow/*.py {
    ↪ flow} 2>/dev/null
        * run   := clear && ./{flow}
        * debug := python -m tiniflow.dsl ./{flow}

        % = wait - run - wait   -- on.always, on.start
        % =        run - debug  -- on.failure

if __name__ == '__main__':
    from sys import argv, exit
    if not argv[1:] or argv[1] == '--complete':
        print('\n'.join(sorted(x for x in __workflow__.workflows if x
    ↪ )))
    elif argv[1] == '--interact':
        from code import InteractiveConsole
        print('Try typing the following to see all workflows:\n\t__workflow__.workflows.keys()')
        print('Try typing the following to run a workflow:\n\t__workflow__.workflows["test/linear.flow"].run(__file__)')
        InteractiveConsole(globals()).interact('')
    elif argv[1] == '--describe':
        target = argv[2]
        for node in __workflow__.workflows[target].nodes:
            print(f'* {node.contents}')
    else:
        for arg in argv[1:]:
            try:
                __workflow__[arg].run(__file__)
            except KeyboardInterrupt:
                break
    exit()
```

A.5 Benchmarks

A.5.1 `throughput.sh`

```
#!/bin/zsh

pv_format="total time: %t    transfer rate: %a    total bytes: %b"

typeset -A tests
tests=(
    "raw throughput"
    "dd if=/dev/zero bs=1G count=10 2>/dev/null | pv -F '$pv_format'
    ↪ > /dev/null"

    "raw sequence"
    "seq 1 $((1e6)) | pv -F '$pv_format' > /dev/null"

    "pipeline"
    "seq 1 $((1e6)) | grep 3 | ts | pv -F '$pv_format' > /dev/null"
)

for test in "raw throughput" "raw sequence" "pipeline"; do
    echo $test
    eval "${tests[$test]}"
    echo
done
```

A.5.2 `throughput.flow`

```
#!/usr/bin/env tf

pv_format = "total time: %t    transfer rate: %a    total bytes: %b"

> raw throughput
    * a  = dd if=/dev/zero bs=1G count=10 2>/dev/null
    * b := pv -F '{pv_format}' > /dev/null
    % = a | b

> raw sequence
    * a  = seq 1 $((1e6))
    * b := pv -F '{pv_format}' > /dev/null
    % = a | b

> pipeline
    * a  = seq 1 $((1e6))
    * b  = grep 3
    * c  = ts
    * d := pv -F '{pv_format}' > /dev/null
    % = a | b | c | d

if __name__ == '__main__':
    for name, workflow in __workflow__.workflows.items():
        if not name: continue
        print(name)
        workflow.run(__file__)
        print()
```

A.5.3 `mcilroy.sh`

```
#!/bin/sh

tr -cs A-Za-z '\n' |\
        tr A-Z a-z |\
        sort |\
        uniq -c |\
        sort -n |\
        tail -n1
```

A.5.4 mcilroy.flow

```
#!/usr/bin/env tf

* translate = tr -cs A-Za-z '\n' | tr A-Z a-z
* count     = sort | uniq -c | sort -n
* report    = tail -n1

% = translate | count | report
```