Prolog.NET

Version 1

Richard G. Todd
code@richtodd.com

http://prolog.codeplex.com

Contents

Introduction 2

Hello, World! 2

Obtaining Prolog.NET 2

Prolog.NET Workbench 4

Transcript View 5

Debug View 5

Program View 6

Trace View 6

Architecture and Design 7

Domains 7

The Standard Library 9

Data Types 9

Operators 10

Predicates 10

Functions 11

Summary 11

Reference 14

Grammar 22

Bibliography 27

Introduction

Prolog.NET is a CLI-based (ECMA International 2006) Prolog interpreter based on the Warren Abstract Machine (WAM) architecture.

The original paper describing what would become known as the Warren Abstract Machine was published by David H. D. Warren in 1983 (Warren 1983). A book clarifying and expanding on the details in this paper was published by Hassan Aït-Kaci in 1991 (Aït-Kaci 1991).

Prolog.NET deviates from the standard WAM architecture in a few important respects. Most significantly, it relies on the garbage collection support provided by the CLI. No explicit memory management is performed by Prolog.NET. All variables, environments and choice points reside in the CLI heap.[1] To ensure objects in the heap can be reclaimed in a timely fashion, Prolog.NET must still respect the context of variables when binding variables to other variables and properly "unwind" variable bindings during backtracking.

In the original WAM architecture, the terms "variable" and "value" in opcode names refer to unbound and bound variables, respectively. Prolog.NET uses the terms "unbound variable" and "bound variable". Further, the WAM architecture uses the term "constant" to refer to integers, strings and other extra-logical objects.[2] Prolog.NET reserves instead the term "value" for this use.

Prolog.NET can be called directly from client applications. A WPF-based IDE is also supplied.

Hello, World!

The following code demonstrates the use of Prolog.NET from client code:

```
CodeSentence codeSentence;

codeSentence = Parser.Parse("hello(world)");
Program program = new Program();
program.Add(codeSentence);

codeSentence = Parser.Parse(":-hello(X)");
Query query = new Query(codeSentence);

PrologMachine machine = PrologMachine.Create(program, query);
ExecutionResults results = machine.Run();
```

Obtaining Prolog.NET

Prolog.NET is available at http://prolog.codeplex.com.

The following projects are included:

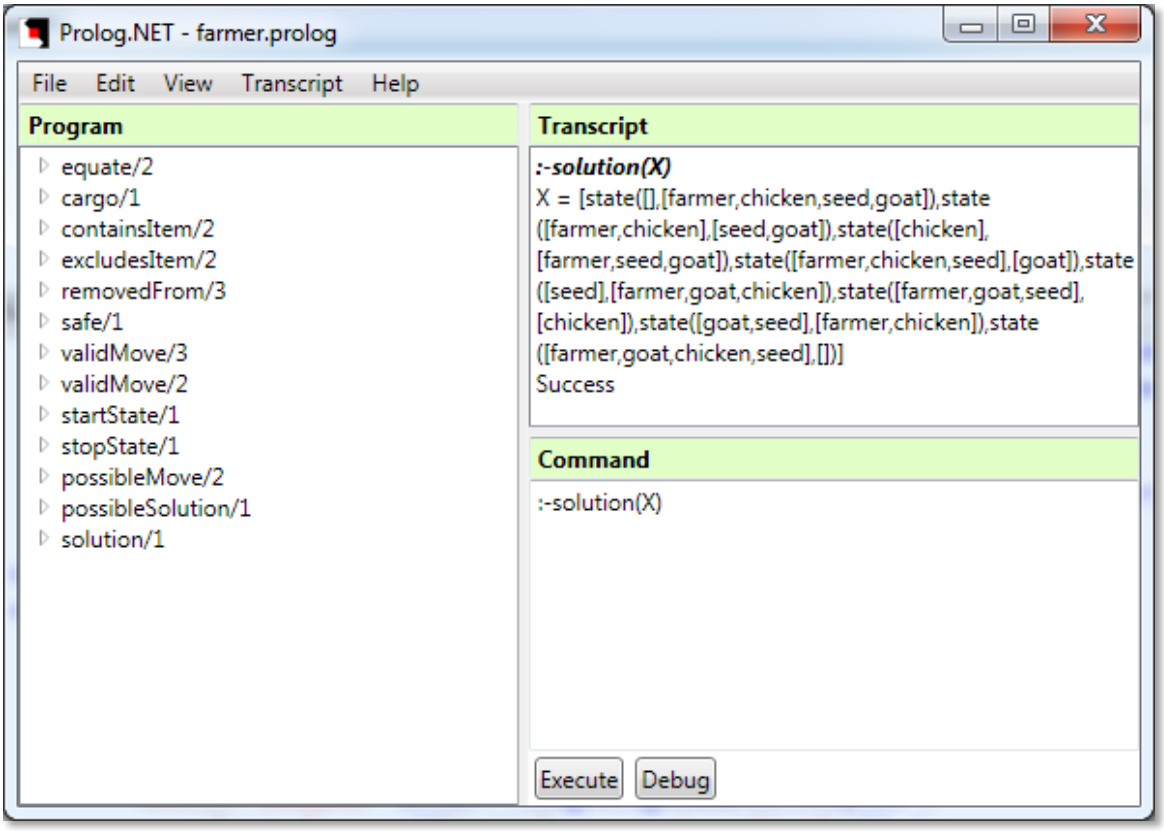
- Prolog: the primary assembly containing the Prolog.NET compiler and interpreter.
- PrologWorkbench: a WPF application used for editing, running and debugging Prolog.NET programs.
- PrologTest: a console application used for testing.

- PrologSchedule: a WPF application demonstrating the use of Prolog.NET from a client C# application.
- PrologLibrary: an assembly containing external functions callable by Prolog.NET programs.
- PrologWorkbenchSetup: a deployment project for PrologWorkbench and other supplemental programs.

The parser used by Prolog.NET is implemented using Lingua.NET. The Lingua.NET runtime is distributed as part of the Prolog.NET package. More information about Lingua.NET is available at http://lingua.codeplex.com.

Prolog.NET Workbench

The Prolog.NET Workbench is a WPF-based application that allows Prolog.NET programs to be developed and tested.



The program has four views:

Transcript

Allows facts and rules to be created and edited, and queries to be executed. This is the initial view when the program is started.

Program

Allows the instruction streams associated with program clauses to be viewed.

Debug

Permits the runtime debugging of Prolog.NET programs.

Trace

Allows tracing information created during program execution to be viewed.

The View menu can be used to select the desired view.

Transcript View

The Transcript view allows facts and rules to be created and edited, and queries to be executed. This is the initial view when the program is started.

All Prolog statements are entered into the Command box. To process the statement, click the Execute button or press Ctrl-Return. Processing results are shown in the Transcript box. New facts and rules are automatically added to the Program tree.

It is only necessary to terminate statements with a period when entering multiple statements. For example:

```
color(red).
color(blue).
color(green).
```

Queries in Prolog.NET must be prefaced with the :- operator. For example:

```
:- color(C)
```

The success or failure of the query is shown in the Transcript box, along with the value of any variables in the query. To find additional solutions, select menu option Transcript | Continue or press F5.

Commands in the Transcript box can be retrieved by double-clicking them or using the Edit menu options.

To edit a program statement, double-click it to display it in the Command window. When the updated statement is executed, it will replace the currently highlighted program statement. If you wish to add, rather than replace, the modified statement to the program, first deselect the highlighted statement by selecting the parent procedure node in the Program tree.

Debug View

The Debug view allows the current execution statement of the program to be viewed. Debugging is started by entering a query in the Transcript view and selecting menu option Transcript | Debug or pressing Ctrl-Shift-Return.

The Call Stack list box displays the facts and rules in the active call stack. Call stack entries that are inactive but preserved by choice points are not shown. Call stack entries based on query statements are listed as Anonymous.

Note that the WAM run time does not maintain an explicit run-time call stack. It is derived from the current chain of environment structures and the current state of the instruction pointer registers.

When a call stack entry is selected, the corresponding WAM instruction stream is shown in the Instructions list box. If an environment is associated with the call stack entry, the Permanent Variables list box will display the current value of any variables in the environment.

The Temporary Variables and Arguments list boxes display the current values of those registers. Note that these registers are global and are not affected by the selected call stack entry.

The Debug menu contains various options for executing one or more WAM instructions:

- Run to Backtrack (F6)
- Run to Success (F5)
- Step Into (F11)
- Step Over (F10)
- Return to Caller (Shift-F11)

Program View

The Program view allows the current program to be inspected and modified. Selecting a program statement displays the corresponding WAM instruction stream in the Instructions list box.

To modify order of program statements, select the Program | Move Up and Move Down menu options.

By default, last-call optimization (LCO) is disabled. To enable LCO, select the Program | Enable Optimization menu option. Optimization settings are preserved when a program is saved.

Trace View

The Trace view is used to view diagnostic information. Tracing is initially disabled. It can be turned on using the Trace | Enable menu option.

The Trace view currently captures tracing information generated by Lingua.NET during grammar generation and statement parsing.

Architecture and Design

Prolog.NET consists of a Prolog compiler and interpreter based on the WAM architecture and modified to make use of the Common Language Infrastructure (CLI) (ECMA International 2006). It is written in C# (ECMA International 2006). This section contains additional information about the architecture and design of Prolog.NET.

Domains

The code in Prolog.NET can be divided into a set of distinct functional domains.

Grammar

The Grammar domain contains the terminals and nonterminals that define the Prolog.NET grammar. The classes in this domain rely on the Lingua.NET compiler library. They produce as output a representation of the parsed input using classes in the CodeDOM domain. See "Grammar" on page 22 for a specification of the grammar used by Prolog.NET.

All classes in this domain are in the Prolog.Grammar namespace.

CodeDOM

The CodeDOM domain provides a run-time representation of Prolog.NET language elements. All CodeDOM classes are immutable and, with some exceptions, are serializable. CodeDOM structures are created by the grammar and passed as input to the Prolog.NET compiler. They also provide the means to pass data to and from external library functions.

CodeDOM classes implement value equality. There is no semantic distinction between CodeDOM structures that differ based solely on reference equality tests.

All classes in this domain are in the Prolog.Code namespace.

See the Code diagram for more details.

Compiler

The Compiler domain contains the WAM compiler used by Prolog.NET. The compiler accepts as input a CodeDOM structure representing a rule, fact or query and produces a WAM instruction stream.

A WAM instruction stream is represented as an array of WamInstruction structures. It can also contain attributes – instances of classes that inherit from WamInstructionStreamAttribute – associated with specific instructions within the stream. Attributes are used to associate the names of permanent variables with instruction registers, and specify the entry points of individual procedure clauses within an instruction stream.

Persistence or serialization of WAM instruction streams is not supported at this time.

See the Compiler diagram for more details.

Program

The Program domain supports the management of Prolog.NET applications. The Program class and related child classes allow programs to be created, modified and persisted. Client applications can create or modify Programs using CodeDOM structures. The Program domain also includes a Query class. As with programs, queries are created using the appropriate CodeDOM structure.

Because CodeDOM structures are immutable, queries and individual procedure clauses can maintain references to user-supplied CodeDOM structures without compromising their integrity.

Programs isolate client applications from the WAM compiler and the WAM instruction streams associated with individual procedures. Programs expose *program* instruction streams –façades over underlying WAM instruction streams – that provide client applications protected access to the underlying WAM instruction stream. Program instruction streams are represented as lists of program instruction objects and, unlike WAM instruction streams, are better suited to WPF binding. Further, program instruction objects expose properties such as IsCurrentLocation that reflect the current run-time state of the underlying WAM machine.

Programs can reference libraries - collections of externally callable functions. By default, all programs reference the library exposed by the Library.Standard property.

See the Program diagram for more details.

WAM Machine

The WAM Machine domain contains the WAM machine and associated run-time support structures. To evaluate a query, a WAM machine is constructed for the query and associated program. When the specified query has been evaluated, the WAM machine becomes obsolete.

All WAM machine classes are declared as internal and are not directly manipulated by client applications.

See the Runtime diagram for more details.

Prolog Machine

The Prolog Machine domain provides client access to the underlying WAM machine and associated run-time structures. As with program instruction streams, the PrologMachine class and related child classes provide a protected façade over the underlying WAM machine. Among other things, they implement the concept of a run-time call stack, something that does not have a direct analog in the WAM machine runtime.

See the Prolog diagram for more details.

The Standard Library

Prolog.NET supports a call-level interface that provides access to externally defined methods. Methods are grouped into libraries which in turn are referenced by programs.

Methods are used to define *predicates* and *functions*. Predicates can be used within the body of a query or rule instead of a program procedure. For example:

```
is_liquid_temperature(F) :- greater_equal(F, 32), less_equal(F, 212).
```

Functions are used within *expressions*. For example:

```
celsius_to_farenheit(C,F) :- F is add(multiply(C, 1.8), 32).
```

Certain methods have associated operators. The above rules can be rewritten using their corresponding operators as follows:

```
is_liquid_temperature(F) :- F >= 32, F =< 212.
celsius_to_farenheit(C,F) :- F := C * 1.8 + 32.
```

With some exceptions, a method can be used as both a predicate and a function. This is described in more detail below.

Data Types

Prolog.NET has the built-in support for the following data types:

Data Type	CLI Data Type	CodeDOM Class	WAM Value Class
boolean	System.Boolean	CodeValueBoolean	WamValueBoolean
date	System.DateTime	CodeValueDateTime	WamValueDateTime
double	System.Double	CodeValueDouble	WamValueDouble
exception	System.Exception	CodeValueException	WamValueException
integer	System.Int32	CodeValueInteger	WamValueInteger
object	System.Object	CodeValueObject	WamValueObject
string	System.String	CodeValueString	WamValueString
type	System.Type	CodeValueType	WamValueType

The CodeValue.Create factory method creates an instance of the appropriate CodeValue subclass for any object. If an instance of an unsupported data type is specified, a CodeValueObject is returned. A CodeValueObject is also returned if null is specified.

No implicit type conversion is performed by CodeValue.Create. For example, if a short is specified, a CodeValueObject is returned.

The WamValue.Create factory method creates an instance of the appropriate WamValue subclass for any CodeValue object. A WamValueObject is returned if null is specified.

Operators

Prolog.NET has support for the following operators:

Priority	Operators			
200	**	^		
400	*	/	rem	mod
	<<	>>		
500	+	-	/\	\/
700	=	?=	\=	=..
	==	\==		
	@<	@=<	@>	@>=
	is	=:=	=\=	
	<	=<	>	>=

The priority values shown are the default operator priorities as defined by ISO Prolog. Support for operators is currently predefined by the Prolog.NET grammar. Operators cannot be defined nor their priorities modified by Prolog.NET programs.

Predicates

Predicates are library methods defined by the Predicate class. The WAM machine calls non-backtracking predicates using a PredicateDelegate:

```
internal delegate bool PredicateDelegate(
    WamMachine machine,
    WamReferenceTarget[] arguments)
```

Methods which support backtracking are accessed using a BacktrackingPredicateDelegate:

```
internal delegate IEnumerable<bool> BacktrackingPredicateDelegate(
    WamMachine machine,
    WamReferenceTarget[] arguments);
```

When a backtracking predicate is used, a WamChoicePoint is automatically created and is used to save the enumerator returned by the delegate. When backtracking occurs, the enumerator's MoveNext method is used to retrieve the next solution. The predicate fails when MoveNext returns false. Unlike non-backtracking predicate delegates, it is not necessary for the enumerator to explicitly return false.

Predicates have direct access to the WAM machine and may not be defined by external assemblies. Predicates normally appear in place of normal procedure calls within the body of a rule:

```
a(X,Y) :- unify(X,Y)
```

Returning false from a predicate called in this manner will cause normal backtracking to occur.

With some restrictions, predicates can also appear within expressions. For example:

```
a(X,Y) :- U is can_unify(X,Y), print(U)
```

prints true or false based on the success or failure of the can_unify. Predicates are not permitted within expressions if:

- They support backtracking
- They cause side effects (e.g. they unify one or more arguments)

When a PredicateDelegate is registered, the caller indicates if the predicate causes side-effects. If a predicate is incorrectly registered, the behavior is undefined.

Functions

Functions are library methods defined by the Function class. The WAM machine calls functions using a FunctionDelegate:

```
public delegate CodeTerm FunctionDelegate(CodeTerm[] arguments)
```

Functions do not have direct access to the WAM machine and may be defined by external assemblies. Functions normally appear in the body of an expression:

```
a(X,Y) :- Sum is add(X,Y), print(Sum)
```

When a function is called, all arguments are dereferenced and WamReferenceTarget objects are converted to their CodeDOM counterparts. Any unbound variables are converted into CodeValueObject's containing null.

When a function returns, the CodeTerm object is converted back to the appropriate WamReferenceTarget object. Functions never cause programs to terminate. Functions should indicate failure by returning a CodeValueException object which is converted to a WamValueException. If an unhandled exception is raised by a function, a WamValueException is created automatically by the WAM runtime.

If a function appears in place of procedure call, the WAM runtime attempts to convert the CodeTerm to a boolean and succeeds if the result is true. Otherwise, backtracking occurs.

Summary

The following table summarizes the methods defined by Library.Standard. All methods can be used as predicates. Methods can be used as functions unless otherwise noted.

The use of operators is optional. For example, "less_equal(X,Y)" is functionally equivalent to "X =< Y".

In the table below, "ISO" indicates support by both ISO Prolog and GNU Prolog; "GNU" indicates support by GNU Prolog.

Note that the standard integer type in Prolog.NET is 32-bit and the standard floating point type is 64-bit.

Unless otherwise noted, all variables are dereferenced prior to use. The term "uninstantiated variable" is properly defined to be either an unbound variable or a variable bound to an unbound variable.

Name		Operator		Function
Term Unification and Evaluation				
unify/2		=	ISO	No
can_unify/2		?=		
cannot_unify/2		\=	ISO	
is/2		is :=	ISO	No
assert/1				No
Control Constructs				
true/0	ISO			
fail/0	ISO			
for/3	GNU			No
All Solutions				
findall/3	ISO			No
Type and Value Testing				
var/1	ISO			
nonvar/1	ISO			
atom/1	ISO			
integer/1	ISO			
float/1	ISO			
number/1	ISO			
atomic/1	ISO			
compound/1	ISO			
callable/1	ISO			
list/1	GNU			
partial_list/1	GNU			
list_or_partial_list/1	GNU			
is_type/2				
is_null/1				
is_empty/1				
Term Processing				
functor/3				No
arg/3				No
composed_of/2		=..	ISO	No
copy_term/2				No

Name		Operator		Function
Type Conversion Expressions				
get_type/1				
type_of/1				
to_integer/1				
to_double/1				
to_string/1				
to_string/2				
to_date/1				
to_date/3				
to_boolean/1				
ceiling/1	ISO			
floor/1	ISO			
round/1	ISO			
truncate/1	ISO			
Arithmetic Expressions				
negate/1		-	ISO	
inc/1	GNU			
dec/1	GNU			
add/2		+	ISO	
subtract/2		-	ISO	
multiply/2		*	ISO	
divide/2		/	ISO	
integer_divide/2		//	ISO	
rem/2	ISO			
mod/2	ISO	%		
bitwise_and/2		/\	ISO	
bitwise_or/2		\/	ISO	
bitwise_xor/2		^	GNU	
bitwise_not/1		\	ISO	
shift_left/2		<<	ISO	
bitwise_shift_right/2				
integer_shift_right/2		>>	ISO	
abs/1	ISO			
sign/1	ISO			
min/2	GNU			
max/2	GNU			
power/2		**	ISO	
sqrt/1	ISO			
atan/1	ISO			
cos/1	ISO			
acos/1	GNU			
sin/1	ISO			
asin/1	GNU			
exp/1	ISO			
log/1	ISO			

Name		Operator		Function
String Expressions				
substring/2				
substring/3				
length/1				
contains/2				
replace/3				
Term Comparison				
term_equal/2		==	ISO	
term_unequal/2		\==	ISO	
term_less/2		@<	ISO	
term_less_equal/2		@=<	ISO	
term_greater/2		@>	ISO	
term_greater_equal/2		@>=	ISO	
Value Comparison				
equal/2		=:=	ISO	
unequal/2		=\=	ISO	
less/2		<	ISO	
less_equal/2		=<	ISO	
greater/2		>	ISO	
greater_equal/2		>=	ISO	
Random Numbers				
randomize/0	GNU			No
set_seed/1	GNU			No
get_seed/1	GNU			No
random/1	GNU			No
random/3	GNU			No

Reference

The following section describes methods defined by Library.Standard.

Unless otherwise noted, arguments may be either instantiated or uninstantiated. Arguments which must be instantiated are preceded by a +. Arguments which must be uninstantiated are preceded by a -.

Methods which are currently defined but not implemented are preceded by a †.

Term Unification and Evaluation

unify(Term1, Term2) (=)

unifies the specified arguments.

can_unify(Term1, Term2) (?=)

succeeds if the specified arguments can be unified.

cannot_unify(Term1, Term2) (\=)

succeeds if the specified arguments cannot be unified.

is(Term, +Expression) (:=)
: unifies Term with the value of Expression.

assert(+Expression)
: succeeds if the value of Expression is true; otherwise, fails.

Control Constructs

true
: always succeeds.

fail
: always fails.

for(Counter, +Lower, +Upper)
: successively unifies Counter with the sequence of integers bounded by Lower and Upper.

All Solutions

findall(Variable, +Goal, Result)
: evaluates all solutions to Goal and unifies Result with the list of all values of Variable as defined within Goal.

  Note: the unification state of Variable does not affect the behavior of findall.

Type and Value Testing

var(Term)
: succeeds if Term is an unbound variable.

nonvar(Term)
: succeeds if Term is a value.

atom(Term)
: succeeds if Term is an atom (i.e. a term of arity 0.)

integer(Term)
: succeeds if Term is an integer.

float(Term)
: succeeds if Term is a floating point number.

number(Term)
: succeeds if Term is either an integer or floating point number.

atomic(Term)
: succeeds if Term is an atom, integer or floating point number.

compound(Term)
: succeeds if Term is a compound term (i.e. a term of arity > 0.)

† list(Term)

succeeds if Term is a complete list (i.e. a list structure with no unbound tails.)

† partial_list(Term)

succeeds if Term is a partial list (i.e. a list structure containing an unbound tail.)

† list_or_partial_list(Term)

succeeds if Term is either a list or partial list.

is_type(Term, +Type)

succeeds if Term is type-compatible with the CodeValueType specified by Type. Uninstantiated terms are type-compatible with System.Object.

is_null(Term)

succeeds if Term is uninstantiated or is a CodeValueObject containing null.

is_empty(Term)

succeeds if Term contains the empty string.

Term Processing

† functor(+Term, Name, Arity)

unifies Name and Arity with an atom and integer representing the name and arity of Term.

† functor(-Term, +Name, +Arity)

unifies Term with a term whose functor name and arity is defined by the atom Name atom and integer Arity.

† arg(+N, +CompoundTerm, Term)

unifies Term with the integer N^{th} argument of CompoundTerm.

† composed_of(+Term, List) (=..)

unifies List with a list whose head contains an atom representing the functor name of Term and whose tail contains the arguments of Term.

† composed_of(-Term, +List) (=..)

unifies Term with a term whose functor name is defined by the atom head of List and whose arguments are defined by the tail of List.

† copy_term(Term1, Term2)

unifies Term2 with a copy of Term1. If Term1 is uninstantiated, succeeds if Term2 is also uninstantiated; otherwise, fails.

Type Conversion Expressions

get_type(+TypeName)

returns a CodeValueType containing the type identified by the TypeName string.

type_of(Term)
: returns a CodeValueType containing the type of Term. If Term is uninstantiated, type System.Object is returned.

to_integer(Term)
: converts Term to a CodeValueInteger.

to_double(Term)
: converts Term to a CodeValueDouble.

to_string(Term)
: converts Term to a CodeValueString. If Term is uninstantiated, or a CodeValueObject containing null is specified, the empty string is returned.

to_string(Term, +Format)
: converts Term to a CodeValueString using the specified Format string.

to_date(Term)
: converts Term to a CodeValueDateTime.

to_date(+Year, +Month, +Day)
: returns a CodeValueDateTime using the specified Year, Month and Day.

to_boolean(Term)
: converts Term to a CodeValueBoolean.

ceiling(Term)
: returns the smallest integer greater or equal to Term.

floor(Term)
: returns the largest integer less than or equal to Term.

round(Term)
: returns the closest integer to Term.

truncate(Term)
: returns the integer part of Term.

Arithmetic Expressions

All math operations support integer and double data types. No implicit type conversion is performed.

negate(Value)
: returns the negation of Value.

inc(Value)
: returns Value + 1.

dec(Value)
: returns Value - 1.

add(Value1, Value2) (+)
returns Value1 + Value2.

subtract(Value1, Value2) (-)
returns Value1 – Value2.

multiply(Value1, Value2) (*)
returns Value1 * Value2.

divide(Value1, Value2) (/)
returns Value1 / Value2.

integer_divide(Value1, Value2)
returns round(Value1 / Value2).

rem(Value1, Value2)
returns the remainder produced by integer_divide/2.

mod(Value1, Value2) (%)
returns Value1 modulo Value2.

bitwise_and(Value1, Value2) (/\)
returns the bitwise-and of Value1 and Value2.

bitwise_or(Value1, Value2) (\/)
returns the bitwise-or of Value1 and Value2.

bitwise_xor(Value1, Value2) (^)
returns the bitwise-xor of Value1 and Value2.

bitwise_not(Value) (\)
returns the bitwise negation of Value.

shift_left(Value,N) (<<)
returns the bits of integer Value shifted left N positions.

integer_shift_right(Value,N) (>>)
returns the bits of integer Value shifted right N positions, preserving the sign bit of Value.

bitwise_shift_right(Value,N)
returns the bits of integer Value shifted right N positions, setting the sign bit of Value to 0.

abs(Value)
returns the absolute value of Value.

sign(Value)
returns 1 if Value is positive, 0 if Value is 0 and -1 if Value is Negative.

min(Value1, Value2)
: returns the minimum of Value1 and Value2.

max(Value1, Value2)
: returns the maximum of Value1 and Value2.

power(Value1, Value2) (**)
: returns Value1 raised to the power of Value2.

sqrt(Value)
: returns the square root of Value.

atan(Value)
: returns the arctangent of Value.

cos(Value)
: returns the cosine of Value.

acos(Value)
: returns the arccosine of Value.

sin(Value)
: returns the sine of Value.

asin(Value)
: returns the arcsine of Value.

exp(Value)
: returns *e* raised to the power of Value.

log(Value)
: returns the natural logarithm of Value.

String Expressions

Except as noted, all arguments are converted to CodeValueString values before processing. If a CodeValueObject containing null is specified, the empty string is used; strings are never permitted to be null in Prolog.NET.

substring(String, Index)
: Returns the substring of String starting at specified character Index.

substring(String, Index, Length)
: Returns the substring of String with the specified Length starting at the specified character Index.

length(String)
: Returns a CodeValueInteger containing the length of String.

contains(String, Substring)
: Returns a CodeValueBoolean indicating if String contains the specified Substring.

replace(String, From, To)
Returns String with all occurrences of From replaced by To.

Term Comparison

Term comparison is based on the following term priority:

- uninstantiated variables
- floating point numbers
- integers
- other value types
- atoms
- compound terms
  - functor arity
  - functor name
  - arguments, left to right

The ordering of uninstantiated variables is unspecified in Prolog.NET.

† term_equal(Term1, Term2) (==)
succeeds if Term1 is equal to Term2.

† term_unequal(Term1, Term2) (\==)
succeeds if Term1 is not equal to Term2.

† term_less(Term1, Term2) (@<)
succeeds if Term1 is less than Term2.

† term_less_equal(Term1, Term2) (@=<)
succeeds if Term1 is less than or equal to Term2.

† term_greater(Term1, Term2) (@>)
succeeds if Term1 is greater than Term2.

† term_greater_equal(Term1, Term2) (@>=)
succeeds if Term1 is greater than or equal to Term2.

Value Comparison

Arguments passed to comparison functions must implement the IComparable interface, otherwise a CodeValueException is returned.

equal(Term1, Term2) (=:=)
succeeds if Term1 is equal to Term2.

unequal(Term1, Term2) (=\=)
succeeds if Term1 is not equal to Term2.

less(Term1, Term2) (<)
succeeds if Term1 is strictly less than Term2.

less_equal(Term1, Term2) (=<)
succeeds if Term1 is less than or equal to Term2.

greater(Term1, Term2) (>)
succeeds if Term1 is strictly greater than Term2.

greater_equal(Term1, Term2) (>=)
succeeds if Term1 is greater than or equal to Term2.

Random Numbers

randomize
sets the random number generator seed to a time-dependent default value.

set_seed(+Seed)
sets the random number generator to the specified seed value.

get_seed(Seed)
unifies Seed with the current random number generator seed value.

random(?Value)
unifies Value with a floating point random number R where $0 \leq R < 0$.

random(?Value,+ Lower, +Upper)
unifies Value with an integer random number R where Lower $\leq R <$ Upper.

Grammar

This section contains the grammar supported by Prolog.NET.

Terminals

Atom	[a-z][a-zA-Z0-9_]{0,99}|.	(?=()
Bar	|	
CloseBrace	}	
CloseBracket	]	
CloseParenthesis	)	
ColonDash	:-	
Comma	,	
Comment	/*[.\n]*?*/	
Cut	!	
LineComment	//(?!/).*	(?=\n)
LiteralBoolean	(true|false)	(?![a-zA-Z0-9_])
LiteralDouble	-?[0-9]{1,10}\.[0-9]{1,10}	
LiteralInteger	-?[0-9]{1,10}	
LiteralString	"([^"]|"")*"	
OpAdd	+	
OpBitwiseAnd	/\	
OpBitwiseExclusiveOr	^	
OpBitwiseNegate	\	
OpBitwiseOr	\/	
OpCannotUnify	=	
OpCanUnify	?=	
OpComposedOf	=..	
OpDivide	/	(?!/)
OpenBrace	{	
OpenBracket	[	
OpenParenthesis	(	
OpEqual	=:=	
OpGreater	>	
OpGreaterEqual	>=	
OpIs1	:=	
OpIs2	is	(?![a-zA-Z0-9_])
OpLess	<	
OpLessEqual	=<	
OpModulo	mod	(?![a-zA-Z0-9_])
OpMultiply	*	
OpPower	**	
OpRemainder	rem	(?![a-zA-Z0-9_])
OpShiftLeft	<<	
OpShiftRight	>>	
OpSubtract	-	
OpTermEqual	==	
OpTermGreater	@>	
OpTermGreaterEqual	@>=	
OpTermLess	@<	

OpTermLessEqual	@=<	
OpTermUnequal	\==	
OpUnequal	=\=	
OpUnify	=	
Period	.	(?!()
ProcedureComment	///.*	(?=\n)
Semicolon	;	
TerminalStop		
Variable	[A-Z][a-zA-Z0-9_]{0,99}	
Whitespace	[\t\r\n]+	

Statements

Program
:= OptionalProgramStatement AdditionalProgramStatements

OptionalProgramStatement
:= Statement
:=

AdditionalProgramStatements
:= Period OptionalProgramStatement AdditionalProgramStatements
:=

Statement
:= Clause
:= Query

Clause
:= OptionalProcedureComments Term OptionalRuleBody

Query
:= ColonDash StatementElement AdditionalStatementElements

OptionalProcedureComments
:= ProcedureComment OptionalProcedureComments
:=

OptionalRuleBody
:= ColonDash StatementElement AdditionalStatementElements
:=

AdditionalStatementElements
:= Comma StatementElement AdditionalStatementElements
:=

Statement Elements

StatementElement
:= BinaryElementExpression700
:= Cut

BinaryElementExpression700
:= BinaryElementExpression700 BinaryOp700 BinaryElementExpression500
:= BinaryElementExpression500

BinaryElementExpression500
:= BinaryElementExpression500 BinaryOp500 BinaryElementExpression400
:= BinaryElementExpression400

BinaryElementExpression400
:= BinaryElementExpression400 BinaryOp400 BinaryElementExpression200
:= BinaryElementExpression200

BinaryElementExpression200
:= BinaryElementExpression200 BinaryOp200 UnaryElementExpression200
:= UnaryElementExpression200

UnaryElementExpression200
:= Element
:= UnaryOp200 Element

Element
:= OpenBrace BinaryElementExpression700 CloseBrace
:= OpenParenthesis BinaryElementExpression700 CloseParenthesis
:= Term
:= Value
:= Variable

Terms

Term
:= Atom OptionalTermBody

OptionalTermBody
:= OpenParenthesis OptionalCompoundTermBody CloseParenthesis
:=

OptionalCompoundTermBody
:= CompoundTermBody
:=

CompoundTermBody
:= CompoundTermMember AdditionalCompoundTermMembers

CompoundTermMember
:= BinaryElementExpression700

AdditionalCompoundTermMembers
:= Comma CompoundTermMember AdditionalCompoundTermMembers
:=

Values

Value
:= List
:= LiteralBoolean
:= LiteralDouble
:= LiteralInteger
:= LiteralString

List
:= OpenBracket OptionalListBody CloseBracket

OptionalListBody
:= ListBody
:=

ListBody
:= ListItems OptionalListTail

ListItems
:= ListItem AdditionalListItems

ListItem
:= CompoundTermMember

AdditionalListItems
:= Comma ListItem AdditionalListItems
:=

OptionalListTail
:= ListTail
:=

ListTail
:= Bar ListTailItem

ListTailItem
:= List
:= Variable

Operators

BinaryOp200
:= OpBitwiseExclusiveOr
:= OpPower

BinaryOp400
:= OpDivide
:= OpModulo
:= OpMultiply
:= OpRemainder

```
                := OpShiftLeft
                := OpShiftRight

BinaryOp500
                := OpAdd
                := OpBitwiseAnd
                := OpBitwiseOr
                := OpSubtract

BinaryOp700
                := OpCannotUnify
                := OpCanUnify
                := OpComposedOf
                := OpEqual
                := OpGreater
                := OpGreaterEqual
                := OpIs1
                := OpIs2
                := OpLess
                := OpLessEqual
                := OpTermEqual
                := OpTermGreater
                := OpTermGreaterEqual
                := OpTermLess
                := OpTermLessEqual
                := OpTermUnequal
                := OpUnequal
                := OpUnify

UnaryOp200
                := OpBitwiseNegate
                := OpSubtract
```

Bibliography

Aït-Kaci, Hassan. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

ECMA International. "Standard ECMA-334, C# Language Specification." 2006.

ECMA International. "Standard ECMA-335, Common Language Infrastructure (CLI)." 2006.

Warren, David H. D. *An abstract Prolog instruction set.* Menlo Park, CA: SRI International, 1983.

[1] Certain instructions (e.g. put_unsafe_value) are not required by Prolog.NET runtime and are not generated by the Prolog.NET compiler. Therefore, instruction streams produced by Prolog.NET are not compatible with standard WAM implementations. On the other hand, the Prolog.NET WAM machine can (or rather, in this release, *has the potential to*) properly execute a "standard" instruction stream by disregarding aspects of instructions such as a put_unsafe_value that are made unnecessary by the Prolog.NET run-time environment.

[2] For optimization purposes, the WAM can also manipulate atoms (0-ary terms) as constants. Prolog.NET does not support this optimization.