2011

Fuga: A Homoiconic Object-Oriented Programming Language

Francisco Mota

FUGA:

A HOMOICONIC OBJECT-ORIENTED PROGRAMMING LANGUAGE

A Thesis Submitted

in Partial Fulfillment

of the Requirements for the Designation

University Honors with Distinction

Francisco Mota

University of Northern Iowa

May 2011

This Study by: Francisco Mota

Entitled: Fuga: A Homoiconic Object-Oriented Programming Language

has been approved as meeting the thesis or project requirement for the Designation University Honors with Distinction

04/29/11
Date — Eugene Wallingford, Honors Thesis Advisor

5/4/11
Date — Jessica Moon, Director, University Honors Program

1. INTRODUCTION

Fuga is a programming language that integrates two different traditions in programming language design: homoiconic programming, and object-oriented programming. Individually, these traditions can aid program construction and reduce the time required to develop a computer application. However, these two traditions have never been combined. This is an unexplored area in programming language design and implementation, so Fuga is the first programming anguage to combine the homoiconic and object-oriented paradigms. Fuga is an attempt to explore this area, and to turn this exploration into a finished product.

This thesis describes the design and implementation process of Fuga, and my reflections on it. Apart from this thesis, this project has lead to the creation of two additional documents:

- the Fuga specification, which formally defines the syntax and semantics of the language, as well as any other requirements that an implementation of the language must conform to, and
- the Fuga interpreter, which conforms to the formal specification. The interpreter can be used as a standalone program, or embedded into other programs to provide scripting support.

The specification is included as Appendix A. The interpreter can be downloaded from `http://github.com/fmota/fuga`. The source code is released under the MIT License, which allows free redistribution and modification to the source code and object files.

2. SOURCE REVIEW

In this section we look into the background of Fuga. We discuss object-oriented programming, metaprogramming, and homoiconic programming. We discuss programming languages that have evolved or used these paradigms, influencing Fuga in some way.

2.1 Object-Oriented Programming

Alan Kay came up with the term "object-oriented programming" to describe his work with the Smalltalk programming language [Kay 1993], but the term has evolved as it has come into wider use. For our purposes, object-oriented programming (OOP) can be defined as the practice of structuring programs in terms of **objects**, which are a grouping of functions and data.

In non-object-oriented programming languages, programs are composed of functions and data that interact in an unstructured way. In an object-oriented programming language, functions and

data that are related are grouped together into objects, such that the interaction between functions and data is more well defined and easier to understand.

In Smalltalk, the original object-oriented programming language, all objects belong to a class. An object's class determines which functions it contains, and what kind of data it can hold. Objects that belong to the same class behave in the same way. To add behavior to the system, the programmer must create a new class, or modify an existing one. The programmer cannot modify the behavior of an object directly.

With the release of Smalltalk-76, all classes became objects as well. Since all objects in Smalltalk must belong to a class, classes must also belong to a class. For this reason, Smalltalk-76 introduced a new class called "Class," which is the class to which all classes belong. But this only raises the question -- which class does "Class" belong to?

Because of this, David Ungar and Randall B. Smith [2007] thought that the class system was complicated and unnecessary, so they sought to develop an object-oriented programming language without classes. In this language, Self, objects can fully describe their own behavior without requiring an external entity, such as a class. Objects can also inherit behavior from other objects, as a replacement for the class system. Thus, similar objects were still grouped together -- not under a common class, but under a common **prototype**. This classless style of object-oriented programming came to be known as prototyping. When an object inherits from another object, we call the latter object a "prototype" for the first object.

There are multiple ways to implement prototyping. Astudillo and Shilling [1993] identified the two main ways in which objects can inherit from other objects:

- Delegation: The new object contains a pointer to the prototype, and the new object keeps track of any functions and data that were added or changed. Whenever a message is sent to the new object, we first look at the new object's functions and data. If we fail to find what we're looking for, we then try to find it in the object's prototype. If we fail to find it there, we look into the prototype's prototype, and so on. This means that a sent message might require looking in various objects, but it has the advantage that changes made to a prototype are propagated to all of its children.
- Concatentation: The new object copies all of the prototype's functions and data, and adds its own. This means that there are now two completely independent objects, so changes in one object don't affect the other. This can be a disadvantage, because if the programmer wanted to change all objects that inherited from a specific prototype, the programmer

would need to come up with some way of keeping track of them. Another potential disadvantage is that a naive approach to concatenation will lead to increased memory consumption.

Self and most other prototyping languages tend to use delegation because it is simpler to implement efficiently, and because changes in prototypes are propagated downwards automatically. Some languages do use concatenation, such as Kevo and Agora. Today (in 2011), the most widely used used prototyping language is JavaScript, and it uses the delegation style: JavaScript objects hold a special slot called `__proto__` that links an object to its prototype [Kienle 2010]. Lua is another contemporary language that uses prototyping with delegation [Ierasumlichy 2007].

2.2 Metaprogramming and Homoiconic Programming

Metaprogramming is the act of modifying the way one programs, by programming. For example, creating an application that generates the code for another application is a form of metaprogramming. Although most languages support metaprogramming to some extent, metaprogramming is used most in the realm of homoiconic programming languages.

A homoiconic language is a language which represents code and data in the same way. In a non-homoiconic programming language, there is a disparity between what the code talks about, and how the code is structured. A homoiconic language is designed to get rid of this disparity. This is typically accomplished by using one of the main data types of a the language to also represent code.

Homoiconic languages go hand-in-hand with strong metaprogramming support. Code is data, so manipulating code is as easy as manipulating data.

The canonical example of this is Lisp, a language that specializes in list processing, and whose code is a list that contains lists that contains lists, and so on. Lisp makes it very easy to work with lists, so Lisp also makes it easy to work with Lisp code. The mechanism that allows Lisp to manipulate its own code is Lisp's macro system, and it is widely considered one of the most powerful metaprogramming tools ever invented. According to Guy Steele and Richard Gabriel [1993], macros are responsible for much of Lisp's malleability, and are part of the reason that Lisp is such a diverse language.

The Io programming language [Dekorte 2003], and its offspring Ioke [Bini 2009], also offer powerful metaprogramming support. Essentially, they both allow the programmer to define functions that don't evaluate their arguments, but give access to the code of the arguments.

These functions can perform arbitrary operations on their arguments, generating new code in their stead, or evaluating them in a completely different way. Io's metaprogramming mechanism is more powerful than Lisp's, but is also slower.

However, neither Io nor Ioke are homoiconic to the degree that Lisp is. In Io, the most important data structure is the object, and the code structures are represented internally as a kind of object. Unlike Lisp, it's not obvious how these code object are structured, and metaprogramming requires either a reference manual, or a lot of experimentation. Another difference between the two is that in Lisp, most data is represented with the same syntax as code, so no evaluation is necessary in order to obtain most data. In Io, there is no way to represent an object directly in code, so evaluation is necessary. Thus, Io still suffers from the disparity between code and data, and cannot rightfully be considered a homoiconic programming language.

Nevertheless, Io is an important step, because it is an object-oriented programming language that is close to being homoiconic. As explained in section 4, Fuga was inspired by Io, and inherits most of its syntax and semantics directly from Io. Io is Fuga's predecessor, and is the reason why Fuga is object-oriented *and* homoiconic.

3. DEFINITIONS

What follows is a list of terms and their definitions according to how they are used in this thesis. Some of these terms have broader or similar meanings in the literature, but most of these terms are used according to canon.

- **function** : A function is an action that takes some input and generates some output.
- **method** : A method is a function associated with a particular object.
- **object** : An object is a value that contains data and methods on that data.
- **OOP (Object Oriented Programming)** : OOP is programming with objects, and structuring your programs in terms of objects rather than in terms of disjointed functions.
- **prototype** : In the context of OOP languages, a prototype is an object that other objects inherit from. As discussed in section 2.1, prototypes can be extended through delegation or through concatenation.

  In the context of software development, a prototype is a working version of a system created early in the software development life-cycle [Ince and Hekmatpour, 1987]. A prototype is used to clarify and solidify the requirements of a system, and to reduce overall

implementation effort.

- **slot** : In many prototyping OOP languages, including Fuga, objects are made up of slots. Typically, slots are a (name, value) pair. In Fuga, the name is optional, and slots can also hold documentation.
- **primitives** : Primitives hold some form of data that isn't inherent in the object system. Some OOP languages compromise their object-orientation by including primitives that lie outside of the object system. For example, Java primitives are not objects. By contrast, Fuga has primitives, but all primitives are objects.
- **lexer (lexical analyzer)** : A component of interpreters and compilers, a lexer takes a stream of characters and converts into a stream of tokens, simplifying the input to the next stage of the interpreter or compiler, the parser.
- **parser** : A component of interpreters and compilers, a parser takes a stream of characters or a stream of tokens, detects syntax errors, and converts the input into a syntax tree. Depending on the application, a parser might convert the input directly into a more useful form (that is, not a syntax tree). This is the case with Fuga, where the parser converts tokens directly into code objects.
- **homoiconicity / homoiconic programming** : Homoiconicity is the paradigm of having a language's syntax match the kinds of data which the language typically manipulates. In a homoiconic programming language is code and data are represented the same way.
- **code object** : A code object is data that represents code, in a homoiconic language. In an object-oriented language, code objects are also normal objects.
- **evaluation** : Evaluation is the action of interpreting a code object in order to obtain its value, or to produce side effects.
- **scope** : The environment in which a code object is evaluated is called its scope. The scope determines how an identifier is bound to a variable or to a value.
- **scoping** : Scoping is the rules by which scopes are organized and used.
- **lexical scoping** : Lexical scoping is scoping based on the structure of the code.
- **dynamic scoping** : Dynamic scoping is scoping based on the call stack.
- **receiver scoping** : Receiver scoping is scoping based on the receiver of the message.

- **lazy evaluation** : Lazy evaluation is the practice or paradigm of evaluating a piece of code only when its result is needed.
- **thunk** : A thunk is a value that has yet to be evaluated. Once the value is needed, it will be evaluated. Once evaluated, it is no longer necessary to evaluate the value again. Thunks are used to implement lazy evaluation.
- **first-class thunk** : A thunk split into its components, the code object and the scope, such that they are accessible to the programmer, is a first-class thunk. First-class thunks enable metaprogramming and the creation of new control structures.
- **garbage collector** : The garbage collector is the part of a program or language that keeps track of data that is no longer being used, in order to reclaim that memory for newer data.
- **space leak** : A space leak occurs when data is not being used, and will never be used, but will never be deleted by the garbage collector. Space leaks make programs run out of memory and run slowly.

4. CREATIVE PROCESS

In this section we will talk about the inspiration behind Fuga, the various decisions that were made along the way, and any other aspects of the creative process that are worth mentioning.

In section 2.2, we mentioned that Io is the closest thing there is to a homoiconic object-oriented language, but it isn't quite there. Io is great at working with objects, but its code objects are composed of messages [Dekorte 2010], so there is a disparity between Io code and Io data. Because of this, there is no way to describe an object directly using Io's syntax. This seems like a major oversight for a nearly-homoiconic language.

The inspiration for Fuga was to "fix" Io. In Fuga, it would be possible and natural to describe an object directly using the language's syntax. The syntax revolves around objects, making Fuga truly homoiconic, unlike Io.

4.1 A Functional Nature

At this point, Fuga's only defining feature was its syntax. Where was the rest of the language? Would it be a good idea to lift everything else from Io directly?

That question was answered by experimenting with the new syntax informally. By having a way to express objects directly, Fuga tended to have a declarative, functional nature. By contrast, objects in Io are constructed imperatively, so Io tends to have an imperative nature. This

indicates that these languages have different philosophies, so it would not be a good idea to copy everything from Io.

From there, I created prototypes of Fuga, coming up with many different versions of Fuga's interpreter in order to try out the various design possibilities. This enabled me to make the qualitatively best choices for Fuga's design. In the following sections I will go into detail about how specific aspects of Fuga's design evolved.

4.2 All Slots are Indexed

The first difficult design decision was regarding the nature of slots in Fuga. In Self and most of its descendants, slots were simple `(name, value)` pairs. This proved problematic in Fuga because functions in Fuga use an object as input, deriving the function's arguments from that object's slots. Since slots were purely `(name, value)` pairs, Fuga could not support positional arguments, only keyword arguments. This was a problem because programmers are more familiar with positional arguments, and forcing programmers to use keyword arguments exclusively reduces the programmer's ability to express themselves.

This problem was solved by allowing `name` to range over the natural numbers, in addition to ranging over symbols. This way, both keyword and positional arguments were made possible.

However, a greater problem persisted even in the face of this change. Unless all slot names are integers, there is no simple order of evaluation for an object's slots. The order in which slots appear in code is not the order in which slots are evaluated. This is highly unintuitive, and presents serious technical difficulties.

I developed a prototype that could evaluate slots in this way, evaluating some slots sequentially, and evaluating all other slots by need. It was at this point that lazy evaluation was introduced into the design of Fuga: in order to cope with the non-linear order of evaluation, the interpreter created thunks for each named slot, and the thunks would only be evaluated once the named slot was invoked. This was a huge increase in complexity, so I looked for a way to get rid of the non-linearity problem.

The solution was for all slots to be indexed, and to make slot names optional. Positional function arguments are then the default -- all slots can be accessed positionally -- and keyword arguments can be implemented by checking for slot names. The order of evaluation problem is solved since slots are evaluated in order. Indexing all slots makes Fuga easier to use and easier to implement.

4.3 Code Slots are Nameless

A second issue regarding slots also concerned homoiconicity. From the beginning, named slots could be specified using the equals (=) operator. Originally, this meant that the slots in the code object had the given name, and evaluation was built around preserving the names in the code object. This caused a problem: what happens when the code object redefines an important slot, such as `eval`? Trying to call the `eval` method on that code object would cause an error, and this was a tricky edge case to handle.

There were two possible solutions: avoid calling methods on code objects, or disallow code objects to have named slots. The first option was unsatisfactory because it broke the spirit of object-oriented programming. It would require the programmer to use types explicitly, making code less extensible and less reusable. The second option also had its downside: the language became less homoiconic, because code objects cannot have named slots, whereas evaluated objects can.

I opted for the second option, because it seemed cleaner and easier. Named slots are still denoted the same way, but they are nameless in code objects. The upside is that Fuga stopped having specialized syntax for slot naming, and methods can be safely called on objects.

4.4 Lexical Scoping

Variable scoping was a difficult issue, because the languages on which Fuga is based all behave differently in this regard. In Io, functions can be either lexically scoped (Io blocks) or scoped by receiver (Io methods). In Self, functions are scoped by receiver. In most functional languages, functions are lexically scoped, although the original Lisp was dynamically scoped.

The main advantage to scoping like Self and Io is that the receiver's slots are easier to access when a function is scoped using the current object. The programmer doesn't have to say `self` or `this` before accessing the receiver's slots -- they can just say the slot's names. This is convenient, but Fuga's functional nature pushed me heavily towards lexical scoping.

I tried to compromise by mixing lexical scoping with scope by receiver, by creating an object that inherited from both the lexical scope and the receiver. This was a terrible idea: it was difficult to implement, and it was difficult for the programmer to predict how the different scoping rules would interact.

In the end I decided to stick with lexical scoping. The simplicity of this approach makes up for the inconvenience of typing `self` occasionally.

4.5 First-class Thunks

Much of Io's power comes from its metaprogramming capabilities. In Io, this is accomplished by implicitly passing an object, `call`, into all functions. This object contains data and metadata on the message that was sent to invoke the function, including the code objects of the arguments. Using `call`, Io functions can conditionally evaluate their arguments, or even replace the function call with something more elaborate.

The downside to the `call` object, and the reason why it isn't included in Fuga, is that it captures too much information about the calling method and stores it in the current lexical scope. In a language that uses receiver scope by default, such as Io, this isn't usually a problem. In a lexically scoped language like Fuga, this makes it impossible to know when certain data is no longer being used. This is known as a "space leak" -- the memory isn't going to be used, but it sticks around for much longer than necessary because there is a reference to it in some lexical scope somewhere. This causes programs to run out of memory and become slow.

To circumvent this, I came up with a different way to implement metaprogramming in Fuga. Rather than pass a `call` object, pass the arguments as a thunk. Then, make it possible for methods to turn thunks into first-class thunks. Internally, a thunk is represented as a code object combined with its evaluation environment. A first-class thunk splits the thunk apart, and gives the programmer direct access to the code object and the environment.

Using first-class thunks, it becomes possible to conditionally evaluate a method's arguments, and it becomes possible to access the caller's environment. It is also possible to modify the code objects, but only to a limited degree. As an example of first-class thunks, `twice` performs an action twice, and `unless` only performs an action if the condition is false:

```
twice(~action) {
    action eval
    action eval
}

unless(condition, ~action) {
    if(not(condition)
        ~action
    )
}
```

A little bit of built-in support is given to make this feature efficient, but as long as lazy evaluation is possible, first-class thunks are easy to implement, and powerful. They give Fuga programmers the power of Io's `call` object, without the main drawback.

First-class thunks can still cause space leaks if misused. However, the potential for misuse is

reduced because first-class thunks are only passed to functions that request them. By contrast, in Io every function is passed the `call` object, so lexical scoping would cause space leaks more often.

4.6 Project Requirements and Timeline

While the design of Fuga evolved as different issues became apparent, the project requirements were established early on. The requirements were:

- There will be two deliverables: the Fuga specification, and the Fuga interpreter.
- The specification will describe the language formally, and describe any other details that conforming implementations must adhere to.
- The interpreter must conform to the specification.
- The interpreter must work both as a standalone interpreter and as a library that enables scripting support for other applications.

In the proposal for this project, I had outlined three stages for the development of Fuga, and a timeline for each stage. The stages were as follows:

- Initial Development: During this stage, the specification and implementation were both to be worked on until basic project requirements were met, with the bare minimum functionality available. This stage was to end on February 8, 2011.
- Iterative Development: During this stage, the specification and implementation were to be modified iteratively. During each iteration, the project requirements would be broken, significant changes or additions were to be made, and then project requirements were once again to be met. This stage was to end on March 1, 2011.
- Incremental Improvement: During this stage, the specification is frozen, and only small changes to the implementation are allowed. In particular, only changes that would not break compatibility with the specification were allowed. This stage was to end on March 15, 2011.

The timeline seemed reasonable at the time, but it turned out to be a little naive in practice. The initial development took much longer than expected, and the last stage remains an ongoing process, as bugs are found and fixed. The specification has also been updated less frequently than the timeline suggests, and it was only truly finished near the end of the project (late April). So, the implementation and the specification rarely matched, even in between iterations.

4.7 Specification

The Fuga Specification (attached as Appendix A) defines the syntax and semantics of Fuga, and any other details that an implementation of Fuga must adhere to. In designing the specification, I looked at de facto specifications for existing programming languages, in order to determine how to organize the Fuga Specification.

The Io Programming Guide defines the syntax of Io, and then the semantics [Dekorte 2010]. The Python Language Reference defines the tokens of Python, then the semantics, and then goes into the various syntactic forms, defining the full grammar of Python only at the end [Python Software Foundation 2011]. The Java language specification is organized in a similar way [Gosling, et al. 2005].

I decided to copy Io's approach, because Fuga and Io are similar, and because Python's approach is more suited for a language with a lot of syntactic irregularities. Fuga has very few irregularities. Thus, the specification explains all of the syntax, followed by all of the semantics. It's important to note that what would be considered syntax in Python is actually sometimes considered pure semantics in Fuga. For example, Python's "if" statement introduces its own syntax. In Fuga, "if" is built out of existing syntax (and semantics). So while the Fuga specification is organized in the style of the Io Programming Guide, its contents are similar to the Python Language Reference, and the Java Language Specification.

The most interesting aspect of working on the specification was that as I worked on the specification, it became clear whenever parts of Fuga were too complicated. As a result, I ended up simplifying the language a lot. For example, Fuga originally was going to have complicated user-defined operator system, with user-defined operator precedences. This was so cumbersome to explain in the specficiation, that I just removed it from the language. The resulting language is easier to, understand, and implement.

5. LIMITATIONS AND RECOMMENDATIONS

Like all evolving programming languages, Fuga is incomplete. There are many ways to extend Fuga, many other avenues to explore within the "homoiconic object-oriented programming language" genre, and many implementation details to improve on. Naturally, Fuga will adopt some of these in the future.

5.1 Parallelism and Actors

Fuga has no parallelism. Computers are increasingly parallel, but Fuga has no way to take

advantage of this parallelism. There are many ways to extend Fuga to support parallel execution, but the most in keeping with Fuga's roots is through actors.

In Io, messages can be sent asynchronously, and futures can be used to store the result of a message sent asynchronously [Dekorte 2010]. This is used to implement the "actor" paradigm for concurrency, where "actors" receive and send messages asynchronously. In this case, the actors are Io objects.

Implementing this feature for Fuga would be a significant investment, but parallelism is probably here to stay, so it would be a worthwhile investment. Plus, it is not too difficult to implement this feature in a non-parallel way, and only later figure out the difficult task of parallelization. This seems like a likely feature of Fuga in the future.

5.2 Security

One of Fuga's use cases is as a component of a larger application, to add scripting support to any application. One downside to this is that it opens the host program up to any vulnerabilities in Fuga, and gives the programmers of the host program a greater chance to hang themselves.

As it stands now, Fuga allows the user to do almost anything. A savvy hacker could probably bring down any application that uses Fuga for scripting support. Worse, a hacker might be able to manipulate a badly designed host program to reveal confidential information.

One source of insecurity in Fuga's current implementation is that its module system gives all modules the ability to modify all other modules, and the ability to modify imporant shared objects (`Object`, for example). This problem can be mostly eliminated by introducing the concept of privileged vs. unprivileged code. This is a basic form of security, wherein the privileged code has the ability to do everything, but the unprivileged code must ask for permission before modifying other modules or global objects, or accessing other restricted features.

Another possible security model involves "capabilities". A capability is an unforgeable key that gives access to these restricted features. A simple example of capability-based security is to present two different versions of the same object to different code -- code with the capability will see the restricted features, while code without it will not.

Any security model added to Fuga has large implications for the design of the language, and its applicability. Before committing to a particular security model, it is important to research and experiment with different designs.

5.3 Efficient Objects

Fuga's current implementation uses a very naive and inefficient data structure to represent objects: the associative array. Most operations on this data structure run in linear time. By contrast, there are data structures which would provide the same functionality more efficiently. For example, a hash table could produce a slot lookup in amortized constant time, and a balanced binary tree could produce a slot lookup in logarithmic time. This is just a matter of programming the interpreter to use these more efficient data structures. This should present no challenge.

A far more interesting research opportunity is to try to combine delegation (i.e. prototyping) with slot lookup. As far as I know, there are no data structures that efficiently capture the necessary semantics, and it is unknown what kind of complexity such a lookup would have. The difficulty here is that changes made to a prototype must be reflected in the derived objects.

Another possibility is to provide some form of caching to go with delegation. Again, the problem is that when a prototype is changed, all relevant caches must be updated. This is not unlike the difference between delegation and concatenation.

5.4 Finer Garbage Collection

The garbage collector is the part of the interpreter that keeps track of objects that are no longer used. Unused object are "freed" so that there is more memory available for new objects. The garbage collector handles all of the messy details regarding memory allocation so that the programmer can focus on more important details.

Currently, Fuga has a simplistic garbage collector. It works, but it is slow, and when the garbage collector is running, nothing else can happen until the garbage collector is finished.

There are two improvements to the garbage collector that I have in mind. First, generational garbage collection would improve its efficiency by collecting younger objects more frequently. This makes sense because most objects have short lifespans, so focusing on the younger objects means we can collect garbage more quickly.

The second improvement is to add a write barrier to the garbage collector. A write barrier enables us to implement incremental garbage collection, so we can start the collector and pause it halfway through a garbage collection run without disrupting it. This is quite difficult to implement correctly, especially to retrofit it onto an existing codebase, but it would enable the garbage collector to be used in more realtime, interactive settings, because there won't be the need for long pauses when the garbage collector is working.

5.5 Standard Library

There are many ways in which Fuga could be improved. All of the sections above suggest improvements to the language, or to the interpreter. Another way to improve Fuga would be to implement a more comprehensive standard library, so that Fuga is easier to use and applicable in more situations. This is the easiest way to improve Fuga, but the downside to expanding the standard library is that it reduces Fuga's embeddability and portability.

6. CONCLUSION

Fuga is the first homoiconic object-oriented programming language. Fuga was an attempt to combine homoiconic programming language design with the object-oriented paradigm, and in this regard Fuga is a success. There were difficulties regarding the integration of these two styles, but they were slowly eliminated, either by exploring a different solution (for example, the introduction of first-class thunks), or reaching a compromise (for example, code slots are nameless). The result is a simple, powerful language.

On a personal level, this project has also been a success. I enjoyed watching the language evolve, and watching the interpreter slowly become what it is today. Nothing quite beats the triumphant feeling of working on a feature for hours, and then finally making it work.

This project has taught me that it's best to approach a problem in multiple directions at once. In this case, the language was designed while it was implemented, and while it was being specified. Both the implementation work and the specification work yielded useful insights into the best direction for the language's design. These correspond to the two main approaches to computer science and to software development: the experimental approach, and the theoretical approach. In my future projects, it will be a good idea to use both approaches to find and solve problems.

This project accomplished everything it set out to do. Fuga is a useful programming language, both in practical and theoretical ways. There is still much work to do to improve Fuga, and there is much room left for other explorations into homoiconic object-oriented programming language design and implementation, but I feel that this is already a valuable contribution to the field.

REFERENCES

ASTUDILLO, H. R. AND SHILLING, J. J. 1993. Alternative Object Organizations using Prototypes, Delegation and Split Objects. *Technical Report GIT-CC-93/31.* College of Computing, Georgia Institute of Technology, Atlanta, GA.

DEKORTE, S. 2005. Io: a small programming language. In *Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* San Diego, CA, October 2005. ACM, New York, NY, 166-167. DOI= http://doi.acm.org/10.1145/1094855.1094916

DEKORTE, S. 2010. Io Programming Guide. http://www.iolanguage.com/scm/io/docs/IoGuide.html

GOSLING, J. ET AL. 2005. The Java Language Specification. http://java.sun.com/docs/books/jls/third_edition /html/j3TOC.html

IERUSAMLICHY, R. ET AL. 2007. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. San Diego, CA, June 2007. ACM, New York, NY, 2-1 - 2-26. DOI= http://doi.acm.org/10.1145 /1238844.1238846

INCE, D.C. AND HEKMATPOUR, S. 1987. Software prototyping -- progress and prospects. In *Information and Software Technology, Volume 29, Issue 1*. January 1987. 8-14. DOI= 10.1016/0950-5849(87)90014-0.

KAY, A. C. 1996. The early history of Smalltalk. In *History of Programming Languages II*, T. J. BERGIN and R. G. GIBSON, Eds. ACM, New York, NY, 511-598. DOI= http://doi.acm.org/10.1145/234286.1057828

KIENLE, H. M. 2010. It's About Time to Take JavaScript (More) Seriously. In *IEEE Software 27, 4*. May 2010. 60-62. DOI= http://dx.doi.org/10.1109/MS.2010.764

PYTHON SOFTWARE FOUNDATION. 2011. The Python Language Reference. http://docs.python.org/py3k/reference /index.html

STEELE, G. L. 1998. Growing a language. In *Addendum To the 1998 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*. Vancouver, BC, Canada, October 1998. J. HAUNGS, Ed. ACM, New York, NY. DOI= http://doi.acm.org/10.1145/346852.346922

STEELE, G. L. AND GABRIEL, R. P. 1993. The evolution of Lisp. In *Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages*. Cambridge, MA, April 1993. ACM, New York, NY, 231-270. DOI= http://doi.acm.org /10.1145/154766.155373

UNGAR, D. AND SMITH, R. B. 2007. Self. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. San Diego, CA, June 2007. HOPL III. ACM, New York, NY, 9-1 - 9-50. DOI= http://doi.acm.org/10.1145 /1238844.1238853

Appendix A: The Fuga Specification

A1. INTRODUCTION

This document defines the Fuga programming language, version 1.0. It describes the syntax and the semantics of Fuga, and any other details that an implementation of Fuga must conform to.

A2. LEXICAL ANALYSIS

Lexical analysis is the first step in the interpretation of Fuga code. The *lexer* reads a string or a stream of code, and emits a stream of *tokens*.

A2.1 Character Encoding

A Fuga interpreter may support either ASCII or UTF8 natively. At the moment, Fuga only supports the ASCII character set, but the groundwork has been laid for native UTF8 support.

A2.2 Ignored Elements

All whitespace except newlines are ignored, as are comments and escaped newlines. A comment begins with a pound sign `#` and ends with a newline. The newline is not part of the comment. For example:

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

An escaped newline is a newline preceded immediately by an escape sign. Thus, lines can be joined:

```
this is all \
   one line
```

A comment does not end at an escaped newline:

```
# This is \
  all one \
  comment.
```

A2.3 Integers

At this time, Fuga supports only decimal integers. An integer is any combination of the digits `0` through `9`. For example:

```
1
123
```

```
0
007
```

The digits cannot be followed (or preceded) immediately by letters. If they are, the token is a name, not an integer.

A2.4 Names

A name is any combination of the letters (`A` through `Z`, `a` through `z`), digits (`0` through `9`), underscore `_`, and the punctuation `?` and `!`. For example:

```
a
foo
a?
hello!
_
1st
_?!
```

A name must contain at least one letter, or an underscore. Names cannot begin with punctuation `?` or `!`.

Names can also be escaped operators, or suppressed operators, as explained below.

A2.5 Operators

An operator is any combination of the operator characters:

```
+ - * / % ^ =
? ! ; : . ` '
@ ~ $ & | < >
```

For example,

```
+
==
<=
!?-
:=
```

An operator cannot be composed entirely of punctuation `?` and `!`, and an operator cannot start with two colons `::`.

If an operator is preceded by an escape, it is actually treated as a name token. These are known as escaped operators. For example, these are actually considered name tokens:

```
\+
```

```
\==
\<=
\!?-
```

The escape is consumed along with the operator characters, but the escape is not part of the name.

Likewise, if an operator is immediately proceeded by a left parenthesis, it is considered a name. These are called suppressed operators. For example, the operator here is actually a name:

```
10 +(20)
==()
```

A2.6 Symbols

A symbol is a colon followed by a name or an escaped operator. For example:

```
:a
:foo
:1st
:\++
:\=
:\:=
```

A2.7 Documentation

A documentation starts with two colons `::` and run till the end of the line. For example:

```
:: This is a documentation.
```

The newline is considered part of the documentation, but is not consumed. When the newline isn't present (say, at the end of the file), it is added.

Although two consecutive documentation tokens are considered separate tokens, they will be joined in parsing, so it is useful to think of them as one long documentation.

```
:: Line 1
:: Line 2
```

An escaped newline is ignored. For example:

```
:: This is one long \
documentation.
```

If there is leading space after the colons, a single space is ignored. So, the following two documentations lines are the same:

```
::Hi
:: Hi
```

But the following two lines are different:

```
:: Hi
::  Hi
```

A2.8 Strings

Strings are enclosed in double quotes. For example,

```
"Foo bar."
```

Strings need not end at the newline. Newlines are included in the string. Strings can have also have escaped newlines, in which case the escaped newline is ignored.

```
# With newline:
"Line 1,
Line 2."

# Without:
"This is \
one line."
```

Strings also allow for escaped characters. The allowed escaped characters are:

```
   |      Value
--------------------
\" |         "
\\ |         \
\n |     newline
\r | carriage return
\t |       tab
```

A2.9 Special Tokens

Apart from the tokens above, Fuga has a set of single character tokens.

```
Token Type |   Character
---------------------------
LPAREN     |       (
RPAREN     |       )
LCURLY     |       {
RCURLY     |       }
LBRACKET   |       [
RBRACKET   |       ]
SEPARATOR  |  , or newline
```

```
EOF         |  end of file
```

EOF is the token produced by the lexer once all the characters have been consumed. ERROR is the token produced by the lexer if a character is not recognized.

A3. SYNTAX

A *parser* takes a stream of tokens, and produces objects. These objects are known as *code objects* to differentiate them from other kinds of objects, but they are normal Fuga objects. This section describes this process.

This is the entire grammar of Fuga:

```
object ::= LPAREN block RPAREN
block  ::= slot SEPARATOR block
         |      SEPARATOR block
         | slot
         |
slot   ::= DOCUMENTATION slot
         | method
         | expr
expr   ::= expr OPERATOR part
         | part
part   ::= OPERATOR part
         | root msgs
root   ::= LBRACKET expr RBRACKET
         | object
         | INTEGER
         | STRING
         | SYMBOL
         | msg
msgs   ::= msg msgs
         |
msg    ::= NAME object
         | NAME
method ::= part LCURLY block RCURLY
```

A3.1 Operator Expressions

Infix operator expressions are left-associative. So `1 + 2 + 3` is parsed as `[1 + 2] + 3`. This applies even with different operators, so `a = 2 + 2` is parsed as `[a = 2] + 2`, and `a * a + b * b` is parsed as `[[a * a] + b] * b`.

We can use square brackets to have the parser evaluate operator expressions differently. So, to "fix" `a = 2 + 2`, we would do `a = [2 + 2]`.

Operator expressions are syntactic sugar. `a op b` is interpreted as `op(a, b)`, where `op` is an operator. Likewise `op a` is interpreted as `op(a)`.

A3.2 Desugaring Methods

Methods, as defined in the grammar above, are also syntactic sugar. Methods have two parts: the method signature and method body. A method defined as

```
SIGNATURE { BODY }
```

becomes

```
def(SIGNATURE, BODY)
```

For example,

```
foo(bar) { baz }
foo { bar, baz }
foo {}
```

become

```
def(foo(bar), baz)
def(foo, bar, baz)
def(foo)
```

A3.3 Canonical Grammar

With all of the syntactic sugar removed, the *canonical grammar* of Fuga is as follows:

```
object ::= LPAREN block RPAREN
block  ::= slot SEPARATOR block
         |      SEPARATOR block
         | slot
         |
slot   ::= DOCUMENTATION slot
         | part
part   ::= root msgs
root   ::= INTEGER
         | SYMBOL
         | STRING
         | object
         | msg
msgs   ::= msg msgs
         |
msg    ::= NAME object
         | NAME
```

A3.4 Conversion into Code Objects

Integer, string, and symbol tokens are converted directly into corresponding integer, string, and

symbol primitives.

Messages are converted into message primitives -- the name part becomes the message name, and the object part becomes the message's arguments. Naturally, if no object is present, the message has no arguents.

Objects are created slot by slot. The documentation becomes part of the slot. Multiple documentation lines are concatenated (preserving the newline). So, for example, the object `(a, b, c)` has three slots, `a`, `b`, and `c`. The object

```
(a, b
 :: Hello
 :: world!
 c)
```

also has three slots, with the same values, but the third slot has some associated documentation. Note that the documentation is associated with the slot, not with the value.

The "part" syntax is converted into an Expr object. The slots of the Expr object are filled with the root of the "part", and with the messages. So, the "part" `a b c` has three slots, `a`, `b`, and `c`, and is an Expr object.

A4. OBJECTS

All values are objects. Objects respond to messages, producing some desired behavior.

A4.1 Prototypes

Except for `Object`, all objects have a prototype. When an object receives a message, it first looks in the object's slots for a possible answer. If no object has a slot with the given name, the message is forwarded to that object's prototype. This way, objects need not specify all of their behavior -- similar objects can inherit from a common prototype.

Typically, a prototype will also have a prototype, creating a prototype chain. An object cannot be in its own prototype chain. This implies that all prototype chains lead to `Object`. Although objects can only have one direct prototype. Using prototype chains, objects can have any number of indirect prototypes.

In this document, when we refer to an object's prototype, we mean an object's direct prototype. When we refer to an object's prototypes, we mean all objects in that object's prototype chain. The process of creating an object with a given prototype is called cloning.

A4.2 Slots

Objects are made up of slots. To facilitate slot manipulation and object evaluation, slots in an object are ordered. The first slot has index 0, the second slot has index 1, and so on. Slot order is in the order of slot creation, and there is no way to modify the order of slots.

All slots have a value. This is a pointer to another object.

Some slots have names. These correspond to methods or member variables in other object-oriented languages. No object can have two slots with the same name.

Some slots also have documentation. Typically, documentation is a string that explains the purpose of a slot. Unlike other languages, the documentation is associated with the slot, not with the slot's value.

There are many slot operations in Fuga. The simplest are:

- **hasRaw**: does an object contain a slot with a given index, or a given name?
- **getRaw**: return the value of the slot with a given index, or a given name.
- **has**: does an object, or one of the object's prototypes, contain a slot with the given name?
- **get**: return the value of the slot with a given name in an object. If that object contains no such slot, look in the object's prototype, and so on.
- **set**: associate a slot with a given index or a given name, with the given value. If no such slot exists, it is created.
- **del**: delete a slot with a given index or a given name. All slots that occur after the deleted slot are shifted down.

There are also operations to manipulate slot documentation. For more information, read about `Object` in section TODO.

The **has** and **get** operations will also accept indexes, but they will not delegate these indexes to an object's prototype. So, for indices, **has** and **get** function just like **hasRaw** and **getRaw**.

An object's length is the number of slots it contains.

As was explained above, all slots in an object have an index. For most purposes, when an index is used, a negative index can also be used. A negative index indicates a slot starting from the last. So, the last slot has index -1, the penultimate slot has index -2, and so on.

A4.3 Primitives

Some objects contain primitive data. These are primitives. Fuga's core primitives are integers, strings, symbols, and messages.

Integers inherit from `Int`. Integers are immutable. Currently, only signed 64-bit integers are supported. In the future, unbounded integers will be supported.

Strings inherit from `String`. Strings are an immutable sequence of characters. Currently, only ASCII strings are supported. In the future, UTF-8, and possibly other encodings, will be supported.

Symbols inherit from `Symbol`. Symbols are also an immutable sequence of characters. Symbol data must follow the lexical rules for name tokens, or for operator tokens. (It is unnecessary and illegal to escape operator symbol data.)

Messages inherit from `Msg`. Messages have two components: the message name, and message arguments. The arguments are represented by the slots of the message object. The name is not. Thus the message name is considered primitive data. The message name is a symbol primitive.

A4.4 Vanilla Objects

We refer to non-primitive objects as vanilla objects.

A module represents a Fuga file. All modules inherit from `Prelude`, which means that any slot defined in `Prelude` is globally accessible in all modules.

A boolean represents a truth value. Booleans inherit from `Bool`. There are only two booleans: `true` for truth, and `false` for falsehood.

`nil` is an object that represents "no value". `nil` is only meant to be used as the result of an action -- never use `nil` to represent an empty value, or to represent `false`. That would cause errors. A slot that evaluates to `nil` is ignored, for the most part.

An expression is a vanilla object that inherits from `Expr`. Expressions are code objects that represent message sending. The first slot in an expression is the root of that expression. Every other slot is a message to be sent to the result of the previous slots.

A thunk is a vanilla object that represents an unfinished computation. Thunks inherit from `Thunk`. Thunks have a `code` slot, which represents the code object to evaluate, and a `scope` slot, which represents the scope in which to evaluate the code object. Thunks are used for metaprogramming. See the section on first-class thunks.

A4.5 Exception Objects

Fuga uses the exception model for error handling.

A5. EXECUTION MODEL

A5.1 Evaluation

In order to execute code, we evaluate it. There are two kinds of evaluation, and they are used in different contexts.

Irreflexive evaluation is the default, and it's used for all kinds of objects. Whenever we talk of

evaluation, we mean irreflexive evaluation, unless otherwise specified. Evaluation corresponds to the sending of the `eval` message, which takes two arguments: the current lexical scope, and the current message receiver. `eval` is defined in such a way that:

- Integer, string, and symbol primitives evaluate to themselves.
- Messages are sent to the current message receiver. The argument is sent wrapped in a thunk with the current lexical scope.
- Expressions evaluate their first component in the current lexical scope with the current receiver. They set the current message receiver to the result of that evaluation, and then they do it again with the second component, etc.
- For other obects, we evaluate slot by slot, appending the evaluated slot value to the result. Slot values that evaluate to nil are ignored. Also, slots are evaluated in a scope in which `_this` refers to the result of the evaluation, and `_doc` refers to the current slot's documentation. This allows functions such as `=` and `def` to work.

In terms of Fuga, this is roughly:

```
Int     eval(scope, receiver) { self }
String  eval(scope, receiver) { self }
Symbol  eval(scope, receiver) { self }

Message eval(scope, receiver) {
   receiver send(self name, thunk(self args, scope))
}

Expr eval(scope, receiver) {
    receiver = var(receiver)
    for(slot, self) do(
        receiver := slot eval(scope, receiver)
    )
    receiver
}

Object eval(scope, receiver) {
    result = Object clone
    scope  = scope clone(this = result)
    for(index, slot, self) do(
        scope thisDoc = self getDoc(index)
        value = slot eval(scope, scope)
        value nil? ifFalse(
            result append(value)
        )
    )
```

```
    result
}
```

Reflexive evaluation is used when evaluating a file, when evaluating the arguments to the `do` method, and when evaluating a method's body. Reflexive evaluation is only used on vanilla objects. In reflexive evaluation, `_this` refers to the scope passed in as a parameter. Only the last evaluated slot is returned (whether or not it is nil), but typically the important result is that the scope is modified. It is called reflexiveevaluation because named slots can be accessed in later slots. For example, `(a = 20, b = 30, a + b)` will evaluate to `50`.

Reflexive evaluation in terms of Fuga:

```
Object evalIn(scope) {
    scope = scope clone(this = scope)
    value = var
    for(index, slot, self) do(
        scope thisDoc = self getDoc(index)
        value := slot eval(scope, scope)
    )
    value
}
```

A5.2 Methods

Methods are functions that are part of an object. All functions in Fuga are part of an object, so all functions are methods. Whenever a message is sent to an object, Fuga looks for a slot that corresponds to the message name (using the **get** operation was explained in section TODO). If the slot's value is a method, the method is called automatically, whether or not arguments were passed. To avoid activating the method automatically, use **get** instead of sending a message directly. For example:

```
>>> (1, 2, 3) len
3
>>> (1, 2, 3) get(:len)
method(...)
```

When sent as a message, `len` is called automatically. Using `get`, you can get the `len` method. You can also use the `.` operator for a similar effect:

```
>>> (1, 2, 3).len
method(...)
```

Some methods are primitive. These methods are implemented in C, and they're used to

implement basic functionality on which other methods are built.

Non-primitive methods are created using the `method` method, the `def` method, or the method syntactic sugar (which reduces to `def`). The following are equivalent, assuming `foo` hasn't already been defined.

```
foo = method((bar), baz)
def(foo(bar), baz)
foo(bar) { baz }
```

Non-primitive methods have three components: `scope`, `args`, and `body`. The `scope` is the lexical scope in which the method was created. `args` is a sequence of patterns (see section TODO on pattern matching below). `body` is a sequence of method bodies. Each pattern in `args` corresponds to a `body`. Using `foo` defined above, we have:

```
>>> .foo scope
(...)
>>> .foo args
((bar))
>>> .foo body
(baz)
```

If the method already exists, `def` (and the method syntax) simply add to `args` and `body`, rather than creating a new method. For example, the following creates a two-argument version of `foo`:

```
foo(bar, baz) { bif }
```

We can see the changes to `.foo args` and `.foo body`:

```
>>> .foo args
((bar), (bar, baz))
>>> .foo body
(baz, bif)
```

A5.3 Exceptions

Fuga uses exceptions to signal errors, or to signal exceptional values. When an exception occurs, all methods are exited until an exception handler is found. If there is no exception handler, the program halts with an error.

All exception values inherit from `Exception`. `Exception` defines the `raise` method, which raises the current exception, attaching an error message if one is given. For example, we raise a `ValueError` exception:

```
ValueError raise("value is out of bounds")
```

To catch an exception, use `try`:

```
try(riskyCode,
    ValueError, exceptionHandler1,
    TypeError, exceptionHandler2)
```

The first argument to `try` is the possibly exceptional value. Then, there are (prototype, exception handler) pairs. If the exception inherits from the given prototye, the corresponding exception handler is called. The exception handler is evaluated in the current lexical scope, with `exception` set to the exception value. For example:

```
>>> try(1 / 0
...     ValueError, exception msg)
"Division by zero."
```

If there are no matching prototypes, the exception is unhandled by this particular `try` call. The exception might be handled further up the call stack.

A5.4 Pattern Matching

Method arguments and the Prelude `match` method use pattern matching to branch and capture data. For example, we could define the factorial function recursively in three different ways.

The traditional way uses `if`:

```
factorial(n) {
    if(n == 0, 1
       factorial(n - 1) * n)
}
```

Instead, we could use pattern matching, with `match`:

```
factorial(n) {
    match(n
        0, 1
        _, factorial(n - 1) * n
    )
}
```

Alternatively, we can use the built-in pattern matching in methods. Again, there are many ways to declare a method. Using the method syntax:

```
factorial(0) { 1 }
factorial(n) { n * factorial(n - 1) }
```

Using `def`:

```
def(factorial(0), 1)
def(factorial(n), n * factorial(n - 1))
```

Using `method`:

```
factorial = method(
    (0), 1
    (n), n * factorial(n - 1)
)
```

To match patterns to values, we define a `match` method on code objects. The `match` method returns an object with all the captured variables, or it raises a `MatchError`. The `match` method is defined more or less as follows:

```
Int    match(n) { if(self == n, (), MatchError raise) }
String match(s) { if(self == s, (), MatchError raise) }
Symbol match(s) { if(self == s, (), MatchError raise) }
Msg match(value) {
    result = ()
    result set(self name, value)
    result
}
Object match(value) {
    if(self len != value len, MatchError raise)
    result = ()
    for(i, slot, self) do(
        result update(slot match(value))
    )
    result
}
```

In short, integer, string, and symbol primitives match only themselves. Message primitives match everything, capturing the matched variable as the method name. Finally, vanilla objects attempt to match each slot to the target, combining all match results.

There is one message matching pattern that isn't discussed until section TODO on first-class thunks.

A5.5 Lazy Evalution

By default, code objects are evaluated whenever they are encountered. Using lazy evaluation, it's possible to delay the evaluation of code until it is absolutely necessary. To do this, use the `lazy` function:

```
>>> a = lazy(print("Hello"))
>>> a
Hello
>>> a
```

As can be seen, a lazy value is only evaluated once. The value is memorized (or memoized) so that it is unnecessary to evaluate the lazy value over and over again.

Lazy evaluation can be used to create infinite structures. Here is an infinite linked list:

```
>>> count(n) {
...     lazy(
...         (value = n
...          next  = count(n+1))
...     )
... }
>>> a = count(0)
>>> a value
0
>>> a next value
1
>>> a next next value
2
```

Be careful when using infinite structures. Printing a would take an infinite amount of time.

If an unhandled exception occurs while a lazy value is being evaluated, the value is not considered evaluated. Attempting to evaluate the value again may result in the same exception, unless whatever was causing the exception has disappeared. For example:

```
>>> a = lazy(b)
>>> a
EXCEPTION:
    get: no slot named 'b'
>>> a
EXCEPTION:
    get: no slot named 'b'
>>> b = 10
>>> a
10
```

A5.6 First-class Thunks

Internally, method arguments are passed as lazy values. In order to give Fuga a metaprogramming ability, first-class thunks were designed to allow the programmer to break apart and manipulate lazy values.

As mentioned in section TODO on pattern matching, there is an additional message matchin

pattern that wasn't discussed. It is used to capture a first-class thunk. The pattern is `~name`: tilde, followed by a message name. This will match any lazy value -- in practice, this will match any argument. A thunk object inherits from `Thunk` and has two components: `code` represents the unevaluated code object; and `scope` represents the lexical scope in which to evaluate the object.

We can define many useful methods using first-class thunks. For example, `quote` returns an unevaluated code object:

```
>>> quote(~t) { t code }
>>> quote(10)
10
>>> quote(foo)
foo
>>> quote((a, b))
(a, b)
```

This corresponds to the backtick operator (`` `) ``, defined in `Prelude` (section TODO).

The tilde operator (`~`) is also used to pass thunk objects to other methods that want them. This enables first-class thunk recursion, or fancy functions such as `swap`, which swaps two values:

```
>>> swap(~a, ~b) {
...     t = ~a
...     ~a := ~b
...     ~b := t
... }
>>> x = 10, y = 20
>>> x, y
10
20
>>> swap(x, y)
>>> x, y
20
10
```

To see all of the operations available for thunk objects, see section TODO, on the `Thunk` object.

A5.7 Modules

Large Fuga programs are organized into many files. Each Fuga file is a separate Fuga module. A directory of modules is a package. Fuga modules have the extension ".fg", so Fuga knows where to look when importing a module.

To import a module, use `import`. `import` looks in the current module's directory (or in the current directory, if called from the interactive interpreter). `import` takes a single argument: the module's name. For example, to import "list.fg", use:

```
>>> import(List)
```

The file, "list.fg", is parsed. Then, it is evaluated reflexively in a scope that inherits directly from `Prelude`. The module is the scope. Thus, all modules inherit from `Prelude`, and any methods or data defined in `Prelude` are accessible from the module.

Once "list.fg" is imported, it is cached, so that "list.fg" is only imported once, no matter how many times `import(List)` is called. Finally, the importing scope receives the imported module, setting it to the slot `List`. Then, "list.fg" can be referred to as `List` forevermore. For example:

```
>>> import(List)
>>> List list(10, 20, 30)
list(10, 20, 30)
```

Support for packages (directories of modules) is still tentative, as of version 1.0. Passing an expression (i.e., a sequence of messages) to `import` forces `import` to look in subdirectories. So, `import(Foo Bar Baz)` would look for "foo/bar/baz.fg" on Unix, and "foo\bar\baz.fg" on Windows. The imported module is set to the given expression. In this case, if there is no `Foo Bar` object, it is created in order to hold `Foo Bar Baz`. If there is a `Foo Bar` object, `Foo Bar Baz` is added to it. The intermediate objects are not descendent of `Prelude` -- they inherit from `Object` directly.

A6. OBJECT REFERENCE

This section explains some of the built-in objects that are part of Fuga 1.0. This section is not comprehensive, for it covers only the most important objects and the most important methods and slots within those objects. For full documentation, use the `help` method from the interactive interpreter (section TODO).

A6.1 Object

`Object` is the root prototype for all other objects. `Object` is the only object that does not have a prototype. Methods and data in `Object` are ubiquitous to all objects. There is the danger that an object will unwittingly override one of `Object`'s important methods.

A6.1.1 Object hasRaw

Determine whether self contains a slot with the given name or index, without looking in the prototype.

```
Object hasRaw(name or index) { Bool }
```

- Params:
  - `name`: can be a string, a symbol, or an empty message, to test for a named slot. Can be a positive integer, to test for an index.
- Return: `true` if `name` is one of `self`'s slot names, or a valid index for `self`, `false` otherwise.
- Exceptions:
  - `TypeError` if `name` is not a string, symbol, message, or index.

A6.1.2 Object has

Determine whether self contains a slot with the given name or index. Look in the prototype if self doesn't contain the given name. (Do not look in the prototype if an index was given.)

```
Object has(name or index) { Bool }
```

- Params:
  - `name`: can be a string, a symbol, or an empty message, to test for a named slot. Can be a non-negative integer, to test for an index.
- Return: `true` if `name` is one of `self`'s slot names, or a valid index for `self`, `false` otherwise.
- Exceptions:
  - `TypeError` if `name` is not a string, symbol, message, or non-negative integer.

A6.1.3 Object getRaw

Look for a slot with the given name or index, and get its value. Do not look in the protoype.

```
Object getRaw(name or index) { slot value }
```

- Params:
  - `name`: can be a string, a symbol, or an empty message, to test for a named slot. Can be a non-negative integer, to test for an index.
- Return: the slot's value, if it was found.
- Exceptions:
  - `TypeError` if `name` is not a string, symbol, message, or non-negative integer.
  - `SlotError` if self does not contain a slot with the given name or index.

A6.1.4 Object get

Look for a slot with the given name or index, and get its value. Do look in the prototype, if the

given name is not in self. (Do not look in the prototype if the an index was given.)

```
Object get(name or index) { slot value }
```

- Params:
  - `name`: can be a string, a symbol, or an empty message, to test for a named slot. Can be a non-negative integer, to test for an index.
- Return: the slot's value, if it was found.
- Exceptions:
  - `TypeError` if `name` is not a string, symbol, message, or non-negative integer.
  - `SlotError` if self does not contain a slot with the given name or index.

A6.1.5 Object set

Create a slot with the given name or index. If the slot already exists in self, update it.

```
Object set(name or index, value) { nil }
```

- Params:
  - `name`: can be a string, a symbol, or an empty message, to test for a named slot. Can be a non-negative integer, to test for an index. If `name` is an index, it can range from 0 to the length of self, inclusive. If `name` is the length of self, the slot is appended.
  - `value`: the value of the slot.
- Return: `nil`.
- Exceptions:
  - `TypeError` if `name` is not a string, symbol, message, or non-negative integer.
  - `ValueError` if `name` is an index that is too large.

A6.1.6 Object append

Create a slot with the given value at the end of self's current slots, increasing self's length.

```
Object append(value) { nil }
```

- Params:
  - `value`: the value of the slot
- Return: `nil`.

A6.1.7 Object len

Return the length of self (i.e., how many slots it has).

```
Object len { Int }
```

- Params: none.
- Return: the number of slots in self.

A6.1.8 Object str

Convert the object into a string. For code objects, this should return a string that, when parsed, is equivalent the original code object.

Beware: `str` will hang on infinite objects, or objects with circular references. This will be improved upon in future versions of Fuga.

```
Object str
```

- Params: none.
- Return: a string that represents self.

A6.1.9 Object eval

Evaluate the object in a given lexical scope, with a given current message receiver. See section A5.1 on evaluation.

```
Object eval(scope, receiver) { value }
```

A6.2 Prelude

`Prelude` is the prototype for all modules. It gives access to all other objects and methods. Thus, `Prelude` has a slot named `Object` that points to `Object`, and so on with all other built-in objects, such as `Int` or `true`.

A6.2.1 Prelude if

Branch based on a truth value. `if` can take two arguments to an unlimited number of arguments. Its most basic form is two arguments:

```
if(condition1, branch1)
```

If `condition1` evaluates to `true`, `branch1` is evaluated. If `condition1` evaluates to `false`, `branch1` is ignored, and `nil` is returned. If `condition1` evaluates to anything else, a `TypeError` is

raised: `if` only accepts booleans.

Adding another argument gives you a branch to evaluate if `condition1` is `false`.

```
if(condition1, branch1
               branchElse)
```

Adding more conditions and branches gives you more conditions to test for in case all previous conditions have failed.

```
if(condition1, branch1
   condition2, branch2
   condition3, branch3
        ...
   conditionN, branchN)
```

You can combine multiple conditions and an else branch:

```
if(condition1, branch1
   condition2, branch2
        ...
   conditionN, branchN
               branchElse)
```

Finally, it's worth noting that `if` can be rewritten as `aif`, `elif`s, `else`:

```
if( condition1
    branch1
) elif (condition2
    branch2
) elif (
  ...
) elif (conditionN
    branchN
) else (
    branchElse
)
```

This works because `nil` has `elif` and `else` methods.

A6.2.2 Prelude print

Print a series of objects to standard output.

```
print(object1, object2, ..., objectN)
```

Each object is converted to a string by calling the `str` method, then the strings are printed one

by one, with spaces in between. A newline is added at the end.

If `print` is called with no arguments, it prints only a newline.

A6.2.4 Prelude import

As explained in section TODO, `import` imports new modules. The imported module is a clone of Prelude.

What wasn't explained is that all new modules have a `Loader` slot that controls how that module imports other modules. `import` is an elaborate hook into the calling module's `Loader`. The `Loader` is necessary in order for modules in different directories to load their local directories. The `Loader` keeps track of the local directory / package.

The `Loader` also keeps track of global directories. On Unix systems, this is set by exporting the FUGAPATH environment variable. FUGAPATH is a string of directories separated by colons (`:`).

A6.2.5 Prelude do

`do` performs a sequence of actions, returning the value of the last action performed. `do` evaluates its arguments reflexively, so named slots are accessible in later slots. For example, `do(a = 10, b = 20, a + b)` returns `30`.

```
do(action1
   action2
   ...
   actionN)
```

- Params:
  - `action`: some value to be evaluated in order.
- Return: the result of `actionN`. If `do` is called with no arguments, `nil`.

A6.2.6 Prelude try

Handle exceptions. As section TODO on exceptions explained, with `try` one supplies a possibly exceptional value, and a series of exception handlers. If an exception occurs, the first prototype

```
try(value
    prototype1, handler1
    prototype2, handler2
          ...
    prototypeN, handlerN)
```

- Params:

  - `value`: the possibly exceptional value
  - `prototype`: exception prototype to test for
  - `handler`: code to execute if matching exception was caught. The exception is accessible through the `exception` variable.
- Return: `value`, if no exception was raised. The result of `handlerI`, if an exception that matches `prototypeI` was raised.
- Exceptions: any exception that `value` raises that isn't handled by one of the handlers.

A6.2.7 Prelude method

Create an anonymous method. The current lexical scope is used as the method's scope. `method` takes a sequence of pairs of method parameters (patterns) to method bodies. When the method is called, the first matching pattern (section TODO) is used, and its corresponding body is evaluated.

```
method(pattern1, body1
       pattern2, body2
            ...
       patternN, bodyN)
```

- Params:
  - `pattern`: the method arguments / parameters / pattern to match.
  - `body`: the corresponding body to evaluate if the pattern matches.
- Return: a method, which inherits from `Method`.

A6.2.8 Prelude def

Create or update a named method. `def` takes a signature and a method body. The signature is used to determine where the method is, and what its parameters are. The method body can be any number of slots: an empty method body corresponds to `nil`, and a method body that is two or more slots long is wrapped in a `do` call. For example:

```
def(foo)           # same as "foo = method((), nil)"
def(foo, bar)      # same as "foo = method((), bar)"
def(foo, bar, baz) # same as "foo = method((), do(bar, baz))"
```

When `def` is used to define a method that already exists, it simply adds a new pattern and body to that method's current definition. This way `def` can be used to define a complex function with many patterns. For example, the fibonacci series:

```
def(fib(0), 0)
def(fib(1), 1)
def(fib(n), fib(n-1) + fib(n-2))
```

`def` is the unsugared version of the method syntax (section TODO). `SIGNATURE { BODY... }` becomes `def(SIGNATURE, BODY...)`.

```
def(signature
    body...)
```

- Params:
  - `signature`: determine's the method's name and arguments.
  - `body`: the method body.
- Return: nil.

A6.2.9 Prelude operators

Infix operator expressions are turned into messages, as was explained in section TODO. This means that an expression such as `a + b` is turned into `+(a, b)`. In many cases, this is undesirable -- we want certain operators to be sent as messages to the left side. For example, we want `a + b` to become `a +(b)`. This is better for some operators because it enables the object-oriented style. For other operators, this would be unnaceptable, because it would limit the applicability of operators. In particular, the `=` and `:=` operators could not exist.

So, in `Prelude`, there are definitions for various operators that turn a `+(a, b)` call into an `a +(b)`. These operators are:

- Arithmetic: `+`, `-`, `*`, `/` (true divison), `//` (floor division), `%`
- Relational: `==`, `<`, `>`, `<=`, `>=`
- Sequences: `++` (concatenation)

These are called canonical operators. Other (non-canonical) operators are explained below.

A6.2.9.1 Prelude =

Declare a named slot, if the left-hand side is only a message name. Otherwise, set a named slot in some object. The right hand side is the value for the new named slot. If the slot already exists, it is updated.

This operator uses the `_this` variable of the current lexical scope to determine where the new operator will reside. If `_doc` is also defined in the current lexical scope, the created slot will use

that as its documentation.

```
name = value
```

- Params:
  - `name`: name or location of the named slot.
  - `value`: value of the slot.
- Return: `nil`

A6.2.9.2 Prelude :=

Update a named slot. This functions just like `=`, but it does not create a new slot, it changes the value of an existing slot with the given name.

```
name := value
```

- Params:
  - `name`: name or location of the named slot.
  - `value`: value of the slot.
- Return: `nil`.

A6.2.10 Prelude help

Provide documentation for a given object or slot.

```
help(object)
help(object slot)
```

- Params:
  - `object`: the location of the object or slot to get documentation for.
- Return: `nil`.