Gib:
The Generic, Embeddable, Interactive Programming Language

Brian Koropoff

June 1, 2007

Contents

1 Introduction **3**
1.1 Abstract . . . 3
1.2 Organization . . . 3
1.3 Works . . . 3

2 Conception **4**
2.1 Principles . . . 4
2.2 Motivation . . . 4
2.2.1 Command Line Paradigm . . . 4
2.2.2 Existing Interfaces . . . 6
2.2.3 Scripting Languages . . . 6
2.3 Intended Applications . . . 7

3 Goals and Design **9**
3.1 Interactivity . . . 9
3.1.1 Completion . . . 10
3.2 Semantics . . . 11
3.2.1 Runtime Architecture . . . 11
3.2.2 Relationship with Interactivity Requirements . . . 12
3.3 Embedding . . . 12
3.3.1 Generality . . . 13
3.3.2 Adaptability . . . 13
3.3.3 Pragmatics . . . 15

4 Applications **19**
4.1 GibShell . . . 19
4.2 GibShell GTK . . . 21
4.3 GibForge . . . 23
4.4 GibXml . . . 24
4.5 GibNgram . . . 25

5 Conclusion **26**

6 Appendices **27**

1 Introduction

1.1 Abstract

Gib is a new programming language designed primarily for interactive use – that is, as the backend interpreter for a command line interface (CLI). It was envisioned as a generic, reusable component which would serve as a scripting layer and command-based user interface as part of a larger application. Indeed, Gib – unlike languages such as C or Python – does not constitute a complete programming language or environment by itself. Rather, it acts as the core of a potentially infinite set of domain-specific languages sharing a carefully-honed subset of syntax and semantics, ready to be specialized for the needs of a particular program and environment.

1.2 Organization

This document will explore the development of Gib in several stages. First, the factors behind its conception will be examined, including motivation, intended applications, and high-level notions about the language's mode of operation. These general considerations will then be expanded and refined into a more concrete set of goals and design considerations, at which point the specific design decisions that went into Gib will be elucidated. The final section will examine several example applications of the language.

1.3 Works

The works produced as part of this research project include the following:

- The Gib core runtime, a C++ library
- GibShell, a text-based Unix command line shell based on Gib
- GibShell GTK, a command line shell based on Gib with graphical enhancements
- GibForge, a plugin to the QuakeForge game engine that replaces the standard command line console with Gib
- GibXml, a library and command line frontend for accessing and navigating XML files

2 Conception

2.1 Principles

The field of programming languages is already quite saturated with contenders, including a healthy array of popular general-purpose languages (Java, C++, Python) and countless numbers of domain-specific languages (MatLab, SQL). Cross-pollination of features and programming paradigms has become so pervasive that most modern languages offer at least the following set of features or equivalents: functions, exceptions, garbage collection, concurrency, packages/namespaces, etc. In addition to these staples, most languages also draw from a pool of features such as classes/inheritance, first-class functions, generators/co-routines, and so forth. Indeed, the task of the modern language designer is akin to cooking – to choose from among these stock features (ingredients) and bind them together with a syntax and runtime model (cooking), hoping for a compelling end product. These choices are guided by several principles:

- Complexity – avoid too many or too few features
- Compatibility – features should work well together and not duplicate functionality
- Audience – features should ease the programming task for the intended users of the language

2.2 Motivation

Given these principles, the motivation for designing a new language is to fulfill the needs of a particular audience through experimentation with a novel combination of features and paradigms. In the case of Gib, the target audience – users of command line interfaces – requires a very particular set of language design choices. However, it is first worth exploring why a new language should be targeted toward this audience in the first place – why create yet another command line interface?

2.2.1 Command Line Paradigm

Despite decades of research and growth in the field of graphical user interfaces, command line interfaces remain a staple in the workflow of technical users. Unix and Unix-like systems (e.g. Linux) are still largely administered through command line interfaces despite the availability of an array of robust graphical work environments. Even on Microsoft Windows, a primarily graphical platform, the command line shell has recently received a major upgrade after overwhelming user demand. In addition, many individual applications on a variety of platforms offer command line interfaces as a supplement or adjunct to their graphical interfaces.

The enduring appeal and staying power of command line interfaces can be traced to several advantages for both users and developers. These advantages are rooted in the way the underlying features of an application are mapped into its user interface – or rather, how instructions issued to the user interface are mapped to underlying operations of the application. This is a non-trivial concern, as the set of problems an application can solve is usually infinite while the interface itself is finite. Thus, from a theoretical perspective, we might view the user interface – command-based or graphical – as a language which describes the desired operation of the underlying program. Just

as a finite grammar can define an infinite language, a finite user interface description can allow an infinite set of input patterns and thus, behavior of the program.

Under this interpretation, the advantages of command line interfaces are immediately apparent. Command line languages, as formal context-free languages, are similar in structure to natural human languages, being of potentially unlimited recursive depth. The ability of human languages to express an infinite number of possible meanings is due to this recursive nature, as the set of grammar rules and the lexicon are both finite in size. Thus, command line languages (and formal programming languages in general) allow a human user to describe the desired behavior of the program in a manner analogous to the process of natural language utterance – generation of a linearly-ordered string of symbols with a potentially infinite deep structure.

Unlike a formal context-free language, a graphical interface is usually limited to some pre-programmed depth – for example, a finite chain of nested dialog windows. This is not a technical limitation, but a pragmatic one – graphical elements are generally designed by hand to be intuitive and visually appealing, and to conform to the limitations of the graphical environment such as screen size. Thus, programmatically-generated or infinitely-recursive interfaces are usually avoided for usability reasons. A graphical interface may employ some manner of visual programming; for example, many applications allow the user to query a database by visually assembling various predicates into a compound Boolean expression (see figure 1). This approach is, in theory, as expressive as a recursive programming language, but is more complicated to use and less time-efficient.

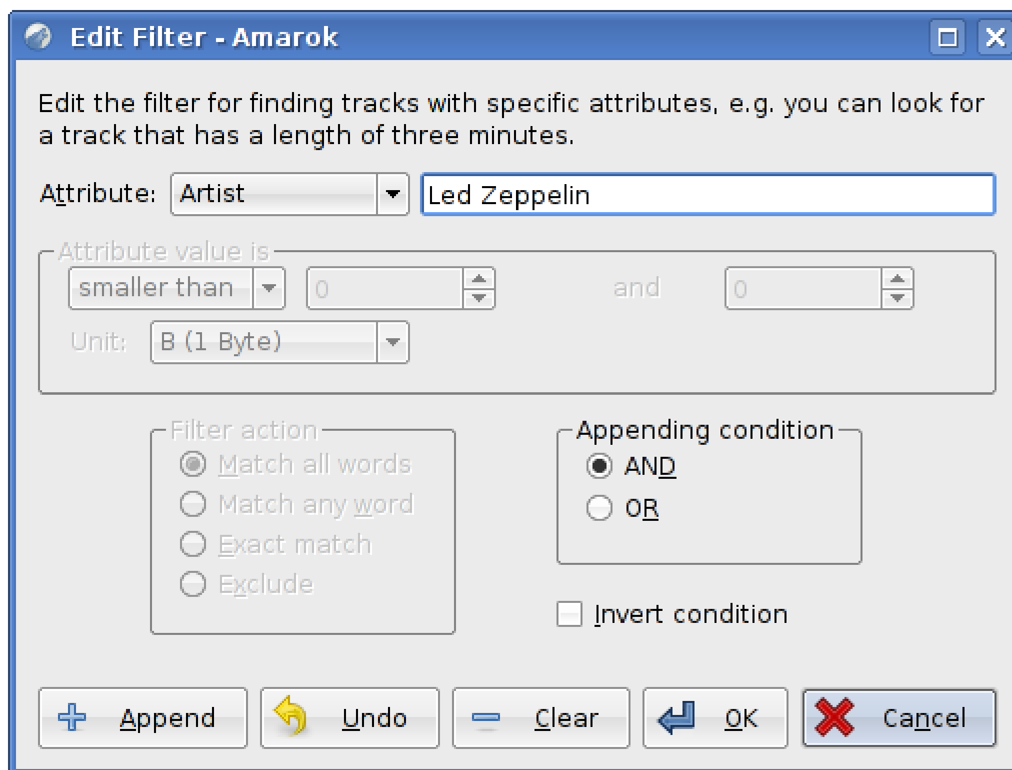

Figure 1: A graphical query editor in a music application

Thus, graphical interfaces generally grow in size and complexity with problem set of the application, while command line interfaces can remain largely unchanged. This reduces the burden of the developer and spares the user increasingly convoluted and expansive interfaces.

2.2.2 Existing Interfaces

Given the merits of command line interfaces, a question still remains: why create yet another? On Unix platforms in particular, command line shells are nearly as numerous as text editors. However, the vast majority of these shells share a common set of flaws and limitations that make them suboptimal.

The most critical flaw is lack of a powerful underlying semantic model. Classical command line languages generally do not provide a compelling set of data types, provide few data structures beyond primitive arrays, and rely heavily on external programs to perform even basic calculations. For example, the popular bash shell views all data as a string, provides only an array data structure, and must call an external program just to test the equality of two strings.

Many of these flaws can be traced to the lack of modern programming language features such as garbage collection. Powerful data structures in particular tend to require dynamic allocation of memory on the heap. Without garbage collection, this memory would need to be freed manually by the user – unacceptable in an interactive session. Thus, many of these shell languages were designed to prevent any operation which could leak memory, severely limiting their abilities.

Reliance on external programs is a symptom of a more general flaw: shells make huge assumptions about their platforms, relying on the existence of a filesystem, external programs which can be invoked by name, and other trappings of Unix environments. Although these assumptions are not a handicap for a Unix shell per se, they prevent the shells from being adapted to other environments or maintaining some common denominator of functionality that allows code reuse. For example, a set of utility functions written in bash script cannot be reused in the debugger gdb, even though both feature command line interaction. Not only does this situation make code reuse impossible, it necessitates a large degree of duplicate effort; every program with a command line must implement it from scratch.

2.2.3 Scripting Languages

There is a class of languages which have overcome many of these flaws: embeddable scripting languages. This class includes popular choices such as Python, Perl, and Lua. These languages can be linked into an existing application and hooked into its internals in order to allow robust scripting with minimal effort. Languages such as Python and Perl were designed as stand-alone interpreters but can also be embedded, while languages such as Lua were designed strictly for this purpose. Gib is similar to Lua in this regard.

Scripting languages avoid many flaws of classical command line languages. Perl in particular was created as an integrated alternative to bash and its accompanying Unix programs such as sed, grep, and awk. These languages feature automatic memory management, a wide array of data types, and powerful, easy data structures. In fact, it is becoming increasingly common for applications to offer Perl/Python/Lua consoles as part of the graphical interface. Although these languages are powerful and capable of being used interactively, they also lack many of the expected features and design characteristics of interactive languages.

For example, most command line interfaces offer some manner of automatic completion; by striking a key such as tab, the user is presented with a list of possible ways to complete the current partially-typed command. This feature, in combination with other language elements such as filename pattern matching (globbing), significantly accelerates common use cases such as manipulating files.

Interactive languages also exhibit certain systematic syntax and semantic properties which lend themselves to interactive use. In order to minimize typing, interactive languages eliminate unnecessary "punctuation" and other syntactic fluff for the common case. Consider the following examples:

```
python: foo("bar", "bob", baz)
bash: foo bar bob $baz
```

In interactive languages, the most common operation in a session is executing a command. Thus, shells such as bash eschew Python-style function call syntax, adopting simple space-separated arguments in lieu of Python's parentheses and commas. Another notable difference is bash's "quotation by default." String literals in Python must be enclosed in quotes, while in bash a typed string evaluates to itself without any special delimiters. However, this entails the use of a dollar sign as a sigil in front of variables, which Python does not require. Thus, interactive language must sometimes shift syntactic complexity away from common interactive use cases to less common ones. It is worth noting that this predilection for simple common-case syntax can be a disadvantage for scripting; bash in particular has rather verbose or convoluted syntax for flow control structures, arithmetic expressions, conditional expressions, and so forth.

Compared to pure scripting languages, interactive languages tend to operate on much higher-level objects, manipulating entire files or calling external programs with a single command. Similar operations, even in very high-level scripting languages, often require use of a more fine-grained interface with multiple steps.

Given these systematic differences, these is still value in designing a new language which combines the modern features and power of a scripting language with the conveniences of interactive languages.

2.3 Intended Applications

Gib was conceived as a language to be embedded into a larger application to quickly add a command line interface. Examples of applications which could use such a command line layer are numerous:

- Debuggers
- Command line shells
- Games
- Computer algebra systems
- Administrative consoles

- File managers

In order to fit into such a wide array of applications, Gib was envisioned as a small C++ library which would be linked into the embedding application. The application would then attach to several well-defined hooks to connect Gib to the internals of the application. In this way, Gib could be written in a very generic manner by delegating specific details to the client application.

3 Goals and Design

Given these motivating factors and notions about the proper operation of a command line language, Gib was designed with a bevy of specific goals in mind. These goals fall into three broad categories: interactivity, semantics, and embedding. Interactivity encompasses features that contribute to a comfortable and efficient command line environment; semantics refer to underlying language meaning, features, and power; and embedding concerns the interface provided to programmers to adapt Gib into their particular applications. Each of these categories and their corresponding goals will be enumerated along with contingent design considerations and the final decisions that shaped Gib's architecture.

3.1 Interactivity

Based on the aforementioned aspects of easy-to-use command line interfaces, the following guidelines were devised:

- The syntax should be straight-forward and designed to minimize typing as much as possible.
- Automatic completion of partially-entered commands should be possible.
- Some manner of globbing or pattern matching should be supported.
- It should be possible to run tasks in the background.
- Manipulating high-level objects, such as files, should be simple.
- Combining commands together in interesting ways should be simple, both conceptually and syntactically.
- It should be possible to easily redirect input and output of commands to and from files and other data sources.

Rather than reinvent the wheel, it was decided that Gib would be patterned after bash in order to satisfy these requirements. The Unix paradigm, enshrined in its command line shell, has not endured for so long without reason; indeed, it handily satisfies all the above guidelines. This is no coincidence, as these guidelines are a reflection of the features and niceties that modern users of command lines have come to expect, largely due to the use of Unix or Unix-influenced environments.

Bash syntax is quite simple for common cases and employs very few "punctuation" or "delimiting" tokens or characters. It supports automatic completion, globbing, background and foreground tasks, quick manipulation of files or entire sets of files and directories, command pipelines, and input/output redirection. Command pipelines in particular are noteworthy as the crux of the Unix workflow: run a command to produce or read data, send it through another command to manipulate it, and so on.

At its core, a pipeline is simply a model for composition of existing functionality; the commands themselves implement some generic operation (read a file, count number of lines, transform strings), while the composed whole solves some specific problem. Thus, the pipeline paradigm

is not terribly different from any other designed to encourage generic programming: procedural programming with standard function libraries, object-orientation with inheritance, first-class functions, etc. Most of these models, however, are a poor fit for an interactive environment. Performing subclassing in a syntactically frugal way is tricky, and mere economy of keystrokes does not mitigate the difficulties of implementing an interface or contract on-the-fly, Java's anonymous inner classes being a prime example. A functional approach would be preferable, as lambda syntax can be quite sparse and there is only one "interface" to implement. However, complex compositions of first-class functions can result in deeply-nested expressions and subtle type safety errors, a situation familiar to ML or Haskell programmers. The procedural paradigm seems an ideal match as command line interfaces are inherently procedural. However, the necessity of individually executing each step of the operation and saving intermediate results is discouraging; compare the economy of map in a functional language to achieving a similar result in C.

Pipelines can be considered a compromise that allows easy composition of existing functions while maintaining the "do x, then y, then z" approach of procedural programming. A pipeline $c1|c2|c3$ could be viewed as mapping the function composition $c3 \bullet (c2 \bullet c1)$ over some list of input values, an approach to solving a problem that would not seem out of place in a functional language. Unix pipelines operate on arbitrary chunks of binary data rather than discrete elements, but the principal is the same.

This approach is syntactically clean and intuitively represents the notion of procedural processing; the underlying mechanism of a data "assembly line" is semantically simple but powerful. Although there are complicating factors in practice, the implementation is theoretically trivial: a simple extension of the stream mechanism used for input/output redirection. Thus, pipelines stand among globbing, paths, input/output streams, and filesystem-like data sources as staple bash features inherited by Gib.

Gib syntax is very reminiscent of bash, but contains numerous improvements and extensions; for examples, refer to Appendix A. The underlying architecture is, at the surface, informed by bash sensibilities, and was specifically designed to allow for pipelines, a navigable "filesystem," globs and paths, and threads (background tasks). However, these sensibilities alone provided only guidelines; in order to satisfy all of the goals behind Gib's creation, other factors had to be considered before settling on a particular architecture.

3.1.1 Completion

The Gib completion framework was designed to support nearly any syntactic element in the language. In order to realize this, completion support was integrated directly into the abstract syntax tree representation. The particular technique was inspired by a common approach to compiler construction in object-oriented languages: syntax tree nodes implement some manner of "emit" or "lower" method which generates an equivalent representation in an intermediate language; recursive calls to the method are performed on child nodes as needed. Analogously, Gib syntax tree nodes support a "complete" method which generates a list of completion suggestions, delegating the task to child nodes as appropriate. The Gib compiler is also designed to handle incomplete commands as gracefully as possible, allowing a partial syntax tree to be constructed and subsequently queried.

This approach is in contrast to shells such as bash, where completions are constructed in a non-syntactic manner involving ad-hoc parsing of the partial command. Gib currently supports

completion on namespaces and nested namespaces, file and directories, variables, and command names; further support is merely a matter of adding completion logic to the remaining node classes.

3.2 Semantics

Although patterning Gib after bash was advantageous for its suitability to interactive use, care had to be taken to avoid many of the aforementioned pitfalls that plague traditional command line languages, particularly their tendency toward bare-bones semantics and runtime models. It was decided that Gib would avoid this issue by providing a runtime system comparable to modern dynamic scripting languages, featuring the following at a minimum:

- Automatic memory management
- Multiple primitive datatypes: string, integer, float, etc.
- Arrays
- Associative arrays ("dictionaries")
- First-class functions
- Lexically- and dynamically-scoped variables
- Namespaces
- Standard control structures: if, while, foreach, ...
- C-style expressions
- A full contingent of binary and unary arithmetic and Boolean operators with support for overloading

In short, Gib was designed to be a flexible scripting language on par with Python and its ilk in addition to a command line environment. Toward that end, architectural elements from these languages were borrowed for use in Gib.

3.2.1 Runtime Architecture

In the tradition of modern interpreted languages such as Perl, Gib was designed to be compiled on-the-fly to an internal bytecode representation which would be executed by a virtual machine. The virtual machine itself was patterned vaguely after the Java virtual machine, borrowing its operand-stack-based computation model, local variable array, and argument passing conventions. However, it was supplemented by several additions and modifications to support a dynamically-typed scripting language: array and dictionary construction, closure construction, dynamic dictionary and namespace lookups, overloading-aware arithmetic instructions, bulk string concatenation, lexical and dynamic variable frames, and so on. Unlike Java, Gib does not directly use primitive types; all values in Gib are passed by reference, with wrapped primitives being immutable.

Because the compiler and virtual machine are both invoked at runtime, no serialized on-disk bytecode representation was devised; rather, the same in-memory data structure is used for both bytecode emission and interpretation.

3.2.2 Relationship with Interactivity Requirements

In order to adequately support features inherited from bash, it was necessary to integrate shell-like constructs into a more ordinary scripting language runtime model. Pipes and input/output streams required the addition of explicit virtual machine support for stream opening, closing, reading and writing, as well as pipe instantiation. It was also necessary to somehow represent the notion of the current standard input and output streams; the easiest solution which presented itself was to store these values in special internal variables in the dynamically-scoped variable frame. This approach did not require additional virtual machine instructions specifically for the task and could later be applied to other dynamic state that needed to be stored such as the current directory, handles for spawned background commands, or application-specific data. Under this paradigm, redirecting the standard input/output streams simply entails entering a new dynamic variable frame with the appropriate variables set.

Globs were initially envisioned as a second-class language construct, albeit with consistent language-wide availability. That is, they could be used to index into dictionaries in addition to appearing as arguments to commands. However, after this implementation proved too limiting, they were re-imagined as first-class objects in a generic pattern matching framework that allowed for them to be passed, stored, and composed with themselves and other matching primitives. This system is supported by virtual machine instructions to create first-class glob and path objects and match them against data sources such as dictionaries; special syntax was added to explicitly expand a pattern to its set of matches.

In order to maintain symmetry and simplicity, the filesystem is viewed by Gib as an associative data structure similar to a dictionary, allowing it to be accessed in the same way. This permits files to be specified and queried by paths and globs as in bash, while being treated as first-class objects with a more powerful representation than a simple path string.

Certain features, such as C-like expressions and flow control, proved difficult to reconcile with the bash-like syntax. In particular, a string such as `5+5` appearing unadorned as a command argument should be interpreted as a literal string, as in the bash command `echo 5+5`. A string such as `5 + 5` appearing in a command would be interpreted as three separate arguments due to bash's space-sensitive syntax. In order to support C-style, whitespace-insensitive expressions, a tiered syntax model was adopted. Under this model, bash-style syntax applies at the level of individual commands, while anything within at least one layer of parentheses is parsed using a C-style grammar, including full math and logical operators with precedence, C-style function calls, and sigil-less variables. This allows the syntactic paradigm appropriate to the task to be used with a minimum of overhead. The actual implementation of this system was simple with modern compiler construction tools, although it significantly inflated the size of the grammar.

3.3 Embedding

Preserving the embeddability of Gib proved to be the design challenge that demanded the most substantial thought and research. The issues that presented themselves in this undertaking fell into three categories: generality, adaptability, and pragmatics. Generality concerns preserving bash-derived semantics and features in an unknown application or environment. Adaptability refers to the potential for the portions of a client application that need to be command line-accessible to be adequately mapped into the Gib runtime model. Pragmatics are practical concerns such as

resource usage, development time, and programming interfaces associated with the use of the Gib core library.

3.3.1 Generality

The semantics of bash, although well-honed for daily interactive use, rely on several assumptions about the runtime environment that are troublesome for an embeddable language: the presence of a filesystem, system environment variables, executable on-disk programs found through a search path, etc. Most operating systems provide these facilities, making it theoretically possible to write a cross-platform command line shell. However, using these facilities directly may not be desirable for a given embedding application. For example, some applications may not wish to allow arbitrary filesystem or system program access for security reasons. In many applications, commands should not dispatch external programs but rather call program-internal functions, and access to the filesystem may be unnecessary in lieu of some other navigable data structure such as a database. In essence, the purpose of Gib is not to interact with a computer system in general but rather the specific facilities of a particular application.

Because it was not possible to make assumptions about the needs or semantics of embedding applications, it was necessary to find some way to preserve the superficial forms of the bash features without specifying a particular underlying representation or implementation. This underlying implementation would have to be delegated to application programmer, who would be cognizant of the particular needs of the program and the abilities of the runtime environment. Luckily, the object-oriented programming paradigm provides a clean, well-understood solution to this problem. By crafting a set of interfaces which specify some minimum functionality necessary for the semantics of Gib, the language itself could be written on top of abstract data types and preserve guaranteed core features. The responsibility would fall on the application programmer to implement these interfaces, thus fleshing out the abstract core language into a functional command line system.

The selection and design of these interfaces, however, was not trivial. The remaining two categories of concerns, adaptability and pragmatics, remained to be satisfied.

3.3.2 Adaptability

In order to adapt to the needs of any potential client application, the interfaces used in Gib had to be kept as flexible as possible while still preserving the semantics of the language, particularly those originating from bash. For this reason, bash was established as the gold standard; Gib would, at a minimum, have to be flexible enough to be used as a basis for reimplementing a bash-like Unix shell. This entailed careful analysis and contemplation of bash's key features: filesystem navigation and globbing, data streams and input/output redirection, pipes, etc. By viewing bash as a potential client application of Gib, and its features as concrete implementations of the desired abstract interfaces, it was possible to work backwards and extract the most basic interfaces which conceivably could be specialized back into full implementations.

After careful consideration, the following set of interfaces – hereafter referred to by their C++ class names – were selected as the core set that would be used by Gib:

- `Object`
  As the base class of all data that can be manipulated by the Gib interpreter, `Object` is note-

worthy for establishing the most basic operations that all types should support, such as hashing. Rather than enumerating the methods of the interface here, operations that required methods to be hoisted into the base class will be mentioned as part of the interfaces that use them.

- `Table`
  An associative array interface, resembling Java's `Map`, which stores and fetches key/value pairs. As `Tables` can potentially be infinitely nested, this abstract data structure is suitable for forming trees; thus, the "filesystem" used by Gib is a client-provided implementation of this interface. Tree structures in general are well-regarded for their ability to store and represent almost any data imaginable in a powerful way. XML, for example, is a tree-structured file format designed exactly for this purpose. Also notable are "virtual" filesystems such as Linux's /proc, which presents program, kernel, and driver information and options as a nesting of simulated directories and files. To a shell, these files are indistinguishable from physical on-disk data, and can be read, written, queried, and manipulated using the same set of tools and techniques. Indeed, the Unix "everything is a file" philosophy has proved the filesystem robust enough to serve as the interface to a variety of underlying systems. `Tables` are even more general, being capable of representing cyclic directed graphs. Thus, this interface was selected for its ability to expose almost any conceivable data to a command line. In order to support indexing and querying of tables, methods for hashing and performing pattern matches (such as globbing) were hoisted into the `Object` class; Java takes a similar approach to implement `HashMap`.

- `InputStream`, `OutputStream`
  These interfaces allow for streams of data to be read and written, respectively. Methods for opening input and output streams were hoisted into `Object` in order to guarantee that input/output redirection could be defined for all conceivable arguments. This also allowed `InputStreams` to double as iterators by overriding these hoisted methods in container classes such as `Table`. In order to be as flexible as possible, streams were designed to operate on single, atomic objects (passed by reference) rather than blocks of untyped binary data. This was particularly necessary because streams are used to implement pipes, which should be capable of passing objects between Gib functions without marshaling them to a byte-oriented representation.

- `Function`
  This interface represents anything invokable, such as a command or function. Indeed, there is no distinction made between the two in Gib; first class functions written in Gib as well as program-specific commands such as built-in functions or external programs are all implementations of this interface. By making Gib functions an interface, client code can expose callable portions of itself to Gib in any way the programmer might see fit.

- `Thread`
  This interface represents a thread of execution within the interpreter. In order to properly support command pipelines, which are run in parallel, Gib required the guaranteed availability of concurrent execution. The same mechanism is also used for background tasks.

Using these interfaces, it was possible to write the Gib compiler and runtime without foreknowledge of specific environments or implementation details. Selecting the most generic, flexible interfaces possible would enable Gib to fulfill the needs of applications with substantially different use case scenarios or data binding requirements.

Aside from the design of these interfaces, several additional features were added to Gib to maximize the ability of a client application to customize the language for its needs. For example, embedding applications can define custom data types simply by subclassing `Object` and registering overloaded operator definitions relevant to those types. This could be used to implement numeric or algebraic libraries among other applications. In order to allow some degree of syntax customization, an XML-like tag system was added to the grammar. The client application can register handlers for certain named tags; when these tags are encountered in the input, the enclosed data is passed verbatim to the handler and the result from the handler inserted into the parse tree. This could allow for custom data type constants, such as inline regular expressions, inline XML, large text or binary data lumps, database schema, or anything else imaginable. The application can even extend the abstract syntax tree class hierarchy with its own node types if necessary; the bytecode emission interface is also exposed.

3.3.3 Pragmatics

Although these interfaces and customization mechanisms solved the conundrum in theory, many practical concerns remained for both the language implementation and hypothetical users of the library. In particular, foisting the majority of implementation work on the application programmer would require substantial effort to make even basic use of Gib. A variety of design patterns, APIs, and tools were explored in order to reduce this workload.

The first step was to minimize the number of interfaces that the programmer must implement before Gib will function. Two approaches were taken here: first, to reduce the number of unique interfaces used; second, to provide partial or complete implementations for common cases to reduce the burden on the client application.

A great deal of complexity was eliminated by using the `Table` interface wherever possible, both language-internally and for integration into client code. In addition to representing the filesystem, Gib uses `Tables` to represent namespaces, variable frames, and dictionary and array data structures used by Gib code. The decision to make `Tables` double as both dictionaries and arrays was inspired chiefly by the Lua scripting language, which takes a similar approach. By using `Tables` for namespaces and variable frames, client applications can expose functions and data to Gib by implementing or manipulating the global environment frame to define the desired variables.

The situation was improved further by providing stock implementations for many of the interfaces. The `HashTable` implementation of `Table`, for example, is used internally for variable frames, arrays and dictionaries constructed by Gib code; there is no need for the client application to provide such basic functionality. When the client application wishes to expose data or functions to Gib through a table, such as by defining functions in the global environment, the `HashTable` implementation will suffice either as-is or with minor modification through subclassing. Even the "filesystem" backend used for globbing and navigation could be filled with a `HashTable` if such an approach is appropriate to the application. The `Function` interface also has a canonical implementation, Closure, which represents first-class functions defined by Gib code. The Gib library also provides an array of standard concrete data types that the language stipulates must be available,

such as `String`, `Integer`, and `Float`.

This approach is hardly groundbreaking; it is considered good practice for a library to provide convenient implementations of common functionality while allowing the client to customize behavior "by hand" if necessary. Also, much of the Gib runtime model – closures, primitive types, and so forth – is implementable without knowledge of the embedding environment or application, so there is little reason to leave such tasks to the application programmer. However, some interfaces proved to be a difficulty; implementing them would be a considerable task for the application programmer, but providing canonical implementations could cause incompatibilities on certain platforms or for certain programs.

The most troublesome was `Thread`. Gib required a concurrency model in order to guarantee the operation of command pipelines, but requiring the client application to implement a threading system – or at least a wrapper around one – is a steep request. This put the onus on the Gib library to provide a threading implementation that would play well with any conceivable client application or environment, an equally tall order.

Several solutions were considered. The first was to simply use system threads through a cross-platform threading library; this solution had the advantage of being well-established and requiring no extra implementation work on the part of Gib. However, it was ultimately rejected for several reasons, the most compelling being that applications using Gib would quite likely be single-threaded. By using system threads, any application using Gib would become susceptible to issues that classically plague multi-threaded programs. The application would have to be written or rewritten to perform locking/synchronization, despite making no use of threads itself. Languages such as Python address this problem by requiring explicit cooperation of the embedding application in order for threads to be supported by the interpreter, allowing the application to force interpreted code to also be single-threaded. As Gib must guarantee the availability of a concurrency model, this was an unacceptable solution.

Also considered were so-called fibers – lightweight, user-space threads scheduled and run by the application process rather than the operating system. This solution was attractive for two reasons. One, because scheduling was controlled in-process, a cooperative task switching system could be used to guarantee the atomicity of code; since a fiber gives up control voluntarily when it is in a known consistent state, explicit synchronization and locking is unnecessary. Two, such a system could appear completely transparent to the client application, thereby allowing Gib to support the use of threads with the application being none-the-wiser.

A variant of Python known as Stackless Python was turned to for inspiration for building a fiber implementation. Stackless Python eliminates use of the C call stack by Python code, instead using heap-allocated, garbage-collected stack frames to keep Python program state. By saving and restoring these frames, Stackless Python could support first-class continuations. This feature – commonly associated with the Scheme dialect of Lisp – allows threading, co-routines, exceptions, and many other call-stack-manipulating constructs to be implemented by explicitly saving and restoring the execution context; continuations are, in essence, a more general version of all these features. Because continuations could allow future additions to Gib such as exceptions, they were adopted as the fiber implementation mechanism.

However, Stackless Python's particular approach was still flawed. Because growth of the interpreter's stack could not correspond to growth of the C stack, native code sandwiched between interpreted code frames would have to explicitly save and restore state and make asynchronous calls back into the interpreter, a massive inconvenience. Modern Scheme implementations solve

this problem by saving and restoring the physical C stack rather than using heap-allocated stack frames. This introduces a caveat: in general, it is not safe to restore a particular context more than once, as doing so could cause double-free and similar errors in C or C++ code. However, as this behavior was not necessary to implement fibers, stack-copying continuations were chosen as a basis for the Gib thread system.

This solution did require some care in the way Gib was designed. In particular, a fiber that causes the process to block would block all running fibers; therefore, it became necessary to use asynchronous or non-blocking I/O, so that control could be transferred to the next fiber when an operation could not be completed. The `InputStream` and `OutputStream` interfaces were designed to support this, which would complicate matters for implementing applications. However, Gib provides default implementations to support pipes and reading/writing to and from `Tables`, as well as wrappers around standard C++ IO streams. In fact, it is not strictly necessary to provide an implementation unless default standard input/output streams are required.

In addition to providing as many canonical implementations as possible, Gib was designed with APIs to minimize the work necessary to implement these interfaces or otherwise customize Gib. Several well-known C++ tricks were employed to reduce or eliminate boilerplate code. Many Gib features, such as operator overloading or tag handling, required explicit registration of the handler with the Gib library, necessitating boilerplate initialization routines. However, Gib provides special C++ classes whose constructors register these handlers when run. By declaring a global static instance of the class, the constructor will automatically be run at program startup and the handler registered. Definition of both the handler function and the initializer class instance can be performed simultaneously using macros. For example, the following C++ snippet defines the addition operator for Gib `Integers`:

```
GIB_BINARY(OP_ADD, Integer, Integer, intr, first, second)
{
    return new Integer(first->getValue() + second->getValue());
}
```

Using this combination of macros with C++ static initializers, registering Gib library hooks becomes concise and automatic.

Binding native functions and data into Gib was still a challenge. In a typical use case scenario where a small library of built-in functions must be exposed to Gib, the client application programmer would have to subclass `Function` for each, instantiate each class and insert it into a `Table` which would act as a namespace, and then insert the table into the Gib interpreter. The first approach taken to reduce the necessary work was similar to the above example of operator overloading: use a macro to define and register a builtin function along with its desired namespace. The interpreter would scan these registered functions and namespaces during initialization and automatically import them into the global variable table.

This solution did not account for the fact that many library functions would frequently be written in Gib itself; namespaces defined by Gib code and those defined within native code had to be somehow merged together. Although feasible, it still created a situation where related functions within the same namespace had to be defined in and loaded from two separate sources. Even in Java, which uses a similar mechanism, the Java source code itself contains a prototype of the native method, keeping all relevant API information in one location.

The auto-registration approach was abandoned in favor of one that allowed inline C++ code to be used as the body of a Gib function definition. Syntactically, this took the form of a special `extern` keyword, followed by the name of the target language (usually C++) and a distinguishing identifier, followed by the actual inline code enclosed within special delimiters. The compiler would replace these expressions with lookups into a global table of known native functions using the identifier and a hash of the code block itself. However, in order for these `extern`s to successfully resolve, the actual compiled inline code had to have been registered in the table in advance.

To do this, an external tool called gib++ was developed that parsed a Gib source file and intercepted the `extern` references. These C++ code blocks were output as a C++ source file with boilerplate code to automatically insert the functions into the global table, along with the Gib source code itself as a static string constant. This essentially converted the Gib code into a self-contained C++ file containing everything necessary to execute the script.

Finally, a system was developed that allowed the programmer to bind all the converted Gib files together into a single named module with a bit of glue code. These modules would be registered automatically with the Gib library upon program startup using static initializers. When the `load_module` directive was encountered in Gib code, the appropriate module would be looked up and each script executed in sequence in the interpreter, with references to inline C++ code now correctly resolved and executed.

To allow further customization, a module loading system was added. The client application can register handlers to load unknown modules in response to `load_module` directives. A default handler was provided to resolve modules by searching for a shared library. Upon the library being loaded by the dynamic linker, the static initializers would run, automatically hooking custom tags, operator overloads, and namespace and function definitions into Gib. This allowed Gib libraries unassociated with a particular embedding application to be written and then used anywhere. For example, the GibXml library mentioned in the introduction can be loaded into any application embedding Gib to add XML support. The Gib standard library is also kept separate from the core runtime in this way, allowing applications that don't require it to avoid the resource usage.

As a final tool at the application programmer's disposal, an `AutoTable` system was devised to ease implementation of the `Table` interface. `AutoTable` is a series of complex C++ templates that reflect methods and fields of a C++ class into Gib as a table. This can be used either as a means of defining a namespace contained neatly within a single C++ class, or to instantiate multiple C++ objects with methods and fields accessible to Gib. Combined with C++'s support for multiple inheritance, this could allow Gib bindings for object-oriented C++ libraries to be written using `AutoTable` as a mix-in to the relevant classes.

The combination of these systems allows easy, powerful, dynamic creation and usage of libraries and native bindings with a minimum of code and maximum centralization and consistency. Some of the example applications using Gib required only a few hundred lines to fully interface with the core library, much of it written in Gib itself.

4 Applications

In order to evaluate the efficacy of Gib in actual deployment situations, several example applications and libraries were developed to exercise language and API features. These applications and their use of the Gib core library will be examined in brief.

4.1 GibShell

As the gold standard of Gib functionality and testbed for new features, GibShell was the most important application to be developed with Gib. GibShell is a text-based Unix command line shell using Gib for command and script evaluation. It consists of two parts. The first is a small library providing implementations of the core Gib interfaces, including:

- `Program`, an implementation of `Function` which dispatches an external Unix program when called, allowing Unix commands to be executed from the shell.
- A `Table` implementation representing a directory which queries the physical filesystem to list keys or retrieve values, along with a new `File` data type to represent physical files. This enables the filesystem to be navigated and queried from the shell using paths and globs, and command input and output to be redirected to and from files.
- Implementations of `InputStream` and `OutputStream` backed by Unix file descriptors, used by the `File` data type when asked for an input or output stream, as well as by `Program` to integrate Unix pipes into Gib command pipelines.
- A `Table` implementation backed by system environment variables, enabling them to be manipulated from the shell as if they were any other variable.
- A `Table` implementation backed by the system program search path which returns instances of `Program`, permitting Unix programs in the path to be run as if they were any other function in the global scope.

In addition to these essential elements, the library provides an enhanced version of `Pipe`, the class used for communication between elements in a command pipeline. The enhanced version, `UnixPipe`, allows direct communication over a Unix pipe when elements on both sides are Unix programs, meaning that a Gib command pipeline employing only Unix commands would theoretically run as fast as one in bash. A special version of the `Thread` class is also provided which tracks extended information such as process IDs and command line arguments. The library also contains additional API to help with common shell tasks and a Gib module of relevant builtin functions.

The second part of GibShell is the actual command line frontend, which uses the GNU readline library to read commands from the user and pass them to a Gib interpreter initialized with elements from the GibShell library. See figure 2 for an example session.

GibShell still lacks many features of mature command line shells such as bash, but serves as a proof of concept that Gib is sufficiently powerful to implement such a shell.

```
gib# ls
adapter.cpp  bytecode.cpp      float.hpp     glob.hpp     module.hpp  stream.hpp  table.hpp
adapter.hpp  bytecode.hpp      frame.hpp     integer.cpp  object.cpp  string.cpp  thread.hpp
boolean.cpp  continuation.cpp  function.cpp  integer.hpp  object.hpp  string.hpp  vm.cpp
boolean.hpp  continuation.hpp  function.hpp  Makefile.am  path.cpp    symbol.cpp  vm.hpp
builtin.cpp  exception.hpp     gc.hpp        Makefile.in  path.hpp    symbol.hpp
builtin.hpp  float.cpp         glob.cpp      module.cpp   stream.cpp  table.cpp
gib# echo $PATH
/home/brianhk/bin:/home/brianhk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/us
r/bin/X11:/usr/games:/home/brianhk/.local/bin
gib# ls ../gib*
../gib-lex.cc  ../gib-parse.cpp  ../gib-parse.h  ../gib-parse.output
gib# foreach value < {1, 2, 3, {foo = x, bar = y}} echo $value
1
2
3
{bar = y, foo = x}
gib# md5su
md5sum             md5sum.textutils
gib# md5sum *.cpp | cut -d" " -f1 | sort
06d103f8292c77423612aa6b9f95ee55
1142ec5229c35d145bf2d067fa5b12c3
150ee535f33e5d2f53589c20ecbc1ef0
159d9a4aec532cdc1571a49b00690398
1c366445b8c76906c5998de19feddf6c
23393f9212007958d63ebe0b3e29086e
2474809f3dca2dbd9b7637c7b5faf3e4
44f18f99aab63a7b419a3ff9320b9fea
48ccfa7dae7b3355d35bc328854971d1
63782a764f5a4261461d72375cca88b7
688ec102b434111eea7416ba1f3fc148
985b374c3aceeb73ad94afbabe5b3283
9d34246d05b9922ef5c4eb3fa3ce63c8
9e4dbed6a12db9ffbfb8058f4cf0070b
b4ff906f7a5c8c95e27521736e8be255
d58add2c30a067748e232d08e858dbdb
f32526b582e356c59740f1cd1c2b2f05
gib# echo {?*.hpp, ?Makefile*}
{continuation.hpp, gc.hpp, bytecode.hpp, frame.hpp, glob.hpp, vm.hpp, builtin.hpp, exception.hpp, ob
ject.hpp, table.hpp, stream.hpp, string.hpp, integer.hpp, symbol.hpp, boolean.hpp, function.hpp, ada
pter.hpp, path.hpp, thread.hpp, module.hpp, float.hpp, Makefile.in, Makefile.am}
gib#
```

Figure 2: GibShell

4.2 GibShell GTK

GibShell GTK is an alternate frontend to the GibShell library which provides a graphically-enhanced command line by subverting the traditional relationship between terminal and shell. On most modern Unix systems, terminal-based programs are often run inside a graphical terminal emulator which handles text input, output, and display. These emulators take the place of physical dumb terminal hardware that was used in the early days of Unix systems. As such, modern Unix systems such as Linux provide so called "pseudo terminals," allocatable character-based interfaces which allow the display-side of a terminal connection to be handled by a program rather than physical hardware or a serial port. Thus, communication between the frontend terminal application and the shell takes place over a primitive stream-based messaging system, with several extended controls to manage screen size, cursor position, and so forth.

GibShell GTK was an experiment to discern what might be possible if this arbitrary schism between shell and terminal were eliminated. Using a terminal emulator widget for the GTK graphical interface toolkit, it presents a terminal to the user that is connected directly to itself rather than a shell program in a separate process. The terminal emulator embeds the Gib and GibShell libraries and takes care of all command execution directly. This permits complete awareness and cooperation between the graphical and textual portions of the interface, allowing for interesting mingling of the graphical and command line interface paradigms.

The new terminal program boasts several small but significant improvements over a traditional shell-in-emulator setup. It uses a true graphical text entry widget which allows for editing of commands using the mouse and standard key combinations – an impossibility in a traditional setup where the terminal emulator in unaware of the workings of the shell's input line. In addition, it provides pop-up completion where possible commands and arguments are displayed in a convenient list and can be selected with the mouse or keyboard, similar to the URL entry bar in many modern web browsers. In a normal setup, the shell has no recourse but to dump the completions to the terminal or cycle through them with subsequent presses of the completion key. A terminal emulator could in theory provide a graphical input field without being integrated with the underlying shell. However, it would encounter several issues. First, completion logic would need to be implemented from scratch, duplicating the facilities of the shell. Second, heuristics would be required to determine whether keyboard input should be directed to the input line or to the actual terminal device, a choice best determined by the presence of an executing foreground command; in GibShell GTK, this shell-internal state is readily available.

Beyond improvements to the fundamental shell experience, GibShell GTK provides a side pane with several extended features. The first is an integrated directory navigator and file chooser which can synchronize the directory being browsed with the shell's working directory. This is a feature offered by some Unix file managers with integrated terminals, but their implementations are typically less robust. Updating the shell's current directory is achieved by inserting a `cd` command into the terminal, producing undesirable results in the presence of a foreground task such as a text editor. It also pollutes the command history with entries not actually typed by the user. In addition, there is no way for the file manager to react to directory changes made by the shell, short of periodically polling its process's current directory; in GibShell GTK, the interpreter informs the interface when the working directory has changed.

In addition to the file browser, a basic text editing widget is available in another side pane tab. This widget subclasses the Gib `Object` class and allows for the contents to be accessed

using standard Gib stream mechanisms; it can be referenced through the `gibshell` namespace and used as an input/output redirection target for commands and command pipelines. Compared to juggling temporary files, this system facilitates more convenient manual previewing and editing of intermediate results. A third tab provides a real-time list of processes and threads currently being run by the shell, complete with process IDs and command line arguments where appropriate.

Although these additions are seemingly trivial, in the hands of more seasoned user interface researchers GibShell could be used to investigate new ways of integrating command line and graphical interfaces for improved power and productivity.

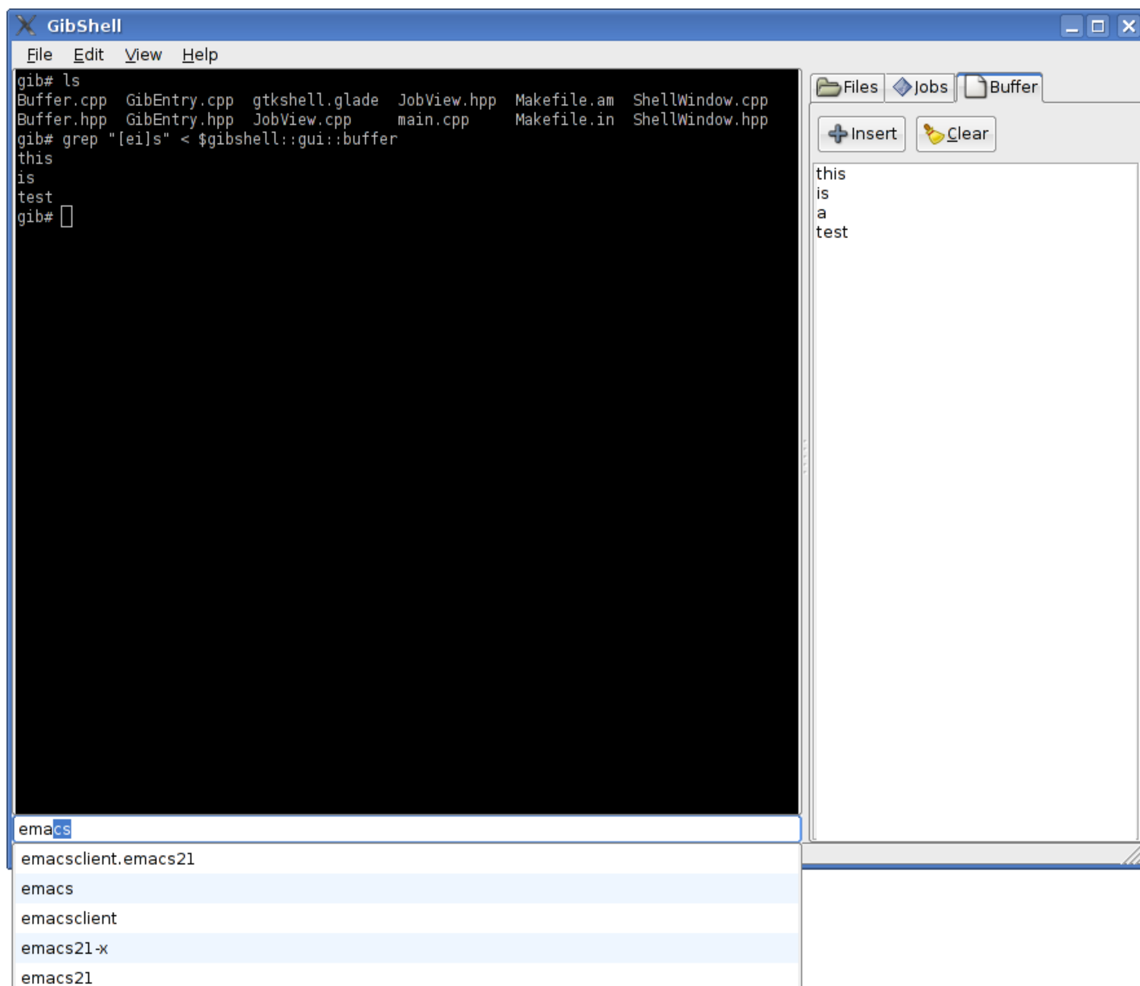

Figure 3: GibShell GTK

4.3 GibForge

GibForge is a plugin for the QuakeForge engine, a 3d computer game system based on the source release of iD Software's Quake. Quake is notable in this context for being among the first computer games to present the user with a command line console for advanced configuration and limited scripting. Although the console was primarily a vehicle for development and debugging, it was quite central to the behavior of the game: actions such as moving and shooting were actually triggered by console commands; pressing the associated keys simply executed them transparently. Portions of the network protocol were also dependent on the console. For example, the QuakeWorld variant of the server steps the client through the initialization process by sending chunks of text to be executed as console commands.

Quake's console proved extremely influential, and to this day many computer games – even by other software companies – feature pull-down consoles in addition to the usual graphical interfaces. This makes such games a perfect use case scenario for Gib – rather than implementing a console language from scratch, developers could simply embed Gib to provide a rich, scriptable command line for development, debugging, and advanced users.

In order to test Gib's suitability for this purpose, a plugin was developed which replaced the usual QuakeForge console with one based on Gib. The features were as follows:

- Integration with existing console commands and variables. Built-in commands for the QuakeForge console were rewired into Gib as subclasses of `Function`, and console variables were made accessible as ordinary variables through an implementation of `Table`.
- Integration with the Quake File System (QFS). Quake and its descendants present a sandboxed view of the filesystem which limits accessible files to user configuration directories and bundled game data (graphics, sounds, levels, etc.). In addition, game data can be contained either in the user's configuration directory, in the system game directory, or in archive files in the system game directory. Thus, the QFS is actually a union of files and directories from several sources. GibForge provides a `Table` implementation backed by QFS access functions, allowing transparent access to all Quake data from Gib.
- Programmable completion. Gib's completion mechanism is far more powerful than that included in the standard QuakeForge console, and also allows for custom completion logic written in Gib. For example, the completion logic for the command map, which causes a particular level to be loaded, could be overridden to suggest valid level names (see figure 4).

Due to Gib's API design, GibForge amounted to only a few hundred lines of code, a lot of it boilerplate that interfaces with QuakeForge's plugin system.

Figure 4: GibForge

4.4 GibXml

GibXml wraps the DOM API of the Xerces XML library in order to present XML files to Gib as a series of nested tables. It was created as an example application of the Gib `Table` interface to non-filesystem backends. The accompanying program, gibxml, provides a simple command line interface where XML data can be loaded and manipulated like standard Gib dictionaries. Because the `Table` interface is also used for representing filesystems, an XML file can be set as the "current directory" and navigated using bash-like commands such as `cd` and `ls`, as well as accessed using paths and globs.

Because the GibXml library is an ordinary Gib module, it can be loaded into any application using Gib – e.g. GibShell. It also provides a tag handler that allows inline XML to appear in Gib source code enclosed by `<xml>` and `</xml>`; the data will automatically be parsed and evaluated to its `Table` representation.

```
brianhk@briscoe:~/workspace/gibxml$ ./gibxml
xml: node = <xml><foo name="test"><bar>one</bar><bar>two</bar></foo></xml>
xml: echo $node[0]
<foo name=test>
xml: echo $node[0][name]
test
xml: echo $node[0][?*]
{<bar>, <bar>, test}
xml: echo $node[?0/*/*]
{one, two, test}
xml: cd $node
xml: ls
0 -> <foo name=test>
xml: cd 0
xml: ls
0 -> <bar>
1 -> <bar>
name -> test
xml: cd 1
xml: ls
0 -> two
xml: cd ..
xml: ls
0 -> <bar>
1 -> <bar>
name -> test
xml: echo ?0/0
one
xml: exit
```

Figure 5: GibXml

4.5 GibNgram

GibNgram, written as part of an unrelated computational linguistics project, is a module providing a C++ implementation of an n-gram statistical language model, exposed to Gib through use of the `AutoTable` feature. The library is not part of a dedicated program – rather, it is dynamically loaded into GibShell, where scripts can access the filesystem to parse training data files and feed them into the model. From there, the command line can be used to investigate the model and test it against evaluation data. This library is notable for being a "real world" use case of Gib, rather than a proof-of-concept. It allowed the n-gram implementation to be rapidly prototyped by removing the need to manually write a front end with parsing logic. Testing and debugging were aided by availability of a command line interface with full completion and access to Unix commands and files. The result was a marked decrease in development time.

Figure 6: GibShell using the GibNgram module to model Japanese phonotactics

5 Conclusion

Gib combines the ease of use of an interactive command line environment with robust scripting capabilities, providing users a familiar bash-like interface without the limitations of traditional shells or the inefficient syntax of popular scripting languages. Developers are offered a convenient pre-packaged solution which integrates tightly into their applications with a minimum of hassle.

These capabilities were made possible by careful analysis of computer interface and programming language paradigms, and subsequent synthesis into a runtime platform and object-oriented API that permit a generic, extensible base to be specialized into sundry potential applications.

Gib, in tandem with similar languages such as Microsoft's PowerShell, could precipitate a command line renaissance in a world increasingly dominated by graphical environments. Programs such as GibShell GTK hint at future research into the integration of command line methodology and graphical interface sensibilities, paving the way for more powerful applications and efficient workflows.

6 Appendices

Appendix A: Example Code Listings

gunit.gib: rudimentary unit test framework

```
// gunit unit test framework
// designed to be run from GibShell
// define gunit namespace
def gunit
{
        // configurable setting
        show_pass_results = $<false>
        // runs a single test (a table containing a
        // run function and an expected result)
        def run_test (name, struct)
        {
                import struct::expect
                import struct::run
                result = ( run() )

                if (result == expect)
                        if (show_pass_results)
                                echo "$name passed ($result)"
                        else
                                echo "$name passed"
                else
                        echo "$name failed ($result != $expect)"
                return (result == expect)
        }
        // runs a fixture of tests (a table mapping test
        // names to individual tests
        def run_fixture (fixture)
        {
                total = 0
                failed = 0
                passed = 0
                foreach test < ( gib::table::keys(fixture) )
                {
                        if (run_test(test,fixture[$test]))
                                passed = (passed+1)
                        else
                                failed = (failed+1)
                        total = (total+1)
```

```
                }
                echo "$total test(s), $failed failed"
                return {$total, $passed, $failed}
        }
}
```

tests.gib: Front-end to gunit.gib

```
// runs all tests in gib scripts in the current directory
// that start with test_
source gunit.gib // pull in gunit definitions
gunit::show_pass_results = $<false>
import gunit::run_fixture // import the function we need
results = {0,0,0}
fixtures = 0
// constructs an array by expanding a glob, then iterates
// the array
foreach file < {?test_*.gib}
{
        echo "-- $file --"
        // define a fixture by executing the script file
        // inside a namespace definition
        def fixture
        {
                source $file
        }
        // run the fixture, tabulate results
        result = ( run_fixture(fixture) )
        foreach i < {0,1,2}
        {
                results[$i] = (results[$i] + result[$i])
        }
        fixtures = (fixtures + 1)
        echo
}
total = $results[0]
passed = $results[1]
failed = $results[2]
echo "-- Tests complete --"
echo
echo "$fixtures fixture(s), $total test(s), $failed failed"
```

test_thread.gib: test fixture for threads and pipes

```
def pipes
```

```
{
        def double_filter ()
                foreach value gib::stream::write (value*2)
        expect = {4,8,12,16}
        def run()
        {
                result = {}
                double_filter < {1,2,3,4} | double_filter > $result
                return $result
        }
}
def unix_pipes
{
        def dup_filter () foreach value gib::stream::write $value$value
        def generate () foreach value < {a,b,c,d} gib::stream::write $value
        def merge()
        {
                result = ""
                foreach value
                {
                        result = $result$value
                }
                gib::stream::write $result
        }
        expect = {aaaabbbbccccdddd}
        def run()
        {
                result = {}
                generate | cat | dup_filter | cat | dup_filter | merge > $result
                return $result
        }
}
def concurrency
{
        expect = {1,a,2,b,3,c}
        def run()
        {
                result = {}
                {
                        foreach val < {1,2,3}
                        {
                                result = {@$result, $val}
                                gib::thread::wait;
                        }
                } &
```

```
                first = $&
                {
                        foreach val < {a,b,c}
                        {
                                result = {@$result, $val}
                                gib::thread::wait;
                        }
                } &
                second = $&
                gib::thread::join $first
                gib::thread::join $second
                return $result
        }
}
```

Appendix B: Trivia

- Gib stands for “Gib isn’t Bash,” due to its superficial similarity to the popular Unix shell but vastly different underlying architecture.
- The majority of the core code was written in a four month time frame, with subsequent additions and enhancements occurring on a sporadic as-needed basis. The example applications were written in approximately one month.
- The core runtime consists of approximately 26,000 lines of code.
- Compiled as a shared library, the core of Gib is approximately 600kb; this figure could be reduced further by more prudent use of C++ templates and other optimizations.