# A Minimal Specification Language*

Walter W. Wilson

Computer Sciences Corporation
6100 Western Place, MZ5302, Fort Worth, TX 76107
& The University of Texas at Arlington
wwilson3@csc.com

**Abstract.** This paper describes a logic programming language intended as a specification language where the user specifies software functionality while ignoring efficiency. The language objectives are (1) a pure specification language -- "what, not how", (2) minimal -- as small and simple a language as possible, and (3) a meta-language -- able to imitate other languages. The language is higher-order in that variables can represent both predicate names and entire predicates. The language also has "string variables", which match zero or more expressions in a sequence.

## 1 Introduction

With current programming languages one writes programs with efficiency in mind. Even in the logic programming field, where we supposedly "tell the computer what to do without telling it how to do it", the programmer must understand the execution process in order to write clauses that terminate and are efficient. Program transformation potentially offers a way of achieving the goal of declarative programming in which one can write nice specifications without considering their implementation and then have them transformed into efficient algorithms. One may thus ask what kind of programming language would we want if we could ignore efficiency and assume that a smart translator could transform our specifications.

This paper describes a previously-proposed logic programming language, called "axiomatic language" [Wil82], that is intended as just such a programming/specification language. The objectives of the language are as follows:

(1) pure specification language -- The programmer writes specifications while ignoring efficiency and implementation difficulty. Program transformation is thus required. We assume that a smart translator can be built that can transform the specifications into efficient programs.

(2) minimal, but extensible -- The language should be as small as possible with semantics as simple as possible. Nothing is built-in that can be defined. This means that arithmetic would not be built-in. (We assume the smart translator can "recognize" the abstract definitions of arithmetic and replace them with the corresponding hardware operations.) Of course, any minimal language must also be highly extensible, so that those features which are not built-in can be easily defined and used for specification as if they had been built-in.

(3) meta-language -- There are many ways in which one might want to specify software: first-order logic, set operations, functional programming, domain specific language, even procedurally, etc. An ideal language would be able to simulate within itself other languages and language paradigms, which the user could then use for software specification. With such a meta-language capability one could not only define functions, but define new ways of defining functions.

The two main features of this language in comparison to pure Prolog are its higher-order property, in which variables can represent predicates and predicate names, and string variables, which match zero or more expressions in a sequence. The higher-order property is essentially equivalent to that of HiLog [CKW93], but with some syntax differences. This language may also be unique for what it

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Series, Dept. of Computer Science, Univ. of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

does not have: no negation, no built-in functions, no built-in inequality, no assert/retract, and no input/output. The difficult problem of transforming specifications to efficient algorithms should be easier when the language is completely pure.

Section 2 defines axiomatic language and section 3 gives examples. A transformation for eliminating string variables is given in section 4. Section 5 gives conclusions.

## 2 The Language

This section defines axiomatic language, which is intended to fulfill the objectives given in the introduction. Section 2.1 introduces the language informally with examples. The core language is defined in section 2.2 and section 2.3 gives some syntax extensions.

### 2.1 Some Introductory Examples

As mentioned earlier, the two main differences of axiomatic language from Prolog are its higher-order property and string variables. The higher-order property comes from the fact that predicate and function names can be arbitrary expressions, including variables, and that variables can substitute for entire predicates in clauses. For example, the Prolog expression

```
father(bill,X)
```

is written in axiomatic language as

```
(father Bill %x)
```

where the predicate name is moved inside the parentheses and is just like any other term. Axiomatic language also has the syntax differences of replacing comma delimiters with blanks and having "expression" variables start with % (which permits uppercase letters to be used for symbols, as shown). Section 3 gives examples of higher-order definitions where variables are used for predicate names and entire predicates.

Clauses, or "axioms" as they are called here, are essentially the same as in traditional logic programming, as shown by the following familiar definitions of natural numbers and their addition and multiplication:

```
(number 0).                     ! zero is a number
(number (s %))< (number %).     ! successor of a number is a number

(plus 0 % %)< (number %).       ! addition of two natural numbers
(plus (s %1) %2 (s %3))<
  (plus %1 %2 %3).

(times 0 % 0)< (number %).      ! multiplication of two numbers
(times (s %1) %2 %3)<
  (times %1 %2 %x),
  (plus %x %2 %3).
```

Note that < replaces the Prolog symbol :- and comments start after !. These axioms generate expressions such as (number (s (s (s 0)))) and (plus (s 0) (s (s 0)) (s (s (s 0)))), representing the statements "3 is a number" and "1 + 2 = 3", respectively. In section 3 a decimal representation for numbers is defined.

The other major feature of axiomatic language is "string variables". A string variable, beginning with $, can match zero or more expressions in a sequence, while an expression variable (%) matches exactly one. String variables allow certain list predicates to be defined more concisely:

```
(append ($1) ($2) ($1 $2)).      ! concatenation of two lists
(member % ($1 % $2)).            ! member of a list

(reverse () ()).                 ! reverse a list
(reverse (% $) ($rev %))<
  (reverse ($) ($rev)).
```

These axioms generate expressions such as (append (a) (b c) (a b c)) and (member 3 (1 2 3)) and (reverse (u v) (v u)). Note that string variables can be considered a generalization of list "tail" variables in Prolog. For example, the Prolog list [a, X | Z] would be represented in axiomatic language as the sequence (a %x $z). Note also that sequences in axiomatic language are used both for lists and for term arguments.

## 2.2 The Core Language

This section defines the core of axiomatic language, with the next section giving some syntax extensions. In axiomatic language, functions and programs are specified by a finite set of "axioms", which generate a (usually) infinite set of "valid expressions", analogous to the way productions in a grammar generate strings or the way axioms in logic generate theorems. An *expression* is defined to be

an *atom* (a symbolic constant),
an *expression variable*,
or a *sequence* of zero or more expressions and *string variables*.

The hierarchical structure of expressions is adopted from the functional programming language FP [Bac78]. Atoms are represented by symbols that start with the backquote character: `abc, `X3, `+. (The non-variable symbols in section 2.1 are *not* atoms, as section 2.3 explains.) Expression and string variables are represented by symbols beginning with % and $, respectively: %expr, %, and $str, $1. A sequence is represented by a string of expressions and string variables, separated by blanks and enclosed in parentheses: (`M %y), (`xyz () $ (`)). It should be mentioned that these and other syntax details are somewhat arbitrary.

An *axiom* consists of a *conclusion* expression and zero or more *condition* expressions, in one of the following formats:

```
conclu < cond1, ..., condn.      ! n > 0
conclu.                          ! an "unconditional" axiom
```

Axioms may be written in free format over multiple lines and a comment may appear on the right side of a line, following an exclamation point. Note that the definition of expressions allows atoms and expression variables to be conclusion and condition expressions in axioms.

A finite set of axioms generates valid expressions by first generating an infinite set of *axiom instances* by the substitution of values for the expression and string variables. Each occurrence of an expression variable in an axiom can be replaced by an arbitrary expression, the same expression replacing the same variable throughout. Likewise, each string variable can be replaced by a string of zero or more expressions and string variables. For example, the axiom

```
(`A %x $str1 $str2)< (`B %x), ($str1 `C ($str2)).
```

has an instance

```
(`A (% `a) $w `*)< (`B (% `a)), ($w `* `C ()).
```

by the substitution of (% `a) for %x, '$w `*' for $str1, and '' (the null string) for $str2.

Axiom instances generate *valid expressions* by the simple rule that if all the conditions of an axiom instance are valid expressions, then 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 the valid expressions (`a `b), ((`a) `b `b), (((`a)) `b `b `b `b), etc. Note that valid expressions are just abstract symbolic expressions without any inherent meaning. Their association with real computation and inputs and outputs is by interpretation, as discussed in section 3.4.

### 2.3 Some Syntax Extensions

This section defines some useful syntax extensions for axiomatic language. The set of atoms is extended to include symbols consisting of single printable characters in single quotes: 'a', '3'. Note that these are abstract symbols and their association to actual characters is only by interpretation.

A character string in single quotes within a sequence is equivalent to the character atoms written separately:

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

A character string in double quotes is equivalent to a sequence of character atoms:

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

A symbol not beginning with ` ' " % or $ is syntactic shorthand for an expression consisting of the atom ` and the symbol as a character sequence:

```
abc  =  (` "abc")
```

This is useful for higher-order definitions and for defining inequality between symbols, which is not built-in.

## 3 Examples

This section gives some examples of axiomatic language and discusses its features. In section 3.1 we give examples of the higher-order property of the language. Section 3.2 discusses the meta-language capability of the language. Section 3.3 comments on those common logic-programming language features that are absent from this language. Finally, section 3.4 describes how programs can be defined by interpreting the set of generated valid expressions.

### 3.1 The Higher-Order Property

This section gives examples of the higher-order property of axiomatic language, in which variables can be used for predicate names and for entire predicates. The following examples show how higher-order constructs enable easier definition of sets and relations:

```
(%set %elem)< (finite_set %set ($1 %elem $2)).
     ! used to define finite sets:
(finite_set day (Sun Mon Tue Wed Thu Fri Sat)).
     ! generates valid expressions such as (day Tue)
```

```
% < (valid $1 % $2).
     ! specify multiple facts in one expression:
(valid (father Sue Tom) (father Bill Tom) (father Jane Bill)).

(one_valid $1 % $2)< % .
     ! at least one expression in list is valid:
(parent %x %y)< (one_valid (mother %x %y) (father %x %y)).
```

Axioms themselves can be represented as expressions:

```
(all_valid).
(all_valid % $)< % , (all_valid $).
     ! all expressions in list are valid

%conclu < (axiom %conclu $conds), (all_valid $conds).
     ! specify axiom in a single expression:
(axiom (sort %1 %2) (permutation %1 %2) (ordered %2)).

(axiom $axiom)< (axiom_set $1 ($axiom) $2).
     ! a set of axioms in a single expression:
(axiom_set ((length () 0))
           ((length (% $) (s %n)) (length ($) %n))).
```

This last axiom generates valid expressions such as (length (a b c) (s (s (s 0)))).

The following axioms make use of symbol representation to "map" functions and relations to sequences of arguments:

```
((` ($rel '*')) $nulls)<          ! empty sequences of arguments
  (zero_or_more () ($nulls)).
((` ($rel '*')) $argseqsx)<
  ((` ($rel '*')) $argseqs),
  ((` ($rel)) $args),
  (distr ($args) ($argseqs) ($argseqsx)).

(zero_or_more % ()).          ! zero or more copies of an expression
(zero_or_more % (% $))< (zero_or_more % ($)).

(distr () () ()).             ! distribute elements over sequences
(distr (%elem $elems) (($seq) $seqs) ((%elem $seq) $seqsx))<
  (distr ($elems) ($seqs) ($seqsx)).
```

These axioms generate valid expressions such as (day* (Sat Tue Tue)) and (append* ((a b) ()) ((c) (u v)) ((a b c) (u v))).

Axiomatic language's higher-order property is essentially the same as that of HiLog [CKW93]. That is, the language has higher-order syntax but the semantics are really first-order. (The XSB [XSB] implementation of HiLog, however, does not allow variables for head predicates in clauses [W,S00].)

### 3.2 The Meta-Language Property

The higher-order property plus string variables plus the absence of commas give axiomatic language its meta-language property -- its ability to imitate other languages. The original paper gave an example of a simple procedural language defined in axiomatic language. In this section we define the evaluation of LISP-like expressions. First we need decimal number representation:

```
(finite_set digit "0123456789").   ! set of decimal digits
(index %set %elem %index)<         ! index for each element in set
  (finite_set %set ($1 %elem $2)),
  (length ($1) %index).

(eval (` (%digit)) %value)<           ! single-digit symbol
  (index digit %digit %value).
(eval (` ($digits %digit)) %valuex)<  ! multi-digit symbol
  (eval (` ($digits)) %value),
  (index digit %digit %n),
  (times %value (s (s (s (s (s (s (s (s (s (s 0)))))))))) %10val),
  (plus %10val %n %valuex).
```

Now we define the evaluation of nested expressions:

```
(eval (quote %expr) %expr).      ! "quote" returns expr unchanged
(eval (%fn $args) %result)<
  (eval* ($args) ($results)),
  (%fn $results %result).
```

These axioms turn some of the previously-defined predicates into functions which can be applied in a LISP-like manner:

```
(eval (times (length (append (quote (a b c))
                             (reverse (quote (e d)))))
             (plus 3 15)
      ) %value)
```

When this expression is used as a condition, the variable %value will get instantiated to the natural number representation for 90.

### 3.3 What This Language Does Not Have

In this section we comment on those common logic programming features that axiomatic language does not have:

(1) non-logical operations -- No execution model, such as SLD resolution [Llo87], is implied by the language semantics, so non-logical operations such as "cut" have no meaning.

(2) built-in predicates -- We assume program transformation can map definitions of these predicates to efficient implementations, as if they had been built in.

(3) built-in inequality -- Inequality between symbols and expressions containing them must be explicitly defined, first by defining inequality between the character atoms, then inequality between character strings and symbols, and finally inequality between expressions of unequal elements.

(4) negation -- Negation must also be defined, either by explicit definition of negative predicates from inequality (such as a “not_member” predicate) or by representing axioms in encoded form and defining an SLD interpreter to generate negation as failure. Explicit definition of negation is a little tedious, but eliminating built-in negation is a great theoretical simplification for the language.

(5) assert/retract -- A set of axioms is static. Modifications to the set are done "outside" the language.

(6) input/output -- There are no input/output operations in the language. Instead, a program that does input/output is defined by interpreting the static set of valid expressions. This is described in the next section.

### 3.4 Defining Programs by Interpretation

This section describes how a real program that does input/output can be defined by a formal system that generates abstract symbolic expressions. As mentioned earlier, axioms and the valid expressions they generate are purely syntactic entities without any inherent meaning. However, they can be used to represent real computations by interpreting the set of valid expressions in some way.

A program that reads an input file and writes an output file can be defined by an infinite set of valid expressions of the form

```
(program <input> <output>)
```

where *<input>* and *<output>* are abstract symbolic expressions that are interpreted to represent actual input and output files, respectively. A text file, for example, might be represented by an expression which is a sequence of sub-expressions, each representing a line of text and consisting of a sequence of character atoms. These program valid expressions would be generated for all possible input files, and would give the corresponding output file for each input file. For example, a program that reads an input text file and writes an output file consisting of each line reversed could be defined by the following axiom:

```
(program %input %output)<
  (reverse* %input %output).    ! reverse each line in input file
```

This axiom, of course, generates program valid expressions for expressions other than text files, but those non-relevant expressions would be ignored.

An interactive program might be represented by a set of program valid expressions that each contain the sequence of inputs and outputs for an execution history. Valid expressions would be generated for all possible execution histories.

## 4 String Variable Elimination

String variables enable convenient pattern matching but they add a level of complexity in that two expressions with string variables may have multiple most-general unifications. For example, ($ a) and (a $) have an infinite number of most-general unifications with the substitutions $ = '' (the null string), $ = 'a', $ = 'a a', $ = 'a a a', etc. In this section we give a simple transformation that eliminates the problem of multiple unifications.

Multiple most-general unifications are caused by "non-tail" string variables -- those that do not occur at the ends of sequences. (In ($ a) the $ is non-tail while in (a $) the $ is "tail".) Note that if only tail string variables are present, unification is analogous to unification of Prolog lists and yields just a single most-general result. We want to eliminate occurrences of non-tail string variables, which have the following form,

```
... (..x.. $1 ..y..) ...
```

where ..x.. is a string of zero or more expressions and ..y.. is a string of one or more expressions and string variables. ($1 is therefore the first string variable and is non-tail.) For each occurrence of such a sub-expression in either a conclusion or condition of an axiom, we replace the string '$1 ..y..' with a new string variable as follows,

```
... (..x.. $1x) ...
```

and add the following condition to the axiom:

```
(APPEND ($1) (..y..) ($1x))
```

If the ..y.. string also contains non-tail string variables, the transformation step is applied recursively to this new condition. Finally we add to the set of axioms the following classic definition of the APPEND predicate, which uses only tail string variables and is distinct from the append predicate defined earlier:

```
(APPEND () ($2) ($2)).
(APPEND (% $1) %2 (% $3))<
  (APPEND ($1) %2 ($3)).
```

The set of generated valid expressions will be the same after this transformation, except for the addition of the APPEND predicate and the elimination of non-tail string variables.

A final transformation is to represent sequences as nested pairs, similar to the way lists are represented in Prolog, in which case tail string variables can be treated as ordinary expression variables. Details are omitted.

## 5 Conclusions

A type of logic programming language intended as a pure specification language has been described. The programmer is expected to write specifications without concern about efficiency. Higher-order definitions and the meta-language capability should provide powerful tools for software specification, but make program transformation essential. The difficult problem of transformation should be helped, however, by the simplicity and purity of the language, such as the absence of non-logical operations, negation, built-in functions, state changes, and input/output. Axiomatic language should thus be well-suited to transformation methods such as unfold/fold [e.g., PP99]. Future work will address the problem of transformation.

## Acknowledgements

Carl Bruggeman of UTA gave much helpful advice. Luis Tovar, D. R. Richardson, and Echo Wilson provided support. Lockheed Martin and Computer Sciences Corporation provided funding.

## References

[Bac78] J. Backus, "Can programming be liberated from the von Neumann style?", Comm. ACM, 21(8):613-641, Aug. 1978.

[CKW93] W. Chen, M. Kifer, & D.S. Warren, "HiLog: A foundation for higher-order logic programming", J. of Logic Programming, 15(3):187-230, Feb. 1993.

[Llo87] J.W. Lloyd, Foundations of Logic Programming (Second Edition), Springer Verlag, 1987.

[PP99] A. Pettorossi & M. Proietti, "Synthesis and transformation of logic programs using unfold/fold proofs", J. of Logic Programming, 41(2&3): 197-230, Nov. 1999.

[W,S00] D.S. Warren, K. Sagonas, private communication, 2000.

[Wil82] W.W. Wilson, "Beyond PROLOG: software specification by grammar", SIGPLAN Notices, 17(9):34-43, Sept. 1982. (at <http://cseweb.uta.edu/~wwwilson>)

[XSB] The XSB Research Group, <http://www.cs.sunysb.edu/~sbprolog/index.html>.