# Prolog/V

By Mike Teng. Port to Squeak by Bolot Kerimbaev bolot@cc.gatech.edu.
This document describes original Prolog/V, as well as comments on differences introduced by the Squeak port.

# 1. INTRODUCTION

This file describes how to use Prolog/V, the version of Prolog that you can easily add to your Smalltalk/V environment. It also explains the syntactical differences between it and standard Prolog, that is, the Prolog language defined by Clocksin and Mellish. Knowledge of Prolog and Smalltalk is assumed, so this is not an introduction to either language.

The emphasis in this implementation is placed on the integration of Smalltalk and Prolog. In the Smalltalk environment, you can send a series of Prolog/V goals to an expert system (an instance of class Prolog or one of its subclasses) from any Smalltalk method. After doing its best to satisfy these goals, Prolog/V returns an array of all solutions which can then be further processed by the Smalltalk code.

From the Prolog side, you can send a Smalltalk message from the 'is' predicate and unify the answer of the message with a Prolog variable. This makes it convenient to use Smalltalk to handle the cursor, to manipulate windows, to perform graphics, to do file I-O, and of course to do object-oriented programming with your own methods. (Therefore, there are no primitives defined in Prolog/V to handle these functions.) You can also implement Prolog predicates using Smalltalk code (see Section 5, Implement Your Own Built-in Predicates). In addition, the class inheritance and polymorphism features of Smalltalk automatically apply to the Prolog/V environment.

As a result of this emphasis, you will find that many of the differences between Prolog/V and standard Prolog are due to the fact that we have chosen a more Smalltalk-like syntax in order to provide better communication between the two languages. The differences are summarized in Section 8.

# 2. ENVIRONMENT

## *Start-up*

To install Prolog, copy the file 'prolog.st' to your current directory and execute the following command using 'do it':

```
        (File pathName: 'prolog.st') fileIn; close
Squeak: "file in prolog.[timestamp].cs file from file list"
```

Note that since the pathName specified above is a relative one, the file must be in the current directory. This file contains all the classes and methods needed by the Prolog/V environment as well as three example classes, Family, Doctor and Dungeon.

## *Entering a Prolog Program*

A Prolog program can be viewed as an expert system. Unlike other Prolog environments that run one expert system at a time, Prolog/V allows multiple expert systems to exist at the same time and consult each other. To add knowledge to an expert system, you first define the classes needed by the expert system and then enter Prolog clauses (facts and rules) into the classes in the same way as you implement Smalltalk methods in Smalltalk classes. The only difference is that you have to enter facts and rules through a Logic Browser window instead of a Class Hierarchy Browser window. The reason is that the Logic Browser has a Prolog compiler associated with it while the Class Hierarchy Browser has a Smalltalk compiler associated with it. However, you can post Prolog questions in either window or in any other Text pane, for example, the pane in a workspace window.

To open a Logic Browser window, evaluate the following Smalltalk expression in any Text pane:

```
        LogicBrowser new open
Squeak: "use regular System Browser, compiler is attached to class, rather than browser"
```

You will be prompted for the frame of the window. After selecting the top left corner and bottom right corner of the window, you will see a window with three panes.

The top left pane contains the class Prolog and its subclasses. Select class Family and the names of all the clauses (corresponding to Smalltalk methods) defined in that class will appear in the top right pane. If you select one of these names, the rules and facts having this name will appear in the bottom pane. You can edit the contents of the bottom pane as you edit a Smalltalk method. Selecting the 'save' choice on the pane menu will cause the clauses to be compiled into Smalltalk internal code and saved in the system.

In short, a Prolog/V program is represented by a group of classes. Each class is broken up into methods. Each method contains clauses with the same name which are edited and saved as a group because they are actually compiled into one Smalltalk method.

### *Questions & Answers*

The Prolog/V environment is fully integrated with the Smalltalk/V environment. You can pose Prolog/V questions in any text pane using 'show it'. For example, if you want to know who is the father of Mary, execute the following:

```
        Family new :? father(x, 'Mary')
Squeak: LCompiler evaluate: 'Family new :? father(x, ''Mary'')'
```

where 'father' is a method in class Family. Prolog/V responds with:

```
        (('John'))
```

This is a Smalltalk Array of all solutions to the goals, which you can further process using Smalltalk code. You can embed a Prolog question in a Smalltalk method. For example,

```
        (Family new :? father('John', x))
        do: [ :eachAnswer |
                Transcript nextPutAll: (eachAnswer at: 1), ' is a child of John'; cr]
```

prints all the children of John in the System Transcript. See Questions & Answers in the LANGUAGE Section for a more detailed description.

## 3. PROLOG/V LANGUAGE

### *Constants*

Prolog atoms are represented by Smalltalk/V Symbols. Thus they must start with a number sign (#) and a letter, followed by zero or more letters and digits, for example:

```
    #Mary     #Mary123     #abc
```

Standard Prolog strings enclosed in double quotes are represented by Smalltalk strings enclosed in single quotes, for example:

```
    'Mary'     'Mary is a girl'
```

Notice that these strings are not stored in lists of ASCII codes as in standard Prolog. Their use is almost as efficient as Symbols.

Prolog/V integers are the same as Smalltalk integers, which can be more than 100K digits long. You can also use other Smalltalk constants in Prolog/V, for example:

```
        16rFF           An integer FF with radix 16.
        1.23            A floating number if you have a floating coprocessor, otherwise illegal.
        #(1 3)          An array of constants.
        $A              A character
        true/false      Pseudo variables representing Boolean true/false.
```

Please refer to your Smalltalk/V Owner's Manual for a further description of constants.

### *Variables*

Unlike standard Prolog variables which always start with an upper case letter, Prolog/V variables start with either an upper case letter (global variable) or a lower case letter (local variable). Some examples of legal variables are:

```
        x    aVariable    fatherOfSomebody    x123    Cursor
```

An underscore '_' is not allowed in a variable name. However, an underscore all by itself can be used as a don't care variable.

### *Structures*

Structures are similar to standard Prolog except that a structure with no arguments must have a pair of parentheses following the name. Some examples are:

```
        owns(#Barbara, 'wuthering heights')    nullArg()
```

Structures in Prolog/V are represented as instances of class Relation, which is a subclass of List. Its functor is a Symbol and its components can be variables, literal objects or another structure.

### *Lists*

In Prolog/V, lists have the same syntax as in standard Prolog, for example:

```
    []    [y, #John, 'a string', #(1 2)]    [[1,2], 3 | [4,5]]
```

However, its internal structure does not contain a dot, '.', as its functor. The functor of a list is its first element. A list is represented as an instance of class List. The internal representations of a list and a structure are exactly the same. They are distinguished by their classes: List and Relation respectively.

### *Predicate is*

In Prolog/V, you have full access to Smalltalk programming. The Prolog predicate 'is' is provided as the bridge between the two worlds. For example, when

```
is(x, Array with: 'Hi' with: 'Prolog')
```
is evaluated as a Prolog goal, its second argument (starting with 'Array') will be evaluated as a Smalltalk expression. The result of the evaluation is an Array with two elements, 'Hi' and 'Prolog', which will then be unified with the Prolog variable 'x'.

However, when you use a Prolog variable in a Smalltalk expression, you must send the message 'value' to the Prolog variable before you use it in the Smalltalk message. For example,

```
factorial(x, y) :- is(y, x value factorial)
```
This is because a Prolog variable is always an instance of classLogicRef whose contents is returned by the 'value' message.

## *Arithmetic & Comparisons*

Arithmetic is performed in Prolog/V via Smalltalk expressions using the 'is' predicate. Therefore, arithmetic expressions in the second argument must obey Smalltalk syntax rules. The notable difference in syntax between Smalltalk expressions and standard Prolog expressions is that in Smalltalk all arithmetic operators have the same precedence. For example,

```
is(x, 2 + 3 * 4)
```
will unify 'x' with 24 instead of 14. You must use parentheses to alter the left to right evaluation order. For example:

```
is(y, 2 + (3 * 4))
```
unifies y with 14 in both Prolog/V and standard Prolog.

Two atomic terms can be compared directly as follows:

```
lt(2, length)          gt(name, 'John')          eq(#apple, fruit)
```
The 'eq' predicate has the extra effect of unifying the two arguments if one of them is unbound.

To compare non-atomic expressions, the expressions must be evaluated outside of the comparison operation as follows:

```
is(temp1, 1 + 2),          is(temp2, 6 // 2),          eq(temp1, temp2)
```
which will unify 'temp1' with 'temp2' successfully.

## *Facts*

A fact is a known relationship between objects. Its syntax is the same as a structure. It is normally terminated with a period in order to separate it from other facts and rules. Following are some examples:

```
father(#John, #Mary).    factorial(1, 1).    append([], x, x).    member(x, [x | _]).    thereIsOneSun().
```
In the first example above, the symbol 'father' is called a predicate, and '#John' and '#Mary' are arguments of the predicate. The last example is a fact without arguments. Notice that a pair of empty parentheses are needed at the end.

## *Rules*

A rule states the conditions for a fact to be true. Its syntax consists of the fact relationship and a series of goal relationships separated by a ':-' operator. For example:

```
append([head | l1], l2, [head | l3]) :- append(l1, l2, l3).
grandPa(x, y) :- father(x, z), father(z, y).
```
The right hand side relationships are called goals because they are goals to be satisfied in order to prove that the left-hand side relationship is indeed a fact.

## *Questions & Answers*

In Prolog/V, a question consists of a hybrid expression: Smalltalk on the left, the question operator ':?', and Prolog on the right. The left part of the question is a Smalltalk expression which is evaluated to determine the receiver of the question, an instance of a Prolog class. The right part of the question is a series of goals to be satisfied by the receiver expert system. The right part of a question has the same syntax as the right part of a Prolog rule.

Questions can be embedded in a Smalltalk method or executed in any text pane by selecting 'do it' or 'show it' from the pane menu.

The answer from executing a question is either 'nil' or an array of arrays. If it is nil, it means that there are no solutions. Otherwise, an array of all solutions is returned. Each solution is, in turn, an array whose elements are the answers unified with each unbound variable contained in the question. The order of the elements in the inner array corresponds to the order of those variables appearing in the question. For example, class Family (a subclass of Prolog) has a 'grandPa:' method as follows:

```
grandPa(x, y) :- father(x, z), OR(father(z, y), mother(z, y)).
Squeak: Currently, you must use OR and AND because #or: and #and: are treated as special case
```
This reads: x is the grandpa of y if x is the father of z and either z is the father of y or z is the mother of y.

Class Family also has the following 'father:' and 'mother:' methods:

```
father('John', 'Mary'). father('John', 'David'). father('David', 'Jack'). father('Arthur', 'Nancy').
mother('Mary', 'Nancy').
```
Try asking the following question:

```
        Family new :? grandPa('John', x)
Squeak: LCompiler evaluate: 'Family new :? grandPa(''John'', x)'
```
Prolog/V answers:

```
          (('Nancy') ('Jack'))
```
which is an array containing two other arrays meaning that there are two solutions to the question. That is, 'John' has two children, 'Nancy' and 'Jack'.

Ask the following question:

```
        Family new :? grandPa('John', x), father(y, x).
Squeak: LCompiler evaluate: 'Family new :? grandPa(''John'', x), father(y, x).'
```
Prolog/V replies:

```
        (('Nancy' 'Arthur') ('Jack' 'David'))
```
Here again, there are two solutions. The first solution says that 'John' is the grandPa of 'Nancy' (variable x) whose father is 'Arthur' (variable y). The second solution says that 'John' is also the grandPa of 'Jack' (x) whose father is 'David' (y).

Now ask the question:

```
        Family new :? father('John', 'Mary')
Squeak: LCompiler evaluate: 'Family new :? father(''John'', 'Mary')'
```
The answer is (()). This is because there is one solution but there are no unbound variables in the question.

## *Methods*

In Prolog/V, all clauses in a class having the same name are grouped together and compiled as one unit which is called a method. The name of a clause is the name of its leftmost relationship which is in turn the name of its leftmost predicate. The name of a Prolog method is the name of its clauses with a colon, ':', at the end, for example, 'father:'. This is because a Prolog method is always compiled into a Smalltalk method having one argument.

The name of a Prolog method must be unique in its class. But different classes can implement methods with the same name. Please refer to the Smalltalk/V Owner's Manual for information on the polymorphism feature of Smalltalk.

## *Classes*

Logic is a subclass of class Object. It contains methods like #unify:with:then: that makes the execution of Prolog methods possible. Prolog is a subclass of Logic. It implements all the built-in predicates. Thus all the Prolog programs should be implemented as a hierarchy of classes under class Prolog in order to inherit built-in predicates and the execution mechanism.

As in Smalltalk, a Prolog/V subclass inherits the predicates implemented in its superclasses. Communication among parallel classes is also possible. As stated before, each Prolog class can be viewed as an expert system in some field. There are times an expert system needs to consult another expert system to solve problems not in its field. The built-in predicate, 'consult', is implemented differently in Prolog/V than in standard Prolog to serve this need. For example, in class Family, a consultation from class Doctor may be used to get the health record of a member of the family:

```
        consult( healthHistory('John', x), Doctor new)
```
In this goal, the second argument, 'Doctor new', creates an instance of class Doctor which is going to render the consultation. The first argument is a goal to be solved by the consultant. In this case, the health history of John will be unified with the variable, 'x'. Evaluate the following

```
        Family new :? health('John', y).
Squeak: LCompiler evaluate: 'Family new :? health(''John'', y).'
```
You will get a health record of 'John'.

## *Database*

A Prolog/V database contains facts which can be dynamically asserted or retracted. All databases are global in the sense that once a database is created, it can be accessed by an instance of any Prolog class. A database is created by the 'database' predicate. For example, execute the following:

```
        Family new :? database(brother()).
Squeak: LCompiler evaluate: 'Family new :? database(brother()).'
```
You have just created a 'brother' database. The argument for the database predicate must be a structure. Since a fact allows a variable number of arguments, when you create a database, you do not need to specify them. Any arguments specified are ignored by the 'database' predicate and are, therefore, for documentation only.

The Prolog/V database is implemented using a dictionary contained in the class variable, Database, of class Logic. The keys of the dictionary are the names of the facts. The value of each key is a list of facts with the same name.

The standard 'asserta', 'assertz', and 'retract' predicates can be used to add a fact to the front of its list, add a fact to the end of its list, or delete any fact from its list, respectively. Note that the facts are not backed out or restored during backtracking.

Execute the following example:

```
        Family new :? assertz(brother('John', 'Stewart')),
                      asserta(brother('John', 'Barbara')), assertz(brother('John', 'Alan')).
Squeak: LCompiler evaluate: 'Family new :? assertz(brother(''John'', ''Stewart'')),
                      asserta(brother(''John'', ''Barbara'')), assertz(brother(''John'', ''Alan'')).'
```

This inserts three facts in the 'brother' database, the second fact before the first one, and the third one at the end.

To remove or retract a fact, for example, to remove John as a brother of Stewart, execute the following:

```
        Family new :? retract(brother('John', 'Stewart')).
Squeak: LCompiler evaluate: 'Family new :? retract(brother(''John'', 'Stewart')).'
```

To retrieve all of the facts in the database 'brother', execute the following:

```
        Family new :? brother(x,y).
Squeak: LCompiler evaluate: 'Family new :? brother(x,y).'
```

Prolog/V responds:

```
        (('John' 'Barbara') ('John' 'Stewart') ('John' 'Alan'))
```

To see all of the databases created in the system, use the Smalltalk inspector to examine the dictionary by executing the following expression:

```
        Prolog database inspect
```

Notice that an inspector window pops up with a list of database entries. Select the entry that you are interested in and you will see its contents shown on the right as a list of what has been asserted.

To purge a database, use for example:

```
        Family new :? purgeDatabase(brother()).
Squeak: LCompiler evaluate: 'Family new :? purgeDatabase(brother()).'
```

When you save the Smalltalk image, all the databases created up to this point will stay in the system.

### *Comments*

Comments have standard Smalltalk syntax: a string enclosed in double quotes ("). A comment can be placed anywhere.

## 4. BUILT-IN PREDICATES

The following predicates are built-in for the Prolog/V environment. Nonstandard predicates are indicated.

### *Success and Failure*

| | |
|---|---|
| fail() | Fail always. |
| true() | Succeed once always, i.e. fail during backtracking. |

### *Classifying Terms*

| | |
|---|---|
| atom(aTerm) | Succeed if aTerm is a Symbol. |
| atomic(aTerm) | Succeed if aTerm is a Symbol or is a kind of Integer. |
| integer(aTerm) | Succeed if aTerm is an Integer. |
| nonvar(aTerm) | Succeed if aTerm is not an unbound variable. |
| var(aTerm) | Succeed if aTerm is an unbound variable. |

### *Creating and Accessing Structures*

| | |
|---|---|
| arg(n, aStructure, x) | Bind x to the n'th argument of aStructure. |
| functor(aStructure, head, arguments) | If aStructure is bound, unify its functor with head, and its number of components with arguments. If aStructure is unbound, unify it to a structure with head as its functor and arguments number of don't care variables as its components. |
| univ(aStructure, aList) | Convert aStructure to a list if aStructure is bound. Convert aList to a structure otherwise. This is equivalent to the standard Prolog '=..' operator. |

### *Affecting Backtracking*

| | |
|---|---|
| ! | The cut predicate succeeds always; exit the containing method during backtracking. |
| exit() | The exit predicate succeeds always; exit the entire Prolog query. Nonstandard. |

### *Creating Complex Goals*

| | |
|---|---|
| structure1 , structure2 | The comma predicate succeeds when structure1 and structure2 both succeed. |
| and(structure1, structure2, ...) | Succeed when all structure arguments succeed. Nonstandard. |
| call(aStructure) | Execute aStructure as a goal. |
| not(aStructure) | Succeed if the evaluation of aStructure as a goal fails and fail otherwise. |
| or(structure1, structure2) | Succeed if the evaluation of either structure1 or structure2 |

| | |
|---|---|
| | succeeds, fail when both succeed. This is equivalent to the standard Prolog ';' operator. |
| repeat() | Succeed always, even during backtracking. |

### *Comparing Terms*

| | |
|---|---|
| eq(term1, term2) | Succeed if the two terms can be unified. This is equivalent to the standard Prolog '=' operator. |
| equiv(term1, term2) | Succeed if the two terms are the same object. This is equivalent to the standard Prolog '==' operator. |
| ne(term1, term2) | Succeed if term1 is not equal to term2. This is equivalent to the standard Prolog '\=' operator. |
| notequiv(term1, term2) | Succeed if the two terms are not the same object. This is equivalent to the standard Prolog '\==' operator. |
| ge(term1, term2) | Succeed if term1 is greater than or equal to term2. This is equivalent to the standard Prolog '>=' operator. |
| gt(term1, term2) | Succeed if term1 is greater than term2. This is equivalent to the standard Prolog '>' operator. |
| le(term1, term2) | Succeed if term1 is less than or equal to term2. This is equivalent to the standard Prolog '<=' operator. |
| lt(term1, term2) | Succeed if term1 is less than term2. This is equivalent to the standard Prolog '<' operator. |

### *Database*

| | |
|---|---|
| database(aStructure) | Create a database which can hold facts to be asserted or extracted. The name of the database is taken from the functor of aStructure. The arguments of aStructure are ignored. |
| purgeDatabase(aStructure) | Purge a database identified by the functor of aStructure. The arguments of aStructure are ignored. |
| asserta(aFact) | Add aFact to the beginning of the database identified by the functor of the fact. The database must have been previously created by the 'database' predicate. |
| assertz(aFact) | Add aFact to the end of the database identified by the functor of the fact. The database must have been previously created by the 'database' predicate. |
| retract(aStructure) | Delete the first fact unified with aStructure from the database that has the same name as the functor of aStructure. Delete the next unified fact during backtrack. |

### *Input-Output*

| | |
|---|---|
| nl() | Write an end of line to Transcript. |
| read(aTerm) | A Smalltalk Prompter is invoked to evaluate a Smalltalk expression. The result is unified with aTerm. This can be used to read a symbol (beginning with #), a number (integer or real), a string (enclosed with single quotes), etc. Nonstandard. |
| write(term1, term2, ...) | Write all the arguments to Transcript. |

### *Debugging*

| | |
|---|---|
| halt() | Halt the execution. The Smalltalk debugger can be invoked at this point to examine the stacked method evaluations. To modify Prolog methods, you must use the Logic Browser window. The resume option on the walkback window can be used as a trace facility. Nonstandard. |

### *Communicating with Smalltalk*

| | |
|---|---|
| consult(aStructure, aSmalltalkExpression) | Use the result of evaluating aSmalltalkExpression as the receiver and ask it to solve aStructure as a goal. Nonstandard. |
| is(aTerm, aSmalltalkExpression) | Evaluate aSmalltalkExpression and then unify its result with aTerm. This Smalltalk expression can be used to do graphics, windows, pattern matching, animation, dictionary lookup, file I/O, etc. Nonstandard. |
| runtime(aTerm) | Unify aTerm with the system time in milliseconds. Nonstandard. |

## 5. IMPLEMENT YOUR OWN BUILT-IN PREDICATES

As in the Smalltalk language, Prolog built-in predicates can be easily extended. All the source code for Prolog/V built-in predicates are included in class Prolog. By examining the source code for the predicates, you can get a good understanding of how they are implemented. You can use them as models to implement additional predicates, or you can modify the existing ones to suit your special needs.

Every Prolog method has exactly one argument, no matter how many arguments there are in the Prolog clauses. This argument is an instance of class Association. The key of the Association contains the List of arguments to be unified with each clause in the method. The value part of the Association contains a block of code, called continuation block, to be evaluated when the predicate succeeds. If the predicate fails, it simply returns the receiver itself.

Most of the built-in predicates succeed only once. In other words, they do not provide an alternative choice during backtracking. These predicates simply return the continuation block instead of evaluating it after success. By doing this, the predicate will not occupy any extra stack space.

# 6. DUNGEON EXAMPLE

If you have installed the 'prolog.st' file and opened a Logic Browser, you can browse the code contained in class Dungeon. This example ties the Smalltalk graphics drawing and animation together with the Prolog/V inferencing power. To run this example, evaluate the following in any text pane:

```
Dungeon new :? go()
Squeak: Dungeon example is not fully implemented
```

After initialization, a map of the dungeon is shown, and a dog tries to go from entry to exit and finds the gold treasure. It turns back when one of the dangerous rooms is encountered.

The speed of the dog can be changed in the 'initialize:' method. The dangerous rooms is also set in the same method. The map of the dungeon can be altered by changing the 'gallery:' and 'position:' predicates.

# 7. LIMITATIONS

Since Prolog methods are compiled into Smalltalk internal code, they share the same execution stack as Smalltalk methods. The default size of this stack is 2K entries. Each execution of a Prolog method takes approximately 20 entries. As you can see, a recursive Prolog method can very quickly exaust the stack space. One remedy for this problem is to increase the stack space. To do this, exit Smalltalk/V and enter the following command to DOS:

```
set STACK=XX
```

where XX can be 01 to 0A representing 1K up to 10K entries. Please refer to the Smalltalk/V Owners Manual for details.

Note that Prolog/V has a few optimizations which cut down the use of the stack. For example, the last clause of every method does not create a stack entry since there are no more alternatives after the last clause.

Squeak Note: some Prolog methods cannot be compiled because they exceed the allocated stack size (for literals and temps)

# 8. MAINTAINING PROLOG PROGRAMS

Each time you save a Prolog method in the Logic Browser window, the method is logged in the 'change.log' file. You can use the Disk Browser window to recover the source by selecting the source in the 'change.log' file and execute the 'install' option. You can also use the file out option in the class list pane of the Logic Browser window to write a whole Prolog class to a file which can later be installed on another system or be used for recovery purposes.

# 9. DEVIATIONS FROM STANDARD PROLOG

The rules for converting standard Prolog to Prolog/V are:

| Standard Prolog | Prolog/V | Comments |
|---|---|---|
| prolog_name | prologName | No underscore allowed in name. |
| PrologObjectName | #prologObjectName | Atoms must be preceded with #. |
| PrologVariableName | prologVariableName | Local variable name starts with lowercase letter. |
| /* comment */ | "comment" | Comments are surrounded by double quotes. |
| '\digits' | | Illegal |
| '\letter' | | Illegal |
| 'character' | $character | Characters must be preceded by $. |
| "string" | 'string' | Strings are surrounded by single quotes. |
| ?- goal(s) | receiver :? goal(s) | Questions require a receiver which is separated from goals by :?. |
| a > b | gt(a,b) | No binary comparison operators allowed. |
| a >= b | ge(a,b) | |
| a < b | lt(a,b) | |
| a <= b | le(a,b) | |
| a = b | eq(a,b) | |
| a <> b | ne(a,b) | |
| is(A, prologExp) | is(a,smalltalkExp) | The second argument must be a Smalltalk expression. |
| RelationNameNoArgs | relationNameNoArgs() | A pair of parentheses is required. |
| aStruct =.. aList | univ(aStruct, aList) | |
| structA ; structB | or(structA, structB) | |

In addition, the semantics of read and consult are nonstandard. Finally, the following standard Prolog predicates are not implemented in Prolog/V: clause, debugging, display, get0, get, listing, name, nodebug, nospy, notrace, op, put, reconsult, see, seeing, seen, skip, spy, tab, tell, telling, told, trace. Notice that most of these are debugging and I/O primitives, and their equivalents are available in Smalltalk/V.

# 10. BNF SYNTAX OF PROLOG/V

```
prologQuestion =
    expression ':?' goalSeries ['.'].
prologMethod =
    clause '.' [clause '.'].
clause =
    fact | rule.
fact =
    relation.
rule =
    relation ':-' goalSeries.
goalSeries =
    goal [',' goal].
goal =
    cutPredicate | smalltalkPredicate |
    relation.
relation =
    name '(' termSeries ')'.
termSeries =
    [term [',' term]].
term =
    literal | list | dontCareVariable |
    variableName | andPredicate | structure.
list =
    '[' [term [',' term]] ']' |
    '[' term [',' term] '|' term ']'.
dontCareVariable =
    '_'.
andPredicate =
    '(', termSeries, ')'.
structure =
    relation.
cutPredicate =
    '!'.
smalltalkPredicate =
    isPredicate | consultPredicate.
isPredicate =
    'is' mixedArguments.
consultPredicate =
    'consult' mixedArguments.
mixedArguments =
    '(' term ',' expression ')'.
expression =
    *** See Appendix 1, Smalltalk Syntax Summary ***
variableName =
    *** See Appendix 1, Smalltalk Syntax Summary ***
literal =
    *** See Appendix 1, Smalltalk Syntax Summary ***
name =
    lowerCaseLetter {letter | digit}.
lowerCaseLetter =
    'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
    'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' |
    'u' | 'v' | 'w' | 'x' | 'y' | 'z'.
letter =
    *** See Appendix 1, Smalltalk Syntax Summary ***
digit =
    *** See Appendix 1, Smalltalk Syntax Summary ***
```