Phonecode in Axiomatic Language

XX
XX
XX, XX, XX
XX

Abstract

The phonecode project of Lutz Prechelt [7] measured program size and programming time for multiple implementations of the same problem in different languages with different programmers. The results support the maxim that programming time per line of noncomment code is roughly constant, regardless of language. A 4-line axiomatic language specification for the phonecode problem is presented, suggesting a productivity benefit.

CCS Concepts: • **Software and its engineering** → *Software notations and tools*; **Functional languages**; **Constraint and logic languages**; **Extensible languages**; **Multiparadigm languages**; **Very high level languages**.

Keywords: programming productivity, phonecode, specification, axiomatic language

ACM Reference Format:
XX. 2026. Phonecode in Axiomatic Language. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

This project makes a quantitative argument for a programming productivity benefit for a type of logic programming called "axiomatic language" [12], [10], [11]. Axiomatic language is intended as a pure specification language, so its implementation requires automatically transforming a user's specification into an equivalent, efficient program — a grand challenge of computer science. That remains to be accomplished, but it is worthwhile to try to assess the language's software engineering benefit to decide whether that grand challenge is worth attempting.

Lutz Prechelt created a test problem called "phonecode" which had multiple implementations by different programmers to provide a quantitative comparison of seven programming languages. [8] He mentions the "old rule of thumb, which says that programmer productivity measured in lines of code per hour is roughly independent of the programming language" (p.28), and this seems to be supported by his data. Thus, a language that enables a shorter phonecode solution would suggest greater programming productivity. This project originated with the realization that we do not need an implementation to write an axiomatic language specification for the phonecode problem. If the result is significantly shorter than other solutions, that would suggest a potential programming productivity benefit if the language can be implemented. Thus, the goal of this project is to see how small a phonecode solution can be in axiomatic language.

Some caveats to be mentioned are that, being unexecuted and thus untested, there is a high probability that errors exist in the solution, but we assume errors can be fixed without significantly increasing the solution size. Also, axiomatic language is extremely minimal, so it must be built-up with assorted utility functions and data types, which typical programming languages have built-in. Definition of this general-purpose functionality would go into a standard library available to all applications and thus should not be counted as source code for any particular application. Finally, unlike most existing phonecode solutions, programming time was not measured. The solution of this paper evolved gradually over several years. The goal was to create the best axiomatic language phonecode solution possible, ignoring time constraints. It is assumed that a hypothetical future experienced axiomatic language programmer, fluent with the utility library, would be able to envision a declarative solution to a typical problem and write it out in time consistent with the size of the specification.

Section 2 reviews axiomatic language. Section 3 discusses the assumption of roughly-constant programming time per non-comment line of code. Section 4 presents the axiomatic language phonecode solution and section 5 defines utility functions used in that solution. Section 6 gives some final comments. The contributions of the paper are the dramatically shorter phonecode solution and the supporting library of utility functions.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

2 Axiomatic Language

This section reviews axiomatic language. Axiomatic language has the following goals: (1) pure specification, (2) minimal, but extensible, (3) metalanguage capability, and (4) beauty. Section 2.1 gives the main idea of axiomatic language. Section 2.2 gives the definitions and rules of the language core, and section 2.3 gives syntax extensions, including a new one. Section 2.4 has some examples and some summary comments.

2.1 Main Idea – Specification by Enumeration

The main idea of axiomatic language is that the external behavior of a program — even an interactive program — can be represented by a static, infinite set of symbolic expressions. Each expression encodes the program input — or sequence of inputs — along with the corresponding output for an execution of the program, as seen by an external observer. The set of such expressions would enumerate all possible executions, one for each possible program input. Our claim is that this infinite set of symbolic expressions idealistically specifies the external behavior of a program, without having anything to say about the internal processing. Note that in axiomatic language the symbolic expressions have no inherent meaning; they are interpreted by the human user and the implementation system to represent bits, characters, lines of text, etc. of the external real-world environment.

For a program that reads an input text file and writes an output text file, each 'Program' expression can represent the input and output files with a sequence of symbolic expressions representing lines, each a sequence of symbolic expressions representing characters. For a program that sorts the lines of an input file, the infinite set of expressions would enumerate each possible input text file along with the sorted output file:

```
(Program () ())    - empty input -> empty sorted output
         ...
(Program ("C" "A") ("A" "C"))  -two 1-char lines sorted
         ...
(Program ("cat" "ox" "ant") ("ant" "cat" "ox"))  3-lines
         ...
```

These symbolic expressions show character strings, but, as section 2.3 explains, these character strings are just a syntax extension for underlying abstract symbolic expressions, which are then interpreted as character strings. The infinite set of these Program expressions specifies this sorting program without specifying a sorting algorithm.

For an interactive program that, say, reads and writes lines of text in a text window, a typical program might write blocks of zero or more output lines alternated with single input lines typed by the user. Each Program expression would contain the sequence of inputs and outputs for an execution history. The infinite set of Program expressions would give all possible execution histories based on all possible inputs the user might type at any point.

Consider a program that accepts arbitrary input strings and checks whether any parentheses present are balanced, giving an error message if not. An empty input string halts the program. An example Program expression (with annotations) might be as follows:

```
(Program
 ("Enter char strings with balanced parentheses."
  "Empty string ends program.")   - initial output
  "((xyz)a b)()"           - correct user input line
 ()                          (no output lines)
  "(W) (W))12)"            - incorrect input line
 ("       ^ unmatched right paren")   - err msg
  "((( )"
 ("     ^ unmatched left paren(s)")
  ""                       - empty input ends prog
 ("End-of-program.")       - final output line
)
```

This symbolic expression represents a possible execution of this interactive program as seen by an external observer. An infinite set of these expressions, for all possible execution histories, can be considered a static specification of this interactive program.

For interactive graphics it shouldn't be difficult to come up with a convention for symbolically representing screen pixels and mouse movements. The implementation system would need to "understand" this convention in order to synthesize a program with a graphical interface.

The emphasis here is that these expressions are just abstract hierarchical symbolic expressions with no built-in meaning. They are just interpreted by the human user and a future implementation to represent an external environment. Note that in order to define Program expressions one will likely need to define predicates for supporting utilities like arithmetic, but these would also be symbolic expressions interpreted by the human user and implementation.

Using an infinite set of abstract symbolic expressions to specify the external behavior of a program provides a clean, idealistic separation between specification and implementation. It is a nice solution to the "awkward" problem of input/output in declarative languages. All that is required of the specification language is that it be a formal system for defining these infinite sets — and that is what axiomatic language is. Unlike Prolog, with its ugly non-logical input/output predicates, axiomatic language is completely pure.

2.2 The Core Language

In axiomatic language a finite set of axioms generates a (usually) infinite set of valid expressions. An **expression** is

an **atom** – a primitive atomic symbol,
an **expression variable**,
or a **sequence** of zero or more expressions and **string variables**.

Syntactically, atoms are represented by symbols that begin with a backquote: `abc, `+. Expression and string variables begin with % and $, respectively. Sequences have their elements separated by blanks and enclosed in parentheses: (`M () (% $1)).

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions:

```
<conclu> < <cond1>, ..., <condn>.
<conclu>.                ! an unconditional axiom
```

Comments start with an exclamation point and run to the end of the line. We also have comment blocks that start and end with !/ and !\, respectively, in column 1:

```
!/    comment
  ... comment lines ...
!\    comment
```

Comment blocks can be nested.

Axioms generate **axiom instances** by the substitution of values for the expression and string variables. An expression variable can be replaced by an arbitrary expression, the same value replacing the same variable throughout the axiom. A string variable can be replaced by a string of zero or more expressions and string variables. For example, the axiom

```
(`A %x $w)< (`B ($ %y %x)), (`C $w).
```

has an instance

```
(`A `x `u `v)< (`B (() `x)), (`C `u `v).
```

by the substitution of `x for %x, () for %y, the string `u `v for $w, and the null string for $.

Axiom instances generate **valid expressions** by the simple rule that if all the conditions of an axiom instance are valid expressions, the conclusion is a valid expression. By default, the conclusion of an "unconditional" axiom instance is a valid expression. For example, the two axioms

```
(`a `b).
((%) $ $)< (% $).
```

generate valid expressions (`a `b), ((`a) `b `b), (((`a)) `b `b `b `b), ...

2.3 Syntax Extensions

The expressiveness of axiomatic language is enhanced with some syntax extensions. A single character in single quotes is equivalent to writing an expression that gives the ASCII code of the character using bit atoms:

```
'A'  ==  (`char (`0 `1 `0 `0  `0 `0 `0 `1))
```

A character string in single quotes within a sequence is equivalent to writing the characters separately in that sequence:

```
(... 'abc' ...)  ==  (... 'a' 'b' 'c' ...)
```

A character string in double quotes represents the sequence of those characters:

```
"abc"  ==  ('abc')  ==  ('a' 'b' 'c')
```

A symbol that does not begin with ` % $ ( ) ' " is syntactic shorthand for an expression that gives the symbol as a character string,

```
ABC  ==  (` "ABC")
```

and uses the atom represented by just the backquote.

A new syntax extension allows one to write an atom, expression variable, or syntax extension symbol immediately followed by the left parenthesis of a sequence without a separating space. This is equivalent to an expression where a sequence encloses the symbol and original sequence:

```
xyz(...)  ==  (xyz (...))
```

This saves 3 characters and improves readability.

2.4 Examples

Here are axioms for natural numbers in successor notation and their addition:

```
(num 0).                  ! zero is a natural number
(num (s %n))< (num %n).     ! successor of nat num

(plus 0 % %)< (num %).      ! 0 + n = n
(plus (s %1) %2 (s %3))<    ! 1+n1 + n2 = 1+n3 if
  (plus %1 %2 %3).          !   n1 + n2 = n3
```

These axioms generate valid expressions such as (num (s (s 0))) and (plus (s (s 0)) (s 0) (s (s (s 0)))), representing the statements "2 is a natural number" and "2 + 1 = 3", respectively.

String variables enable more concise definitions of list predicates:

```
(in % ($0 % $1)).       ! expr is member of sequence
(cat ($0) ($1) ($0 $1)).  ! concatenation of 2 seqs
```

Axiomatic language can be summarized as pure, definite Prolog with Lisp syntax, string variables, and HiLog[2]-like higher-order generalization. Lisp syntax provides a single uniform representation for data lists, terms, functions, and predicates. It provides syntactic flexibility for new language features, like infix operators, and natural support for higher-order forms, where code is treated as data. String variables complement expression variables. An expression variable represents a single expression; a string variable represents a string of zero or more expressions and string variables within a sequence.

Note that axiomatic language does not include built-in true/false values. However, this concept is easily defined and one can then define Boolean functions. Axiomatic language is more like a type of formal language, except that instead of generating words as flat strings from a finite alphabet, axiomatic language generates recursively-enumerable hierarchical expressions formed from an infinite set of atom symbols and variables.

Axiomatic language also differs from Prolog in its goal of minimality and purity – no input/output operations, no state changes, no non-logical operations, no built-in predicates of any kind. For example, there is no built-in function for inequality between distinct atoms, but the utility library shows how this can be defined. The axiomatic language emphasis on pure specification means there is no operational semantics. Specification by enumeration defines program external behavior without defining internal computation steps. The only "semantics" for axiomatic language is the inference rules for generating axiom instances from an axiom and for generating valid expressions from axiom instances.

```
! phonecode - encode phone numbers with words from a dictionary
!/
  A letter>digit map is applied to a dictionary file of words, one-per-line,
  to produce a matching of digit strings to words.  Each word has >0
  upper/lowercase letters.  Phone numbers from a text file, one-per-line,
  are encoded as a string of words whose digit strings combine to form
  the number, with single phone number digits inserted where there is no
  matching word starting with that digit (but no consecutive digits allowed).
  Phone number-code pairs are written to an output file, one-per-line:
     phnum: |d|[d ]word{[ d] word}[ d]    - code words/digits (& blanks)
  Words and phone numbers can have special characters, which are ignored in
  the encoding but included in the output.  A phone number may have >=0 codes.
!\

1 (Prog: ^(e jnq rwx dsy ft am civ bku lop ghz) digs>wd ph#enc pcfmt).
       !  0  1   2   3  4  5   6   7   8   9   - digit for each uc/lc letter
  ! -> (Program _dic _ph#s _outf)  - generated Program valid exprs

! Supporting functions and stack states (arguments numbered 0..):
    !stk:      _dic    _ph#s     - initial stack w input text files
! ^(e jnq ... ghz)     - "constant fn" pushes ltr>dig map expr onto stack
    !stk:    (e jnq ... ghz)   _dic   _ph#s

2 (: digs>wd *sel1 x>d ltr> *&>{^ *rev).  ! get digits>word map
    !stk:     _digs>wd    _ph#s    - digs>wd map is seq of (digstr word) pairs

3 (: ph#enc dup1 (1 1*dig?{ *eat1^ *~cnsc?dig?{) *join).   ! get codes
    !stk:    (.. (_ph# _ph#codes) ..)   - each phnum and its seq of codes

4 (: pcfmt *./join cat* (*ins1 ": ") (*:2ins_ ' ') *chxcat).  !format p-c pairs
    !stk:      _outf    - output text file w formated ph#-code pairs
```

Table 1. Phonecode solution in 4 non-comment, non-utility source lines

3 Roughly Constant LOC/hr?

The idea that programming time per noncomment line of code (LOC) is roughly constant regardless of language level was an early observation in software engineering. Fred Brooks writes "Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include" [1] (p.94). Prechelt [8] (p.25) refers to "Common software engineering wisdom, which says that 'the number of lines written per hour is independent of the language'".

Prechelt's phonecode results support this claim, where lines-of-code are "anything that contributes to the semantics of the program in each of the program source files, e.g. a statement, a declaration, or at least a delimiter such as a closing brace (end-of-block marker)" [9] (p.31). Based on median program lengths and programming times, "non-scripts [C, C++, Java] are typically two to three times as long as scripts [Perl, Python, Rexx, Tcl]" (p.31) and "scripts (total median 3.1 hours) take only about one third of the time required for the non-scripts (total median 10.0 hours)" (p.37), with some caveats, such as the script times being self-reported. The median LOC/hr values for the seven languages are shown in Fig. 24 (p.41) and are all between 20-40.

Erann Gat repeated Prechelt's phonecode study with Lisp implementations [3]. The min, max, mean, and median program lengths for Lisp were 51, 182, 119, and 134, respectively, and for C/C++/Java were 107, 614, 277, and 244, about double the Lisp values. Median development times (from Fig. 1) for Lisp (about 5 hours) was about half that of C/C++/Java (about 10-11 hours), giving similar LOC/hr for the two groups.

The Prechelt and Gat results justify the project assumption of roughly constant LOC/hr regardless of language. Paul Graham [4] argues this and makes the logical inference that "the main point of high-level languages is to make source code smaller". For very-high-level, declarative, functional, and logic languages, there may be unrealized potential to make our programs smaller and thus achieve greater productivity, without sacrificing readability. Exploring that possibility is the motivation for this project.

4 The Phonecode Solution

Table 1 shows the source file for an axiomatic language solution to the phonecode problem. (The general-purpose utility library is defined in other files linked in the next Section.) A terse description of the phonecode problem is included in the source file comments. An expanded description with examples can be found in the Prechelt references.

This source file gives four 1-line phonecode-specific axioms. The top-level 'Prog:' axiom defines the 'Program' valid expressions

which hold instances of the dictionary and phone number input text files along with the corresponding output text file. A functional style is adopted in which function compositions operate on a stack. (Axiomatic language is good at incorporating other paradigms.) The program is defined as a composition of three phonecode subfunctions, digs>wd, ph#enc, and pcfmt, which themselves are defined entirely from utility functions. Note that for conciseness this solution uses short names including single-character prefixes for common higher-order forms, which make them easily composable. An expanded source file on the language website gives the phonecode axioms with added tutorial comment blocks that explain how the utility functions operate on the stack.

One item to mention is that this phonecode functionality is slightly more general than the Prechelt version. This solution allows arbitrary special characters in words and phone numbers, but the original phonecode problem only allows - " and - / in words and numbers, respectively.

Other short phonecode solutions include the three smallest in Prechelt's study: Python (42 lines), Tcl (44 lines), and Perl (49 lines) [9](p.59-60); Peter Norvig's 45-line Lisp solution [6]; and a 27-line Haskell solution by John Hamilton [5].

5 Used Utilities

This section gives an overview of used utility predicates and gives links to the source files that contain them.

.../util/0form.ax.txt - utilities involving expression form:
`((^ x) x)` - constant function
- expr *x* is pushed onto front of stack
`(rev seq rev'd-seq)` - reverse a sequence
`(join x0 x1 (x0 x1))` - join two exprs into 2-elem seq
`(cat* ((..str0..) .. (..strk..)) (..str0.. .. ..strk..))`
- catenate a seq of >=0 seqs into one
`(ins_ x) seq seq')` - insert expr between elems of seq

.../util/1ho.ax.txt - higher-order predicates and functions:
`(Prog: ..fn..)` - define Program exprs from fn composition
`(: fcname ..fn..)` - define name for a function composition
`((* pred) ..argseqs..)` - pred valid for tuples formed from arg seqs
`((& fn) ..args.. (..args.. res))` - fn input args merged w fn result
`((. fn) (..args..) res)` - fn input args are grouped into a seq
`((/ binfn) arg0 arg1seq resseq)`
- / binary fn on arg0 and arg1 seq gives result seq
`((: ..fn..) (..stk..) (..stk'..))` - apply fn compos to stack args in seq

.../util/2natnum.ax.txt - natural number utilities
.../util/3bit.ax.txt - bit atom utilities
.../util/4atom.ax.txt - atom "declarations" & predicates
- Utilities in these source files are not directly used.

.../util/5char.ax.txt - character predicates and functions:
`* & . / : d` - these h-o forms can be fn/pred sym prefix `(sel`*d* *seq elem*`)` - select element[*d*] in a sequence
`(x>d xseqs x>d)` - given seq of <=9 expr seqs,
get map of expr seq elems to digits '0'..<='9'
`(ltr> l>x l>x')` - generalize letter map to both cases
fn`^` - stack arg0 is made fn parameter
`dup`*d* - duplicate stack arg[*d*] onto front of stack
`(`*d* *..fn..*`)` - apply fn compos to stack suffix starting at arg*d*
`(`*d* *fn*`)` - apply *fn* to arg*d* stack suffix
`(chxcat chexpr chstr)` - flatten char expr into a char string
(char expr is char or seq of >=0 char exprs)

.../util/6bool.ax.txt - Boolean (related) functions:
`dig?` - returns true if char expr arg is digit, else false
type?`{` - filter seq exprs - keep when *type?* is true
`(>{` *map*`)` - apply map to seq exprs when defined, else skip
`cnsc?`*type?* - true for seq with consecutive exprs of *type?*
`~`*boolfn* - complement a Boolean fn

.../util/7stream.ax.txt - stream processing:
`(eat1` *bitemap*`)` - "eat" input seq making output seq
- when no "bite" possible, copy 1 input expr to output

See the language website for the utility source files and a file of abbreviations used in code and documentation.

6 Conclusion and Future Work

An axiomatic language solution to the phonecode problem has been defined in just 4 non-comment, application-specific lines — much shorter than other phonecode solutions that have been found. More example programs need to be written in axiomatic language to see if a small size is typical. The utility library would continue to be evolved to support a variety of applications. At some point user experience with axiomatic language specifications would be sufficient that development time could be measured. If development time per non-comment line of code is not much longer than that of other languages, that would be strong quantitative evidence for a programming productivity benefit and this assessment could be obtained before an implementation is available.

The attributes of axiomatic language contribute to the small size of specifications. A conventional language program defines the implementation algorithm, which implicitly defines the program external behavior. An axiomatic language specification defines just the external behavior — a proper subset of the conventional program information.

The minimal, purely-symbolic nature of axiomatic language means that basic functions can get idealistic, elegant, foundational definitions. The language extensibility makes it easy to define powerful higher-order forms and to define new functions from existing ones with fine-grained modularity. The Lisp syntax may be the simplest, most-general form for representing data and code. The metalanguage capability of axiomatic language means that new language features, even new paradigms, can be defined and used without one leaving the language. Axiomatic language can thus subsume other languages for the purpose of specification.

Axiomatic language may provide other software engineering benefits. The small size and purity of axiomatic language makes it well-suited to proof. Proof would be used to guarantee equivalence between a user's specification and the generated efficient program. Proof would also be used to prove assertions about a specification to validate it, which would be more powerful than just testing.

Additional syntax extensions could be defined for increased expressive power. A convention for in-line blocks of text in axiomatic language source files could be useful. It would be interesting to use byte codes x80 - xFF for new character symbols to represent

commonly-used functions and higher-order forms. These enhancements would not modify the definitions and rules of the core language.

The grand challenge of axiomatic language, of course, is its implementation — a system that can automatically transform a specification into an equivalent efficient program. An implementation would need comprehensive "knowledge" of programming concepts and specification patterns in order to "understand" the input specification. Then it would use algorithm knowledge with embedded proof to generate an equivalent program from that understanding. One can argue that no system would be able to automatically transform all possible specifications that a user might write, but that may not be necessary. A successful implementation would be one that could automatically transform straightforward specifications for typical problems. If a specification is not so straightforward or the problem not so typical, then an expert would need to add knowledge to the system so that automatic transformation could proceed.

Acknowledgments

... may be provided ...

References

[1] Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley. https://web.eecs.umich.edu/~weimerw/2018-481/readings/mythical-man-month.pdf

[2] Weidong Chen, Michael Kifer, and David S Warren. 1993. HiLog: A foundation for higher-order logic programming. *The Journal of Logic Programming* 15, 3 (1993), 187–230.

[3] Erann Gat. 2000. Lisp as an Alternative to Java. *Intelligence* 28, 1 (Winter 2000). doi:10.1016/S0160-2896(00)00035-7

[4] Paul Graham. 2002. Succinctness is Power. Retrieved May 2, 2026 from https://paulgraham.com/power.html

[5] John Hamilton. 2006. *Phone number*. Retrieved May 9, 2026 from https://wiki.haskell.org/Phone_number

[6] Peter Norvig. 2000. *Lisp as an Alternative to Java*. Retrieved May 9, 2026 from https://www.norvig.com/java-lisp.html

[7] Lutz Prechelt. 2000. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. Technical Report. Dept. of Informatics, Univ. of Karlsruhe. https://page.mi.fu-berlin.de/prechelt/Biblio/jccprtTR.pdf

[8] Lutz Prechelt. 2000. An empirical comparison of seven programming languages. *Computer* 33, 10 (October 2000). doi:10.1109/2.876288

[9] Lutz Prechelt. 2003. *Are Scripting Languages any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java*. Advances in Computers, Vol. 57. Elsevier, Karlsruhe, Germany, 205–270. https://page.mi.fu-berlin.de/prechelt/Biblio/jccprt2_advances2003.pdf

[10] Walter Wilson and Yu Lei. 2012. A Tiny Specification Metalanguage. In *Proc. of 24th Intl. Conf. on Software Engr. & Knowledge Engr. (SEKE'2012)*. Knowledge Systems Institute Graduate School, Skokie, IL USA, 486–490. http://ksiresearchorg.ipage.com/seke/Proceedings/seke/SEKE2012_Proceedings.pdf

[11] Walter W. Wilson. 1982. Beyond PROLOG: Software Specification by Grammar. *SIGPLAN Not.* 17, 9 (Sept. 1982), 34–43. doi:10.1145/947955.947959

[12] Walter W. Wilson. 2001. *Axiomatic Language Home Page*. Retrieved May 2, 2026 from http://axiomaticlanguage.org

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009