Modelling Cognition

Matthew Hinton[1*] and Preston Pan[2†]
March, 2024

[1]University of Victoria

Abstract

Cognition is a system wherein components of the language are public to the language itself. It is written in pure postfix notation, and removes the prefix characteristics of other similar languages. This enables the Cognition interpreter to be able to manipulate its state based on code read on the fly in a token stream, giving it the property of tokenizing and parsing future characters differently than it does current characters. The result is an unopinionated language that can replicate a wide variety of different syntaxes, allowing Cognition to modify syntax entirely during runtime. This paper investigates the implications of this programming language, which has potential applications in parsing and self-modifying systems.

1 Introduction

Current programming languages operate with parsing requirements. This means that the parser must read ahead of the current token to decide what to do with the current token being passed into the parser. This usually causes the langauge to have a static syntax, due to the parser being too rigid for the AST to be made public. Metaprogramming languages solve this problem by making parts of the AST public, and in particular, concatenative programming languages such as forth, factor, and stem solve a part of this problem by introducing postfix notation for symbols usually known as *words*, which reduce the need for parsing by allowing tokens to be evaluated one-at-a-time. Many of these languages then utilize metaprogramming in order to alter the execution order of the tokens, usually using quotes. However, they don't remove the requirement of *reading ahead*. Instead, they outsource this process to quotes, which are required for turing completeness in the form of recursion. Worse, many of these languages have strings, which require the parser or tokenizer to read ahead in order to package the string as a single token. Therefore, many of these languages are not perfect candidates for a perfectly postfix language.

Cognition solves this problem using three mechanisms: first, tokens are one character long by default and information about how Cognition will tokenize based on certain delimiter rules are made public to the language; second, Cognition uses a unique metacrank system which allows it to modify the execution order of the code, allowing Cognition to *modify the syntax for changing the execution order itself* due to the metacrank system's postfix syntax, and allow for metaprogramming; third, it uses what we call an *falias* system which we believe to be equivalent to metacrank, but makes for easier bootstrapping to an environment that is more desirable to program in.

*Corresponding author: matthewhinton@uvic.ca
†Corresponding author: preston@nullring.xyz

2 Language Design

As of the release of this paper, the language design is incomplete yet still covers many general cases. Tradeoffs in the implementation have been made such that the esscence of the design is there but the implementation is simplified. In this section, we describe the high level overview of the public components of Cognition, and we compare and contrast it with other similarly flexible programming languages such as Forth and Lisp.

In languages like lisp and forth, syntax can be modified using *macros*. The usage of macros allows for the publicity of the AST; that is, it allows for the change in execution order of symbols or words. Similarily, Cognition allows for this behavior, by using a *metacranking* system, which allows it to specify every *nth* word *n* items down the stack to be executed, where all the other words are not evaluated. Metacranking behavior can be modified using a postfix syntax, allowing metacrank execution order itself to be changed, meaning not only can the AST be modified, but the system used to modify the AST is itself a part of the AST and can be modified. This allows full AST modification, and we believe it to not be possible with many more naive implementations of Forth and many, if not all implementations of Lisp.

Because everything is written in postfix notation, there is almost no need for a special character that denotes the AST like in the case with the parentheses in Lisp. Instead, there is an falias system which force evaluates the top item on the stack in a *fast* way, without waiting for the cranker. It is conceptually possible to modify the bracket syntax for lisp and for Forth, but it is unwieldy due to its prefix nature.

This gives Cognition the benefit of never reading ahead. Because it never has to read more characters than it has to, only obeying a couple of delimiter, ignored character, and singlet laws, it allows the tokenization strategy of the parser to be changed without affecting the current token or future tokens in unwanted or conflicting ways. This allows other special characters that denote the delimiter between different tokens, for example, to be made public, and allows them to therefore be changed dynamically and programmatically, giving even more power to the AST manipulation as operations modifying the tokenizer can be stored in the stack and changed via the metacranker. Furthermore, string operations in Cognition modify the tokens post-tokenization, making post tokenization decisions about token execution possible.

The hash table that stores variable definitions is also made public, allowing the program to access currently defined variables, modify them, and delete them. This principle is made more powerful with metastack operations which have the main focus of reducing the possibility of hash table collisions and making program simulation and program introspection easy.

Metastack operations extend the definition of Forth quotes, attaching a hash table and other data structures required to make a complete cognition environment to quotes, making them self similar to the original environment. This is important for implementing scope, containerization, and self-simulation. This allows Cognition to be used as a permission system in a Cognition-centric virtual machine or operating system in the future.

Otherwise, Cognition is much like the programming language forth, with a stack-based postfix programming environment.

2.1 Parser Design

The parser is designed to make public the specifics of how characters are parsed into tokens by delimiter rules, and by a few other constructs called *ignores* and *singlets*. Ignores allow the parser to ignore a list of characters at the start of the read-eval-print loop, and delimiters and singlets allow the parser to determine the end of the next token.

The parser has three flags for these features, where the flags represent the decision to whitelist or blacklist the characters to be ignored, or to be singlets or delimiters that stop the parser from collecting the token. When the parser hits a delimiter, the parser stops and the token gets packaged, whereas when the parser hits a singlet, the parser skips the singlet character. This enables a wide range of source file text manipulation. If characters were not ignored, at least one character must be parsed into a word, and if not, a word containing no characters can be created with singlets.

By default, the parser starts with no ignored characters, no blacklisted delimiters (everything is a delimiter), and no singlets.

2.2 Falias Design

Force aliases, or faliases for short, are a list of special tokens that evaluate the top word on the stack unconditionally when tokenized. One can change the list of faliases and one can even make the list empty, effectively removing the falias functionality.

By default, *f* and *ing* are the only faliases. The ing falias is a pun on english verbs.

2.3 Metacrank Design

The metacranker is a cyclic automatic word evaluation system, where *cranking* refers to metacranking the first element on the stack, meaning that the top element of the stack executes once every n words executed. *Metacranking* refers to the mth element down the current stack executing once every n words executed. There can only be one word executed every crank, although all metacrank positions are incremented every cycle. Therefore, the top of the stack executes first, and the highest metacrank has the lowest priority, executing only if every other crank above doesn't execute.

There are two kinds of words that can be evaluated: macros, and definitions. Macros evaluate nonrecursively, whereas definitions evaluate recursively. Expansion of these structures in order to alter the execution order can be done easily, because words such as unglue allow for the public access of word definitions as data.

The metacrank default is set to zero, which you can take to represent an infinite crank cycle (it doesn't make sense to talk about a zero metacrank base otherwise). In this state, the only way one can execute words is using the *falias* system. However, one can quickly set a crank and get into a comfortable environment without having to use faliases.

2.4 Metastack Design

The metastack is a system by which the current working stack can be set. Each current working stack has a hash table associated and an error stack associated, which keeps track of the errors that occur in the environment. The metastack, or stack stack, is a global object that keeps track of the current directory in the same way one might have a present working directory on unix-like operating systems.

This allows for easy containerization of elements, easy object-oriented programming, and namespace collision avoidance.

2.5 Error Stack Design

The error stack is a stack that stores errors emitted by the current running program. Each container contains an error stack, meaning one can source the error of a word to a particular running metastack. The error stack can be manipulated and read into the regular stack of a given container, but it is not programmable in the sense that the regular stack is.

2.6 Builtins and the FLIT

Builtins are created to make all of these components transparent and modifiable to the present working container. The ability to access, delete, and check for stored data in all these structures and functionalities, and a complete computational basis for metacranking as well as conventional stack operations, as well as capabilities for control flow and other general combinators are needed. Each of these builtins are C functions, and can be loaded in during runtime dynamically using the word clib, which alters the foreign language interface table, or FLIT, to contain symbols and objects from other C shared libraries. This allows for the dynamic introduction of low-level features.

3 Future Improvements

In Cognition 1.0, we implement a wide variety of the infrastructure required to run a completely general system. However, such a system is not complete without the introduction of general aliases. In its current state, the falias system can evaluate a word in a *fast manner*, which is equivalent to calling the eval word fast. General aliases would implement this functionality in a more general way, allowing for the execution of any macro in a fast way, not just the eval word. This system potentially introduces a more general syntax span, and makes the implementation of many common syntax features trivial, instead of challenging.

Another such optimization that can be made is in some backend implementations of various features, including the removal of the FLIT and introduction of bulitins into the word table.

3.1 The Cognition Compiler

An implementation of a cognition compiler has yet to be implemented. This would optimize much of the overhead cognition currently has, and would enable cognition to bootstrap its own cognition machine, which would be a virtual machine environment in which cognition is ran as a system capable of managing users and permissions concurrently within containers. Another possibility would be a baremetal cognition based operating system wherein cognition is used as a shell environment and the container structure of cognition is used in order to tightly control the permissions of certain containers. Such a system would allow for a compatibility layer between all running programs to be established, and would integrate fully with the cognition programming language, enabling kernel programming to be done in Cognition.

4 Conclusion

In this paper we propose a general programming system that we argue has yet unseen capabilities to augment its own syntax during runtime, including the introduction of a metaprogramming system capable of altering metaprogramming syntax, and a robust system for making the process of delimiting tokens public.