P# Manual (version 1.1.3)

Jon Cook — October 1, 2003

1 Introduction

P# is a Prolog implementation designed for use with Microsoft's .NET Framework. Specifically it translates Prolog to C# source code, allowing interoperation with C# and hence with other .NET languages. This allows the development of software which combines Prolog back-ends with C# front-ends. Much of P# is shared with SICStus Prolog and the tool from which P# is derived, Prolog Cafe (version 0.4.4). Details of predicates and operators which are shared with SICStus Prolog are not given, and in these cases the user is referred to the SICStus Prolog documentation.

P# can be downloaded from the web-page: `http://www.dcs.ed.ac.uk/home/jjc`.

P# 1.1.3 improves on P# 1.1.2 by including support for floating point numbers (which brings it in line with Prolog Cafe 0.6.1), a `stop/1` predicate which aborts a P# Prolog thread and the `:/2` predicate for calling C# methods and converting their return data to Prolog types without needing to use `cs_term`. Also the constructors of `PrologInterface` have been made more sensible, and in line with C# naming conventions P# runtime methods visible from the C# programmer's code begin with capital letters. The old method names with lower case letters still work, but generate compiler warnings.

As of Mono 0.26 and P# 1.1.3, P# runs and compiles under Mono. Mono currently allows you to compile and run C# programs under Linux and some other versions of UNIX. Thus, P# can be now be used under some versions of UNIX.

2 Compiling a Prolog program to an executable

In order to run the interpreter, `PsharpIntrp.exe`, you can either copy the DLL, `Psharp.dll` into the same directory as the interpreter executable or add the DLL to the Global Assembly Cache (GAC).

To translate a Prolog source file, called `myfile.pl`, say, run P# and enter the command

```
compile('myfile').
```

then press CTRL-Z to exit P#.

If the compilation was successful there will now be a number of C# files in the directory. Copy the file `Loader.cs` into the directory as well. The file `Loader.cs` can be downloaded from the P# homepage.

Assuming that your csc (C# compile) command is reachable from the path, and that you have put your `Psharp.dll` file in the folder `C:\psharp`, say, you should then be able to compile these into a P# application with the following command:

```
csc /r:"C:\psharp\Psharp.dll" /out:MyProgram.exe *.cs
```

This generates an executable file which when run will load the P# DLL and start executing the predicate `main/0` in the Prolog file you compiled. To run another predicate of arity zero, say `my_pred` you can use the command:

```
MyProgram.exe MyPred
```

Notice how Prolog predicate names are renamed (if in doubt just look at the class name in the generated C# file).

To run P# Prolog code from the P# interpreter use the built-in predicates `consult/1`, `compile/1` and `plcomp/1` (see below).

Alternatively, use the P# Graphical interface.

3 Predicates shared with SICStus Prolog

3.1 Interpreter

consult/1, listing/0, debug/0, nodebug/0, trace/0, notrace/0, spy/1, nospy/1, nospyall/1, leash/1

Note that the support for these predicates is not as extensive as in SICStus Prolog. A list of files to consult can be typed directly in the terminal e.g.

```
?- [file1, 'File2'].
```

however `[user].` cannot be used to type in code on the terminal.

3.2 Input of Terms

read/1, read/2

3.3 Output of Terms

display/1, write_canonical/1, write_canonical/2, write/1, write/2, writeq/1, writeq/2

3.4 Character I/O

nl/0, nl/1, get0/1, get0/2, get/1, get/2, put/1, put/2, skip/1, skip/2, peek_char/1, peek_char/2, tab/1, tab/2, ttynl/1, ttyflush/1, ttyget0/1, ttyget/1, ttyput/1, ttyskip/1, ttytab/1

3.5 Stream I/O

open/3, close/1, current_input/1, current_output/1, set_input/1, set_output/1, flush_output/0, flush_output/1

3.6 Arithmetic

(is)/2, (=:=)/2, (=\=)/2, (<)/2, (>)/2, (=<)/2, (>=)/2

3.7 Comparison of Terms

(==)/2, (\==)/2, (@<)/2, (@>)/2, (@=<)/2, (@>=)/2, (?=)/2, compare/3, sort/2

3.8 Control

otherwise/0, true/0, fail/0, false/0, repeat/0, (,)/2, (;)/2, (->)/2, (\+)/1, (!)/0, call/1, once/1

3.9 Meta-Logic

var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, ground/1, compound/1, callable/1, functor/3, arg/3, (=..)/2, atom_chars/2, number_chars/2, name/2

3.10 Modification of the Program

asserta/1, assertz/1, assert/1, clause/2, retract/1, retractall/1, abolish/1

3.11 All Solutions

findall/3

3.12 Definite Clause Grammars

'C'/3, expand_term/2, (`-->`)/2

3.13 Miscellaneous predicates shared with SICStus Prolog

copy_term/2, (`=`)/2, length/2, numbervars/2, halt/0, abort/0, statistics/0, statistics/2, op/3, current_op/3

4 Predicates shared with Prolog Café

4.1 Expressions

The types of expression which may occur on the right of a call to `is/2` are listed in Table I.

4.2 Operators

See Table II for a summary of P#'s Prolog operators. The extra operators added for Linear Logic Programming (LLP) support are included in Table III.

4.3 Input of Terms

read_token/1, read_tokens/2, read_with_variables/2, readint/1, readline/1, parse_tokens/2

4.4 Reading in Programs

compile(+FileName)
compile(+ListOfFilenames)

Compile a Prolog file or list of Prolog files to C#. The Prolog files are found in the currently selected directory and the C# files are output in the same directory. The Filename(s) should not include the .pl suffix. For example, to compile test.pl, use `compile( test )`. To compile `Test.pl`, use `compile( 'Test' )`. To compile both use `compile( [ test, 'Test' ] )`.

fcompile(+FileName)
fcompile(+ListOfFilenames)

Same as compile/1.

Table I: Expressions

`5`	integer
`3.4`	floating point number
`+5`	unary plus
`-3`	unary minus
`3 + X`	addition
`X - 1`	subtraction
`X * Y`	multiplication
`X / Y`	division
`X // Y`	integer quotient (round down)
`8 mod 3`	modulus
`max(4,X)`	maximum
`min(X,7.0)`	minimum
`2 << 4`	left shift
`128 >> 2`	right shift
`7 /\ 4`	bitwise and
`1 \/ 2`	bitwise or
`~23`	bitwise negation
`10 # 3`	bitwise exclusive or
`abs(-5)`	absolute value ($\|-5\|$)
`integer(4.5)`	convert to integer by rounding down
`float(3)`	convert to float
`random`	a random float value between 0.0 and 1.0
`pi`	the constant π
`e`	the constant e
`sin(X)`	sine
`cos(X)`	cosine
`tan(X)`	tangent
`asin(X)`	arcsine
`acos(X)`	arccosine
`atan(X)`	arctangent
`sqrt(9.7)`	square root
`log(Y)`	natural logarithm
`exp(X)`	exponentiation (e^X)
`pow(2,3)`	power (2^3)
`degrees(X)`	convert from radians to degrees
`radians(X)`	convert from degrees to radians

Table II: Prolog operators

1170	xfx	`:-`	
1170	xfx	`-->`	
1170	fx	`:-`	
1170	fx	`?-`	
1150	fx	`mode`	[ignored]
1150	fx	`public`	[ignored]
1150	fx	`dynamic`	
1150	fx	`multifile`	[ignored]
1150	fx	`block`	[ignored]
1150	fx	`meta_predicate`	[ignored]
1100	xfy	`;`	
1050	xfy	`->`	
1000	xfy	`,`	
900	fy	`\+`	
900	fy	`spy`	
900	fy	`nospy`	
700	xfx	`=`	
700	xfx	`is`	
700	xfx	`=..`	
700	xfx	`==`	
700	xfx	`\==`	
700	xfx	`@<`	
700	xfx	`@>`	
700	xfx	`@=<`	
700	xfx	`@>=`	
700	xfx	`=:=`	
700	xfx	`\=`	
700	xfx	`<`	
700	xfx	`>`	
700	xfx	`=<`	
700	xfx	`>=`	
550	xfy	`:`	
500	yfx	`+`	
500	yfx	`-`	
500	yfx	`#`	
500	yfx	`/\`	
500	yfx	`\/`	
500	fx	`+`	
500	fx	`-`	
400	yfx	`*`	
400	yfx	`/`	
400	yfx	`//`	
400	yfx	`<<`	
400	yfx	`>>`	
300	xfx	`mod`	
300	xfx	`~`	
200	xfy	`^`	

Table III: LLP Operators

1200	fx	`forall`
1190	xfx	`\`
1150	fx	`:- resource`
1060	xfy	`&`
950	xfy	`-<>`
950	xfy	`=>`
900	fy	`!`

4.5 User defined hash-tables

See the Prolog Café documentation for details of using hash-tables.

open_table(?Table)
close_table(+Table)
clear_table(+Table)
set_table(+Table)
current_table(?Table)
get_term(+Key, ?Term)
get_term(+Table, +Key, ?Term)
put_term(+Key, ?Term)
put_term(+Table, +Key, ?Term)

asserta(+Table, +Clause)
assertz(+Table, +Clause)
assert(+Table, +Clause)
clause(+Table, +Head, ?Body)
retract(+Table, +Clause)
retractall(+Table, +Clause)
abolish(+Table, +Functor/Arity)

4.6 Miscellaneous predicates shared with Prolog Café

version/0

Output the version string.

time(+Goal)

Calls the Goal and outputs a message indicating how long the goal took to execute.

interpreter/0

Run the Prolog interpreter.

url_source(+URL, -Text)

Look up the URL (an atom) and returns the html contents of that location as an atom.

5 Calling C# from Prolog

load_assembly(+AssemblyToLoad)

Load a .NET assembly so that the predicate classes defined within it can be called from Prolog. P# first tries passing the string provided (as an atom) to the C# method `System.Reflection.Assembly.Load()`, and if this fails to load an assembly it then tries `System.Reflection.Assembly.LoadWithPartialName()`.

For example:

```
load_assembly( 'System.Windows.Forms' ).
```

When the Prolog engine looks for a predicate, it first searches in the P# DLL, if it is not found there it searches the loaded assemblies starting from the last loaded assembly and ending at the first loaded assembly. If an assembly which has already been loaded is loaded again then that assembly is moved to the top of the list so that it will have priority over all of the others.

cs_constructor(+Class(+Args, ...), -CsObjectTerm)

Construct a C# object and return the corresponding C# object term. For example:

```
cs_constructor( 'System.Collections.ArrayList', AL ).
```

or

```
cs_constructor( 'System.Collections.ArrayList'( 10 ), AL10 ).
```

cs_method(+Class, +StaticMethod(+Args, ...), -ReturnValue)
cs_method(+Object, +Method(+Args, ...), -ReturnValue)

Call a C# method. When you call Prolog from C# and then call back to C# from the Prolog side you have to call PrologInterface.AddAssembly(...) before the call. This enables cs_method to find the method in the calling assembly. For example:

```
PrologInterface sharp = new PrologInterface( );
sharp.AddAssembly( System.Reflection.Assembly.GetExecutingAssembly( ) );
sharp.SetPredicate( ... );
sharp.Call( );
```

The Prolog looks like, for example:

```
cs_method( 'System.Console', 'WriteLine'( 'Hello World!' ), _ ).
cs_method( 'System.Math', 'Max'( 3, 4 ), M ).
```

cs_term(?CsObjectTerm, ?PrologTerm)

Convert between C# objects represented as CsObjectTerm's and Prolog values. It is often necessary to convert values returned by cs_method using this predicate for example:

`cs_method( 'System.Math', 'Max'( 3, 4 ), M )` instantiates M to System.Int32(4). Then call `cs_term( Int, M )`, which instantiates Int to 4.

+ClassOrInstance : +MethodCall [new in P# 1.1.3]

The colon operator combines cs_method/3 with cs_term/2. It is used as follows:

`'System.Math':'Max'( 3, 4, Int )` instantiates Int to 4.

or

`Object:'Method'( Arg1, Arg2, ReturnValue ).`

cs_get_field(+ClassOrInstance, +Field, -Value)

Get the value of a C# field. For example:

```
cs_get_field( 'System.Int32', 'MaxValue', MV ).
```

Instead of a class a variable instantiated to a C# object term can be used as the first argument.

cs_set_field(+ClassOrInstance, +Field, +Value)

Set the value of a C# field.

cs_object(?X)

Succeeds if X is a C# object term.

6 Miscellaneous predicates new in P#

plcomp(+File)

Compile the given Prolog file to C# and then compile the C# internally so that the predicates defined in the Prolog file can be called thereafter in the interpreter session.

plcomp(+File, +Output)

Same as plcomp/1 except that an executable file with the given Output name is generated as well.

plcs(+File)

Same as compile/1, except that only one file may be given.

7 Concurrency

Note that concurrency features will not work for interpreted code (code loaded by `consult/1` or by entering a list at the prompt). To run concurrent code from the interpreter internally compile the Prolog file using `plcomp/1`.

7.1 Forking threads and message passing

wait_for/1, fork/1, fork/2, stop/1

In order to be able to create new threads, we add a primitive called `fork/1`. The `fork` predicate takes a structure as an argument and then forks a new thread which evaluates the structure.

A `fork/2` primitive is also available, which forks the first argument and returns a Prolog representation of a C# object representing the new thread. This object can then be returned to the C# part of the program where it can be used to stop that thread. A predicate, named `stop/1` is provided, which can be used to stop a thread from the Prolog side. The predicate `stop/1` should not be used in P# 1.1.2 and earlier.

Having called `fork` with a structure containing an uninstantiated variable, anywhere in the syntax tree of the structure, a thread can use that variable to interact with the newly forked thread.

7.2 Communication between threads

The `wait_for` predicate takes as an argument a variable which is shared with an already forked thread. It then waits until one of the threads instantiates that variable and succeeds with the given instantiation. Except for this the instances of variables on different threads do not interact.

Consider the following program:

```
a( 2, 7 ).

and( Y ) :-
  fork( a( 1, Y ) ),
  fork( a( 2, Y ) ),
  fork( a( 3, Y ) ),
  wait_for( Y ).
```

Three threads are forked, each calling the predicate `a/2` with different values of the first argument. Only the second will instantiate `Y`, the second argument to 7. `wait_for(Y)` will wait until this happens and then `and/1` will succeed with `Y = 7`. It is also acceptable for a forked thread to wait for the thread which forked it or for forked threads to fork more threads.

The following example shows that forking integrates with backtracking.

```
alpha( 'a' ).
alpha( 'b' ).
alpha( 'c' ).
alpha( 'd' ).

correct( X, Y ) :-
  \+ var( X ), % prevent cheating
  X = 'c',
  Y = X.
```

```
guess( Z ) :-
  alpha( X ),
  fork( correct( X, Z ) ),
  fail.
guess( Z ) :-
  wait_for( Z ).
```

The `guess/1` predicate knows that the correct answer is **'a'**, **'b'**, **'c'** or **'d'**. However it can only find out which by calling `correct(X, Y)` with the correct letter as `X`, in which case `Y` is instantiated to that letter. The `guess/1` predicate forks a thread for each letter and waits for one of them to succeed. The variable `Z` correctly retains its concurrent information on backtracking, as it comes into existence as soon as `guess(Z)` is called.

The following example is similar to the last, except that tail-recursion is used instead of backtracking. The user enters a square integer between 0 and 20^2. The program forks 21 threads to try each of the possible square roots, and then waits for one of them to signal that the answer has been found.

```
sqroot( S, R ) :-
  sqroot_threads( S, R, 0 ),
  wait_for( R ).

sqroot_threads( S, R, 21 ) :-
  !.
sqroot_threads( S, R, N ) :-
  fork( ( S =:= N * N, R = N ) ),
  N1 is N + 1,
  sqroot_threads( S, R, N1 ).
```

Note that the double brackets in the `fork` are necessary as the fork takes only one argument, which in this case is a structure with functor `,/2`. This provides a way of writing the code to be forked "in-line".

7.3 Queueing of multiple solutions

It may be that the programmer wishes to use a concurrent variable more than once, indeed if he or she cannot, then some algorithms will require unnatural implementations.

If a bound concurrent variable is later unbound by backtracking, and then bound again to the same or a different value, then that new binding is enqueued on the queue of messages to be consumed.

Thus, a producer can give multiple bindings to a concurrent variable, possibly composing a set of solutions; and a consumer can repeatedly call `wait_for` to take each binding.

The `wait_for` predicate can be called repeatedly by using the usual `repeat/0` predicate, however `wait_for` also creates a choice-point and will yield the next solution on backtracking.

The following code creates two threads, a producer and a consumer. The producer generates integer values from 0 up to 10 and the consumer consumes each produced value, doubles it and outputs the corresponding result. The producer uses a predicate `pulse/2` which makes a binding and then undoes it straight away. The first clause of `pulse` makes the binding, and then fails. This failure causes backtracking to the last choice-point, which undoes the binding we made in the first clause and then executes the second clause which succeeds. Thus, the predicate call succeeds having made no lasting binding. This allows us to imperatively give successive bindings to the same variable.

```
main :-
  fork( prod( X ) ),
  cons( X ).

cons( X ) :-
  wait_for( X ),
  X2 is X * 2,
  write( X2 ),
  nl,
  fail.
cons( X ).
```

```
prod( X ) :-
  enum( X, 0 ).

enum( _, 11 ) :-
  !.
enum( X, N ) :-
  pulse( X, N ),
  N1 is N + 1,
  enum( X, N1 ).

pulse( X, N ) :-
  X = N,
  fail.
pulse( _, _ ).
```

When it is detected that all of those threads which have a copy of a concurrent variable are waiting for that variable, then all of those calls to `wait_for` fail. Thus, in the example above both of the threads eventually terminate, and in the square root example above, if the user asks for the root of a non-square integer the query will fail. If all of the forked threads with a variable succeed or fail having sent no message then a call to `wait_for` on the remaining thread will fail. However, care must be exercised. If we had defined `main` to fork both the producer and the consumer, then the main thread running under the interpreter would still have a copy of the variable `X` although it would never use it. This would stop the consumer thread from terminating. It is still possible to fork both threads by forking a thread which itself first forks the producer thread and then runs the consumer code. In this case the variable `X` is introduced on the consumer thread, not the interpreter thread.

7.4 The Global Table

global_table/1, global_assert/1, global_asserta/1, global_assertz/1, global_retract/1, global_retractall/1, global_abolish/1, global_call/1

Each forked thread is equipped with a private database which it can use in the normal way. In addition we provide a global database, which is shared between all the threads. Accesses are automatically protected by a mutex. The database can be modified by primitives `global_assert/1`, `global_retract/1` and so on. To query the database a `global_call/1` predicate is provided.

Since the set of assertions in the global table for a variable are specific to that variable, the assertions for different variables do not interfere with one another.

7.5 Monitors

lock/1, unlock/1

Our system includes two further primitives to ensure mutual exclusion among executing threads, namely, `lock/1` and `unlock/1`. Both of these take as an argument any Prolog term, and respectively acquire or release a monitor lock on the C# object representing that term.

backtrackable_lock/1

A `backtrackable_lock/1` predicate is also provided. The effect of calling `backtrackable_lock` is that everything deeper down the proof tree forms a critical region.

The P# runtime system keeps track of each lock and unlock operation and maintains a variable which stores the current depth of locking. When the P# Prolog thread terminates all of its locks are automatically released. This mirrors the semantics of the C# `lock` keyword, where a `finally` clause releases the monitor when the critical region is exited. Thus, if the thread is aborted, all of its locks are released.

7.6 Interoperation with C#

The locks dealt with by `lock` and `unlock` are C# locks (on C# objects). Similarly `fork` initializes and starts a new C# thread, unification calls the `PulseAll()` method on an object and `wait_for` calls the `Wait()` method on an object, although they do far more besides this.

A C# program which calls a P# Prolog predicate may wrap such a call with a fork.

Any variables passed to the predicate then become concurrent, allowing communication between the C# code and the P# Prolog before the Prolog terminates.

A P# Prolog predicate can call a C# method in the following way:

```
'System.Console':'WriteLine'( 'Hello World!', _ ).
```

The middle argument consists of the method name and any actual arguments. These C# arguments may include uninstantiated variables, in which case the C# will be passed a `VariableTerm` object. Thus, a concurrent variable can be passed from the P# Prolog side to the C# side. This time the use of `:/2` should be wrapped in a fork, for example:

```
run_cs_method( In, Out, ObjectToCall ) :-
    fork( ObjectToCall:'CsThreadStart'( In, Out ) ).
```

This would be matched on the C# side by code of the following form:

```
public object CsThreadStart( VariableTerm vt ) {
  ...

  // send message
  vt.Send( new IntegerTerm( 7 ) );

  // or await a message
  int msg = (int)( vt.Receive( ).toCsObject( ) );

  ...
  return ...
}
```

The `Send()` and `Receive()` C# methods use a temporary P# engine to respectively perform a unification and execute the `wait_for` predicate. Each undoes any existing binding of the concurrent variable that it is given first, and thus may be called repeatedly from the C# code. Such repetition must, however, be matched by backtracking on the P# side.

7.7 Miscellaneous Concurrency Predicates

sleep(+TimeInMillis)

Sleep for the specified time period.

zap_queue(+Variable)

Clear the queue of produced terms waiting to be consumed for a given concurrent variable.

wait_for_one/1

Similar to wait_for, but does not retrieve subsequent solutions on backtracking.

8 Calling Prolog from C#

8.1 Imports

This section details how Prolog can be called from C#. The C# code which calls Prolog must contain the following imports:

```
using JJC.Psharp.Lang;
using JJC.Psharp.Predicates;
```

and if linear logic is used, also the following:

```
using JJC.Psharp.Lang.Resource;
using JJC.Psharp.Resources;
```

You should compile your C# together with the C# files generated from Prolog by P#.

8.2 Creating a Prolog Interface

When you wish to call Prolog, you first need to create a Prolog Interface. PrologInterface has a number of constructors.

To create a new PrologInterface as a daemon thread which uses the C# System.Console class for input and output, which is the usual thing to do, use a command of the form

```
PrologInterface sharp = new PrologInterface( );
```

If you wish to set the thread to be a non-daemon thread, that is the C# application will be prevented from terminating if the PrologInterface thread had not terminated, use the constructor:

```
new PrologInterface( false )
```

If you wish to set different input, output and error streams use the following constructor, where new_in is a `JJC.Psharp.Lang.PushbackReader`, and new_out and new_err are both of type `System.IO.TextWriter`.

```
new PrologInterface( new_in, new_out, new_err )
```

A PushbackReader can be constructed in the following way, for example:

```
PushbackReader pr = new PushbackReader( System.In );
```

There is also a constructor which combines the last two PrologInterface constructors discussed:

```
new PrologInterface( new_in, new_out, new_err, false )
```

You must create a new PrologInterface object for each call into Prolog.

8.3 Adding the calling assembly

The next thing that you program should do, if you wish the Prolog to call back to C# is to add the calling assembly. Do this as follows:

```
sharp.AddAssembly( System.Reflection.Assembly.GetExecutingAssembly( ) );
```

Alternatively the following method can be called, but this will fail to work correctly under the current version of Mono (0.26).

```
sharp.AddCallingAssembly( );
```

8.4 Creating Terms

It is possible to create Prolog terms in C# which can then be passed to the Prolog side (see next subsection).

To create an integer term, use

```
IntegerTerm i = new IntegerTerm( 3 );
```

To create a floating point number term, use

```
DoubleTerm d = new DoubleTerm( 2.5 );
```

To create a symbol (atom), use

```
SymbolTerm s = SymbolTerm.MakeSymbol( "a_string" );
```

To create a C# object term, for example of a StringBuilder, use

```
CsObjectTerm o = new CsObjectTerm( new StringBuilder( ) );
```

To create a variable, use

```
VariableTerm v = new VariableTerm( );
```

If the P# Prolog code instantiates the variable, after the call to Prolog has been made, `v` can be dereferenced to obtain the instantiation, for example:

```
IntegerTerm it = (IntegerTerm)( v.Dereference() );
```

To create for example the list `[1,x,2]`, you could use the code:

```
ListTerm empty = SymbolTerm.MakeSymbol( "[]" );
ListTerm item1 = new IntegerTerm( 1 );
ListTerm item2 = SymbolTerm.MakeSymbol( "x" );
ListTerm item3 = new IntegerTerm( 2 );
ListTerm list = new ListTerm( item1,
                    new ListTerm( item2,
                        new ListTerm( item3, empty )
                    )
                );
```

To create for example the structure `age('Jon',25)`, you could use the code:

```
Term[] args = { SymbolTerm.MakeSymbol( "Jon" ),
                new IntegerTerm( 25 ) };
StructureTerm st = new StructureTerm(
     SymbolTerm.MakeSymbol( "age", 2 ),    // ( functor, arity )
     args );
```

8.5 Calling a Predicate

Suppose that you wish to call a predicate called lives_in/2 with semantics `lives_in( Person, City )` and you wish to obtain all solutions to the query

```
lives_in( Person, 'Edinburgh' ).
```

You would do this as follows:

```
VariableTerm person = new VariableTerm( );
PrologInterface sharp = new PrologInterface( );
// the next line is not necessary in this case.
sharp.AddAssembly( System.Reflection.Assembly.GetExecutingAssembly( ) );
sharp.SetPredicate( new LivesIn_2(
    person,
    SymbolTerm.MakeSymbol( "Edinburgh" ),
```

Table IV: Data Conversions

Prolog datatype	**C# datatype**
atom (SymbolTerm)	string (System.String)
integer (IntegerTerm)	int (System.Int32)
float (DoubleTerm)	double (System.Double)
(flat) list (ListTerm)	object[] (System.Object[])
C# object term (CsObjectTerm)	object (System.Object)
variable (VariableTerm)	JJC.Psharp.Lang.VariableTerm
structure (StructureTerm)	JJC.Psharp.Lang.StructureTerm

```
    new ReturnCs( sharp ) ) );

// call predicate to obtain all solutions
for( bool r = sharp.Call(); r; r = sharp.Redo() ) {
    Console.WriteLine( "Name: {0}", person.Dereference() );
}
```

If you only wanted one solution, instead of the `for` loop, you could just use `sharp.Call()`.

See the Prolog Café documentation for details of how to manipulate other Prolog terms from the C# side, such as `ListTerms`.

The data conversions between Prolog and C# are analogous to those for Prolog Café version 0.4.4, see Table IV.

9 Linear Logic Features

For information on the linear logic features see the link below.

Useful URLs

P# homepage (Jon Cook's homepage): `http://www.dcs.ed.ac.uk/home/jjc`

Downloading P# 1.1.3: `http://www.dcs.ed.ac.uk/home/jjc/psharp/psharp-1.1.3/dlpsharp.html`

Prolog Café: `http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/index-jp.html`

Prolog Café documentation: `http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/PrologCafe061/doc/`

SICStus Prolog manual: `http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html`

LLP (Linear Logic Programming): `http://bach.istc.kobe-u.ac.jp/llp/`