Programming Language Design and Implementation

CSCI-200 Independent Study Project

Phillip A. Wells

April 30, 2018

Abstract

This paper proposes that the programming language is the most important tool available to the programmer. A well-designed language, like a well-designed car, is enjoyable to use, beautiful, and safe. We describe the basic principles of design, as well as the methods of programming language implementation. The motivation and design of the soda programming language are also described.

List of Figures

Contents

Introduction

Consider the programs given in figure 1. Both programs are variations on the Unix utility

```c
//echo.c
#include <stdio.h>
int main()
{
  int c;
  while((c = getchar()) != EOF)
    putchar(c);
}
```

```
//echo.awk
{ print $0 }
```

Figure 1: Two `echo` programs

`echo`, and perform essentially the same function at comparable speeds. One program is written in the C programming language, the other in awk [4]. Most importantly, one program is seven lines long (more if it were written in an unidiomatic style) while the other is so simple it can be typed and run straight from the command line.

What accounts for the differences between programming languages? All general-purpose languages are formally equivalent, and thus no one language has an advantage over another in terms of power. Yet there are thousands of languages in existence, and at least a few hundred are used in research and industry each day [5].

The short answer is that while any program can be written in any programming language, it is not necessarily easy to do so. Some languages—in particular, C—provide excellent facilities for memory management and are good for writing "low-level" programs such as operating systems. Other languages provide features well-suited to writing web applications, computer games, and so on. The purpose of a language informs the features it possesses, and as computers have improved in speed and storage, the priorities of the languages used

to program them have changed.

This paper explores a general theory of programming language design, beginning with a study of formal languages. Principles of design and implementation techniques are addressed in the latter sections. The final section describes the motivation and design of soda,[1] a new language for text processing.

1 Theory of Programming Languages

The theory of programming languages begins with an observation: Writing software is difficult. The properties that make a language desirable tend to coincide with those that make the writing easier. Familiar notation, automation of difficult tasks, and portability are key to the success of many modern languages. All these characteristics go a long way in disguising the fact that, under the hood, most programming languages are nearly identical. All languages have discrete lexical, syntactic, and semantic components. Alphabets describe the symbols that can be used to write words. Grammars provide a set of rules from which all legal sentences can be constructed. And the meanings assigned to each sentence describe a language's semantics [6].

Designing a new programming language may seem, at the atomic level, as simple as assembling a set of simple parts. In fact, it *is* as easy as it seems, as evidenced by the sheer number of languages available [2]. But creating a good programming language is a task involving equal parts mathematics, psychology, and design. Because computer programs are read much more frequently than they are written, a language that is difficult for humans to understand is next to useless. The look and feel of a language must be crafted to incline

[1] when the name of a programming language is given, we use the style preferred by its authors (e.g., LISP and FORTRAN are written in all caps, awk and soda in lowercase except at the beginning of a sentence.

its users towards effective problem solving. The features of a language should be carefully selected to minimize overlap and encourage efficiency, both of the programmer and of the machine. A language designer has to keep in mind all these elements, in addition to the technical details. Fortunately, just as the elements of writing style have been refined with time, a number of principles have emerged to guide the development of new languages.

1.1 Formal Languages

Before discussing these principles in any great detail, it may help to lay out the definitions and notation common to the study of formal languages. A formal language is defined as a set of strings, or sequences of symbols, over an alphabet Σ [6]. Suppose B is the language of all binary natural numbers. We let $\Sigma = \{0, 1\}$, and $\Sigma^* = \{\epsilon, 0, 1, 10, 11, ...\}$. The set Σ^* represents all possible strings over Σ, which always begins with the empty string ϵ. If Σ is not the empty set, then Σ^* has a countably infinite number of elements—that is, its elements are unending and can be placed in a one-to-one correspondence with the elements of the set of natural numbers $\mathbb{N}$. We say that B is a language over Σ if and only if $B \subseteq \Sigma^*$. Of course, B meets this criterion because it contains all but one string in Σ^*: the null string, which is not a number. B also contains an infinite number of strings, and while the majority of useful languages are infinite, this is not a requirement. There is even a null language $L_0 = \varnothing$, which contains nothing. Note that this is different from $L_\epsilon = \{\epsilon\}$, the language that contains the null string.

Not all languages are created equal, and a hierarchy of classes developed in part by Noam Chomsky has become the de facto standard for describing the differences between them. Languages belonging to the narrowest class are called *regular languages* [1]. The

strings in a regular language are described by structures called regular expressions, which are simple enough that they can be understood by stateless machines (e.g., vending machines can be said to accept regular expressions over an alphabet of coins). Strings in the binary language B are constructed from regular expressions, as are simple mathematical sentences such as $1+1-4$. Regular expressions see a great deal of use in pattern-matching engines and word search programs. But despite their ubiquity, regular languages fail to capture most of the work that computers are expected to handle. As an example, balancing parentheses is a task even a desk calculator can perform. But in order to find a language that can identify mismatched parentheses, we must move to the next linguistic class.

Context-free languages are excellent at describing the nested structures common to most programming languages. These languages are specified using context-free grammars, sets of recursively-defined rules for enumerating strings. Our binary number language B could be specified by the grammar given in Figure 2. As is traditional, we use capital letters to signify

$$S \Rightarrow SB|\epsilon$$
$$B \Rightarrow 0|1$$

Figure 2: A simple CFG

non-terminal sequences, and the or | symbol to separate different productions, or substitution rules [5]. The same grammars can be used to specify languages containing only palidromes, matched parenthesis, etc. Even better, all context-free languages are also regular languages, so a well-defined grammar can easily encompass the usefulness of any regular expression. This is not to say that there are no problems with context-free languages. If a grammar is not properly structured, it may be ambiguous, meaning that there are multiple ways to

derive the same string. Operator precedence and associativity are other sticking points and, if not factored into the design of a grammar, can produce yet more ambiguities. And because most useful grammars are recursive, it also is possible to accidentally define a grammar that generates infinite strings—not a terribly useful feature of a programming language.

Believe it or not, a context-free grammar is basically all that is required to parse most programming languages. Syntactic analysis is relatively simple, so long as the proper recursive techniques are mastered. Semantic analysis, on the other hand, is quite difficult. In order to perform any kind of evaluation of the meaning of a string, we must move on to the final class of languages, the *recursively enumerable (RE) languages*. A language is recursively enumerable if there exists a special model of computer, known as a *Turing machine*, that enumerates the strings of the language [6]. Turing machines are largely outside the scope of this paper, but it should be known that all general-purpose programming languages are recursively enumerable, and therefore can be evaluated by some TM. Furthermore, all recursively enumerable languages can be used to simulate any computable function (including any Turing machine!). This fact of mathematics is not just theoretically interesting, it is the reason that so many programming languages exist—any RE language can be used to implement any other RE language, so it is possible to write a LISP interpreter in C, then turn around and write a C compiler using the very same LISP interpreter [5].

1.2 Principles of Design

There is no reason except for convenience that programs are written in some languages and not others; a programmer can construct a very large piece of software in native computer language with the efficacy of an engineer building a skyscraper out of ham sandwiches. It

can literally be done, but it is not feasible given the limited time, memory, and sandwiches we are afforded. The final goal of any programming language is to assist the programmer in producing reliable, robust software [3], and to that end a variety of languages exist. R is particularly good at modeling statistical problems, Go supports concurrency and scales remarkably well, and Python is excellent for writing short "paragraph programs," such as web crawlers.

But why develop new languages at all? It seems on the surface much more efficient to just tack new features onto existing languages so that nobody is required to turn their workspace upside down to accommodate new programming techniques. The answer is that modifying existing languages is extremely difficult. Small changes can have unforeseen consequences for backwards compatibility, and the issue only compounds as a language gets larger [2]. Not to mention that a language of incredible size is unsettling to new programmers. Individual programmers would also fail to benefit from most additions: the addition of operating system-level features is of little use to a web developer, and the resulting language feels bloated and unwieldy.

New languages also present an opportunity to experiment. Without an entrenched user base, a language designer can feel free to test out new notation, add strange features, and research new implementation techniques.

Once we are motivated by the desire to develop a new programming language, we turn to questions of good practice. The principles of design are subject to argument and change, especially as the public perception of style changes. There are, however, some ideas that are good because they produce consistently good results. This section explores a few of these principles in the context of real-world programming languages.

1.2.1 Simplicity

The principle of simplicity states that languages should be as simple as possible [5]. Languages with a minimum number of features are more readily learned and understood by the broader programming community, which leads to less error-prone code. A simple language is also beneficial for the reader: if there are fewer opportunities for a programmer to think through the use of one feature over another, readers do not have to repeat the same thought process. In other words, a simple language begets simple, straightforward solutions.

As far as counterexamples go, the worst violators of the simplicity principle are probably C++ and Java. Both are dominant languages of education and industry, both have a comically large number of built-in features, and both are frequently criticized for their complexity.[2] Why do languages like these continue to thrive? Part of the answer likely lies in the complexity of modern computing problems. Problems like serving web pages to millions of concurrent users is sufficiently difficult that a language like C++ seems the best fit. Predicting well in advance the types of features a programmer needs during a large build is challenging enough that many companies prefer to hedge their bets and work with a feature-rich language. Another possibility is that working within a large, ever-evolving linguistic ecosystem is exciting. Clever programmers are no doubt masters of their tools, and new additions to the toolbox can reinvigorate the otherwise laborious task of writing code. However, none of these reasons seem compelling enough to reject the simplicity principle, even in part. Programs written in minimal and maximal languages are no different in execution, and a program written in a simple language is easier to maintain and change.

[2]Java is the target of so much ire that there is an entire Wikipedia page devoted to the topic: wikipedia.org/wiki/criticism_of_java

1.2.2 Abstraction

The principle of abstraction states that a programming language should never require something to be stated more than once [5]. Recurring patterns and redundancy are desirable traits in natural languages, where they guard against information loss due to poor enunciation or background noise. Fortunately for programmers, a computer cannot suffer from hearing problems, and so computer languages can afford to be terse. This brevity is often preferable, because it reduces the number of times code has to be copied by human fingers, and thus reduces the odds for copy errors. Better yet, providing the features for abstraction encourages well-structured programs. Writing subroutines to take care of discrete pieces of computation is beneficial for the programmer and to the language ecosystem, because well-tested modules can be shared with others. In a world without abstraction, every C programmer would be forced to reimplement the entirety of the standard input/output library for even trivial programs.

1.2.3 Orthogonality

In linear algebra, two vectors are *orthogonal* if they are perpendicular (in more general terms, when their inner product is zero). Sets of orthogonal vectors have many nice properties—they are guaranteed to be linearly independent, thus they form a basis for their vector space. If programming languages are vector spaces, then language features are their vectors. The principle of orthogonality states that good programming languages strive for orthogonality when possible, because it guarantees that each feature operates independently [5]. However, orthogonality is not a substitute for simplicity, and in extreme cases it can even be detrimental to the language. For example, a language with a built-in exponentiation operator

should also, by the principle of orthogonality, provide access to an operator that computes nth-roots. But such a function is rarely necessary—a square root function, perhaps, but does a general language really need a 20th-root function?—so its inclusion violates the simplicity principle.

A notorious counterexample to the principle of orthogonality is C, which has four looping mechanisms: `for`, `while`, `do-while`, and `goto`. Not only are three of these statements redundant, *The C Programming Language* even explicitly advises against the use of `do` and `goto` [4].

1.2.4 Automation

Related to the principle of abstraction is the principle of automation, which states that repetitious activities are best left to a computer [5]. Modern languages are particularly adept at this, sometimes obviating entire aspects of programming. Memory management, once critical to the discipline, has been phased out of most newer languages in favor of garbage collection systems. Even older languages have automatic memory management libraries, and using them provides a great benefit for not much cost. For languages with a specialized purpose, the principle of automation is less a guiding light and more a foundation upon which the entire language is designed. Awk takes the principle to an extreme by providing what is in effect a built-in file parser, and encouraging users to simply enter programs at the command line.

1.2.5 Other Principles

Of course, there is much more to language design than the four principles given above. But in modern languages, where principles of security and efficiency are automated, a language

designer would be hard-pressed to identify four rules more significant than these.

Additional principles not mentioned (but important nonetheless) include: elegance, portability, syntactic consistency, and security.

2 Implementation of Programming Languages

We now turn our attention briefly to the implementation of programming languages. The techniques presented here are nearly universal, and most programming languages are implemented using some variation on the standard model.

2.1 General Methods

Programming language implementations can be plotted on a spectrum. On one end is *interpretation*, a style that relies on the computer to execute programs on the fly. An interpreted language reads its input lines and produces output immediately, without first translating to another language. Languages commonly implemented using interpreters include LISP and PostScript.

On the other end of the spectrum is *compilation*, the process of translating one program into an equivalent program in another language. A language is typically described as "compiled" if a program in that language can be translated to native machine code and stored in an executable file. Languages commonly implemented using compilers include C and Haskell.

Many languages do not fit strictly into the compiled/interpreted dichotomy. Python, for example, is typically implemented as a *bytecode interpreter*, meaning that Python source code is first compiled into a special language understood by the Python virtual machine, then interpreted. Java is also implemented as a bytecode interpreter, but with an additional

component called a *just-in-time compiler*, which waits until runtime to translate the source program into native machine code.

2.2 Lexical Analysis

Source programs are first analyzed by a program called a *lexical analyzer*, also called a lexer, scanner, or tokenizer. This program scans the input file one character at a time, passing on structures that it recognizes (numbers, special symbols, and keywords) and eliminating characters that are not part of the main program (typically whitespace characters such as spaces and newlines). Some lexical analyzers are implemented as simple machines that recognize regular expressions, while other lexers are more complex [1]. The soda lexer, for instance, maintains a record of the last structure consumed and inserts extra information when necessary. Many lexers also maintain state in the form of line and column numbers, to make reporting errors easier.

2.3 Syntactic Analysis

The structures identified by the lexer (known as *tokens*) are passed on to a program called a *parser*, which matches them against a context-free grammar. If a sequence of tokens matches a production, a node is added to a data structure known as an *abstract syntax tree*. A naive interpreter may simply wait until the AST is finished, then recurse along the nodes of the tree, executing code as necessary. Otherwise, the finished tree is passed on to a compiler.

2.4 Compiling

Once the general structure of a program is known, the compiler can translate it. A typical evaluation involves the compiler recursively walking the abstract syntax tree, emitting

code in an *intermediate representation*, or—if the language is implemented as a bytecode interpreter—emitting bytecodes for the virtual machine to evaluate.

As an example, consider how the soda compiler translates the input string `1 + 1`. The lexer reads the string and produces four tokens: `1`, `+`, `1`, and the special token `END`, which signals the end of a statement. The tokens are passed to the parser, which matches them against a production in the context-free grammar. The parser produces a tree that describes the structure of the program—in this case, the `+` token forms the root node, and the `1` tokens are leaf nodes (the `END` token is useful only to the parser, and is discarded during this step). The compiler begins at the root and evaluates its children, translating each node into the appropriate bytecode. Figure 3 gives the output of this translation as a table of values: each row contains a bytecode instruction and its argument.

```
LOAD_CONST 0
LOAD_CONST 1
       ADD 0
DROP_CONST 0
```

Figure 3: Example bytecode

The first two bytecodes pop the values `1` and `1` from the program stack, while the third adds them together and pushes the result back on the stack for later use. The final bytecode is generated when the compiler determines that the constant on the stack is unnecessary, and removes it to save memory and execution time.

If an intermediate representation is generated, that code is passed to yet another program called an *assembler*, responsible for generating assembly language for the target computer architecture. The assembly code is then *linked* to the compiled object code of any library files, and then translated into absolute machine code [1].

3 The Soda Programming Language

Soda is a procedural programming language with facilities to simplify text processing tasks. This section provides a brief overview of the language with regards to the principles of language design given in §1.2. Example soda code is provided in §B.

3.1 Simplicity

Soda has 13 reserved words and one predefined function, `len()`, which returns the length of an array or a string. The language is restrictive, sometimes to a fault; programs must be structured in a way that the compiler finds agreeable. Operators and operands must be separated by whitespace, and multiple statements per line is prohibited (except in `for` loops).

In general, soda was designed so that there is exactly one idiomatic way to write any given program. All primitive data are fundamentally strings, and the only container type is the associative array (referred to simply as an "array"). Control-flow is provided by `for` loops and `if` statements, but the syntax of these structures is highly regular. The `for` loop comes in three flavors: `for` *init*`;` *condition*`;` *post*, or the traditional C-style loop; `for` *condition*, or the traditional while loop; and `for` *var* `in` *expression*, which iterates over the indices of an array.

The `if` statement is less varied, and is always written as `if` *condition* `then` *statements* `else` *terminating statement*. All three parts are required. The regularity disambiguates whether a block of code belongs to an `if` (a problem with languages such as Pascal and Python), and because only one statement may follow an `else`, programmers are encouraged

to structure their code in relatively uniform way.

3.2 Abstraction

Functions in soda are incredibly terse. A complete function is defined as `fn` *name* `(`*parameters*`)` `=` *return value* `where` *statements*. There are no return statements, so the value of a function is specified up front and clarified in the `where` clause if necessary. In fact, the `where` clause is entirely optional—a function that doubles its input can be written on a single, readable line: `fn Double(x) = x * 2`. Note that there is no `return` statement. Because there is no null in soda, all functions must produce a value when called.

Soda also has a robust module system, inspired by the import mechanisms of Go, Java, Modula, and to a lesser extent, C. Using the `fetch` keyword, functions and constants from other packages can be imported and used. Each function and variable belongs to a namespace named for the package it originates in. The `io.Println()` function, for example, belongs to the `io` namespace, which has to be named in any file outside `io.na`. Repeat imports are ignored, so the programmer is free to `fetch` whatever files are necessary for a program to run without worrying about bloat.

3.3 Orthogonality

Soda obeys the principle of orthogonality within reason. Included are the standard mathematical operators, as well as corresponding addition and substraction operators for strings: `a ++ b` concatenates strings `a` and `b`, and `a -- b` returns `a` with all instances of substring `b` removed.

Unlike languages in the C family, soda has only one looping mechanism, though the addition or omission of certain syntactic structures changes its behavior. Similarly, there is no redundancy in functions, and necessary information is included in an optional clause added after the function's return value.

3.4 Automation

Soda has no built-in functions for scanning input from file streams. Instead, soda programs accept command-line arguments specifying the location of input data. The files are opened, scanned, and closed automatically, and the data is provided to the program in the form of the predeclared arrays `chars`, `words`, and `lines`. If multiple files are specified, then the program is run using the data from each file in sequence. Thus, performing computations using many files in a directory is as easy as typing `soda myProgram.na myDirectory/*` at the command line.

4 Conclusion

Does the perfect programming language exist? If all programming languages are formally equivalent, and all languages are implemented according to the same basic model, then linguistic differences must be accounted for by notation and features. Given the diversity of interests among programmers, not to mention the variety of problems they are expected to solve, it is reasonable to conclude that no programming language will ever be a "one-size-fits-all" tool [5]. Continued research into functional and nonprocedural programming will shape the field in strange and unforeseen ways. Even the industry-standard software tools

will continue to evolve as hardware changes. The best languages are invariably the languages that best suit the problems they were intended to solve. And perhaps the languages of the future will create new users, problems, and solutions of their own.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc, 1986.

[2] C. A. R. Hoare. "Everything You've Wanted to Know about Programming Languages but Have Been Afraid to Ask". Typed manuscript; "probably consultancy report to DoD workshop on what became Ada". 1978.

[3] C. A. R. Hoare. *Hints on Programming Language Design*. Tech. rep. Stanford, CA, USA: Stanford University, 1973.

[4] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988.

[5] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. 3rd ed. Oxford University Press, 1999.

[6] Jerud J. Mead. *Programming Languages Through Translation: A Tutorial Approach*. Tech. rep. Lewisburg, PA: Bucknell University, 2014.

A Sample Source Code

A.1 target.py

```
1  from rpython.jit.codewriter.policy import JitPolicy
2  from rpython.rlib.streamio import open_file_as_stream
3  from soda.interpreter import interpret
4  from soda.parser import parser
5  from soda.bytecode import compile_ast
6  from soda.fetcher import fetcher
7  import os
8  import sys
9
10
11 def main(argv):
12     isdump = False
13     norun = False
14     sourcefound = False
15     extrafiles = []
16     for arg in argv:
17         if arg.startswith("-"):
18             if arg == "--dump":
19                 isdump = True
20             elif arg == "--norun":
21                 norun = True
22         elif arg.endswith(".na"):
23             if not sourcefound:
24                 sourcefound = True
25                 root = arg.rstrip(".na")
26                 root = os.getcwd() + "/" + root
27                 fetcher.addpackage(root)
28         else:
29             extrafiles.append(arg)
30     del extrafiles[0]
31     if sourcefound:
32         bc = compile_ast(parser.parse(fetcher.gettokens()))
33         if isdump:
34             print(bc.dump())
35         if not norun:
36             if extrafiles == []:
37                 bc.create_arrays("")
38                 interpret(bc)
39             else:
40                 for filepath in extrafiles:
41                     try:
42                         sourcefile = open_file_as_stream(filepath)
43                         data = sourcefile.readall()
44                         sourcefile.close()
45                         bc.create_arrays(data)
46                         interpret(bc)
47                     except OSError:
48                         print("file %s not found" % filepath)
49                         os._exit(-1)
```

```
50      return 0
51
52
53  def jitpolicy(driver):
54      return JitPolicy()
55
56
57  def target(driver, args):
58      driver.exe_name = "csoda"
59      return main, None
60
61
62  if __name__ == "__main__":
63      main(sys.argv)
```

B Soda Standard Library

B.1 io.na

```
1  # Package io governs the basic input/output functionality of soda.
2  # The wrapper functions Print and Error are declared here,
3  # as are functions for formatted I/O.
4
5  # Wrapper function.
6  # Writes to the standard output.
7  fn Print(arg) = 0
8
9  # Wrapper function.
10 # Writes to the standard error.
11 fn Error(arg) = 0
12
13 # Writes to the standard output.
14 # Spaces are added between arguments, and a newline is appended.
15 # Returns number of arguments.
16 fn Println(arg, vargs) = argNum where
17     argNum := len(vargs) + 1
18     Print(arg)
19     for j := 0; j < len(vargs); j := j + 1
20         Print(" ")
21         Print(vargs[j])
22     end
23     Print("\n")
24 end
25
26 # Writes to the standard error.
27 # Spaces are added between arguments, and a newline is appended.
28 # Returns number of arguments.
29 fn Errorln(arg, vargs) = argNum where
30     argNum := len(vargs) + 1
31     Error(arg)
32     for j := 0; j < len(vargs); j := j + 1
33         Error(" ")
34         Error(vargs[j])
35     end
36     Error("\n")
37 end
```

B.2 arrays.na

```
1  # Package arrays defines useful functions for manipulating arrays.
2
3  # Reports the first instance of an item in an input array.
4  # Returns "false" if no index is found.
5  func Index(array, item) = index where
6      index := "false"
7      for idx in array
8          if array[idx] == item
9              then index := idx
10                 break
11             else ""
12     end
13 end
14
15 # Normalizes the indices of an input array.
16 # Returns the normalized array.
17 func Normalize(array) = normArray where
18     normArray := []
19     for idx in array
20         normArray[len(normArray)] := array[idx]
21     end
22 end
```