A User's Guide to Picat

Version 4.0

Neng-Fa Zhou and Jonathan Fruhman

Copyright ©https://picat-lang.org, 2013-2026.
Last updated November 27, 2026

Preface

Picat is a general-purpose language that incorporates features from logic programming, functional programming, constraint programming, and scripting languages. The letters in the name summarize Picat's features:

- **P**attern-matching: A *predicate* defines a relation, and can have zero, one, or multiple answers. A *function* is a special kind of predicate that always succeeds with *one* answer. Picat is a rule-based language. Predicates and functions are defined with pattern-matching rules. Picat also supports Prolog-style Horn clauses and Definite Clause Grammar (DCG) rules.

- **I**ntuitive: Picat provides assignment and loop statements for programming everyday things. An assignable variable mimics multiple logic variables, each of which holds a value at a different stage of computation. Assignments are useful for computing aggregates and are used with the `foreach` loop for implementing list and array comprehensions.

- **C**onstraints: Picat supports constraint programming. Given a set of variables, each of which has a domain of possible values, and a set of constraints that limit the acceptable set of assignments of values to variables, the goal is to find an assignment of values to the variables that satisfies all of the constraints. Picat provides four solver modules: `cp`, `sat`, `smt`, and `mip`. These four modules follow the same interface, which allows for seamless switching from one solver to another.

- **A**ctors: Actors are event-driven calls. Picat provides *action rules* for describing event-driven behaviors of actors. Events are posted through channels. An actor can be attached to a channel in order to watch and to process its events. All the propagators used in `cp` are implemented as actors.

- **T**abling: Tabling can be used to store the results of certain calculations in memory, allowing the program to do a quick table lookup instead of repeatedly calculating a value. As computer memory grows, tabling is becoming increasingly important for offering dynamic programming solutions for many problems. The `planner` module, which is implemented by the use of tabling, has been shown to be an efficient tool for solving planning problems.

The support of unification, nondeterminism, tabling, and constraints makes Picat more suitable than functional and scripting languages for symbolic computations. Picat is more convenient than Prolog for scripting and modeling. With arrays, loops, and comprehensions, it is not rare to find problems for which Picat requires an order of magnitude fewer lines of code to describe than Prolog. Picat is more scalable than Prolog. The use of pattern-matching rather than unification facilitates indexing of rules. Picat is not as powerful as Prolog for metaprogramming and it's impossible to write a meta-interpreter for Picat in Picat itself. Nevertheless, this weakness can be remedied with library modules for implementing domain-specific languages.

The Picat implementation is based on the B-Prolog engine. The current implementation is ready for many kinds of applications. It also serves as a foundation for new additions. The project is open, and you are welcome to join as a developer, a sponsor, a user, or a reviewer. Please contact `picat@picat-lang.org` and join the news group `https://groups.google.com/g/picat-lang`.

License

The copyright of Picat is owned by https://picat-lang.org. Picat is provided, free of charge, for any purposes, including commercial ones. The C source files of Picat are covered by the Mozilla Public License, v. 2.0 (http://mozilla.org/MPL/2.0/). In essence, anyone is allowed to build works, including proprietary ones, based on Picat, as long as the Source Code Form is retained. The copyright holders, developers, and distributors will not be held liable for any direct or indirect damages.

Acknowledgements

The initial design of Picat was published in December 2012, and the first alpha version was released in May 2013. Many people have contributed to the project by reviewing its ideas, design, implementation, or documentation. We thank Roman Bart'ak, Nikhil Barthwal, Mike Bionchik, Lei Chen, Veronica Dahl, Claudio Cesar de S'a, Agostino Dovier, Sergii Dymchenko, Julio Di Egidio, Christian Theil Have, Håkan Kjellerstrand, Annie Liu, Nuno Lopes, Marcio Minicz, Richard O'Keefe, Lorenz Schiffmann, Paul Tarau, and Jan Wielemaker.

We are especially grateful to Håkan Kjellerstrand, who has been programming in Picat and blogging about it since May 2013. The hundreds of Picat programs he has shared at https://github.com/hakank/hakank/ helped the system mature quickly.

We also thank Bo Yuan (Bobby) Zhou for designing the Picat website (https://picat-lang.org), Sanders Hernandez for implementing the interface to the FANN neural network library, and Domingo Alvarez Duarte for porting Picat to MinGW (https://github.com/mingodad/picat).

Finally, we thank Cristian Grozea for creating Picat's experimental branch (https://github.com/picat-lang/Picat/tree/experimental), which hosts several early-stage projects. Some of his experiments led to features now available in the official branch, including lambda definitions and the external SAT interface.

The Picat implementation is based on the B-Prolog engine. It uses the following public domain modules: token.c by Richard O'Keefe; getline.c by Chris Thewalt; Espresso (by Berkeley); FANN by Steffen Nissen. In addition, Picat also provides interfaces to the SAT solver Kissat (https://github.com/arminbiere/kissat), Gurobi by Gurobi Optimization, Inc, Cbc by John Forrest, GLPK by Andrew Makhorin, SCIP by the Zuse Institute, and z3 by Microsoft.

The Picat project was supported in part by the NSF, CUNY RF, and others.

Contents

1 Overview **1**
- 1.1 Data Types . . . 1
- 1.2 Defining Predicates . . . 5
- 1.3 Defining Functions . . . 7
- 1.4 Assignments and Loops . . . 8
- 1.5 Tabling . . . 11
- 1.6 Modules . . . 11
- 1.7 Higher-order Calls and Anonymous Definitions . . . 13
  - 1.7.1 Higher-order Call Conventions . . . 13
  - 1.7.2 Anonymous Definitions . . . 14
  - 1.7.3 Higher-order Routines . . . 15
- 1.8 Constraints . . . 15
- 1.9 Exceptions . . . 16
- 1.10 Action Rules . . . 17
- 1.11 Prebuilt Maps . . . 18
- 1.12 Programming Exercises . . . 19

2 How to Use the Picat System **20**
- 2.1 How to Use the Picat Interpreter . . . 20
  - 2.1.1 How to Enter and Quit the Picat Interpreter . . . 20
  - 2.1.2 How to Use the Command-line Editor . . . 21
  - 2.1.3 How to Compile and Load Programs . . . 21
  - 2.1.4 How to Run Programs . . . 23
  - 2.1.5 How to Run Programs Directly . . . 23
  - 2.1.6 Creating Standalone Executables . . . 24
- 2.2 How to Use the Debugger . . . 24

3 Data Types, Operators, and Built-ins **26**
- 3.1 Variables . . . 28
- 3.2 Atoms . . . 29
- 3.3 Numbers . . . 30
- 3.4 Compound Terms . . . 33
  - 3.4.1 Lists . . . 33
  - 3.4.2 Strings . . . 36
  - 3.4.3 Structures . . . 36
  - 3.4.4 Arrays . . . 37
  - 3.4.5 Maps . . . 39
  - 3.4.6 Sets . . . 39
  - 3.4.7 Heaps . . . 40

3.5 Equality Testing, Unification, and Term Comparison . . . 40
3.5.1 Numerical Equality . . . 41
3.5.2 Ordering of Terms . . . 42
3.6 Expressions . . . 42
3.7 Higher-order Predicates and Functions . . . 43
3.8 Other Built-ins in the `basic` Module . . . 44

4 Predicates and Functions **47**
4.1 Predicates . . . 47
4.2 Functions . . . 48
4.3 Patterns and Pattern-Matching . . . 49
4.4 Goals . . . 50
4.5 Tail Recursion . . . 51

5 Assignments and Loops **53**
5.1 Assignments . . . 53
5.1.1 If-Else . . . 53
5.2 Types of Loops . . . 54
5.2.1 Foreach Loops . . . 54
5.2.2 Foreach Loops with Multiple Iterators . . . 55
5.2.3 Foreach Loops with the `break` Condition . . . 56
5.2.4 While Loops . . . 56
5.2.5 Do-while Loops . . . 57
5.3 List and Array Comprehensions . . . 58
5.4 Compilation of Loops . . . 58
5.4.1 List Comprehensions . . . 60

6 Exceptions **62**
6.1 Built-in Exceptions . . . 62
6.2 Throwing Exceptions . . . 63
6.3 Defining Exception Handlers . . . 63

7 Tabling **64**
7.1 Table Declarations . . . 64
7.2 The Tabling Mechanism . . . 68

8 The `planner` Module **69**
8.1 Depth-Bounded Search . . . 70
8.2 Depth-Unbounded Search . . . 72
8.3 Examples . . . 74

9 Modules **76**
9.1 Module and Import Declarations . . . 76
9.2 Binding Calls to Definitions . . . 77
9.3 Binding Higher-order Calls . . . 78
9.4 Library Modules . . . 78

10 Higher-order Calls and Anonymous Definitions **79**
10.1 Higher-order Call Conventions . . . 79
10.1.1 Arity and Supplying Arguments . . . 80
10.2 Anonymous Functions . . . 80
10.2.1 Parameter Patterns . . . 81
10.3 Anonymous Predicates . . . 81
10.4 Context Variables and Nested Definitions . . . 81
10.5 Passing Anonymous Definitions as Arguments . . . 82
10.6 Modules and Call Resolution . . . 83
10.7 Implementation of Anonymous Definitions . . . 83
10.8 Compiled Bodies and Callable Terms . . . 84

11 I/O **85**
11.1 Opening a File . . . 85
11.2 Reading from a File . . . 86
11.2.1 End of File . . . 88
11.3 Writing to a File . . . 89
11.4 Flushing and Closing a File . . . 91
11.5 Standard File Descriptors . . . 91

12 Event-Driven Actors and Action Rules **92**
12.1 Channels, Ports, and Events . . . 92
12.2 Action Rules . . . 93
12.3 Lazy Evaluation . . . 95
12.4 Constraint Propagators . . . 96

13 Constraints **97**
13.1 Domain Variables . . . 98
13.2 Table Constraints . . . 99
13.3 Arithmetic Constraints . . . 100
13.4 Boolean Constraints . . . 102
13.5 Global Constraints . . . 103
13.6 Bit-Vector Constraints (sat only) . . . 109
13.7 Solver Invocation . . . 110
13.7.1 Common Solving Options . . . 111
13.7.2 Solving Options for `cp` . . . 111
13.7.3 Solving Options for `sat` . . . 112
13.7.4 Solving Options for `mip` . . . 112
13.7.5 Solving Options for `smt` . . . 113
13.8 Posting and Solving Multiple Constraint Problems . . . 113

14 The `nn` (Neural Networks) Module **114**
14.1 Creating, Printing, and Destroying Neural Networks . . . 115
14.2 Activation Functions . . . 116
14.3 Training Data . . . 117
14.4 Training Neural Networks . . . 118
14.5 Saving and Loading Neural Networks . . . 118
14.6 Running Neural Networks . . . 119

15 The `os` Module **120**
15.1 The $Path$ Parameter . . . 120
15.2 Directories . . . 120
15.2.1 The Current Working Directory . . . 121
15.3 Modifying Files and Directories . . . 121
15.3.1 Creation . . . 121
15.3.2 Deletion . . . 121
15.4 Obtaining Information about Files . . . 122
15.5 Environment Variables . . . 123

A The `math` Module **124**
A.1 Constants . . . 124
A.2 Functions . . . 124
A.2.1 Sign and Absolute Value . . . 124
A.2.2 Rounding and Truncation . . . 125
A.2.3 Exponents, Roots, and Logarithms . . . 125
A.2.4 Converting Between Degrees and Radians . . . 126
A.2.5 Trigonometric Functions . . . 126
A.2.6 Hyperbolic Functions . . . 127
A.2.7 Random Numbers . . . 128
A.2.8 Other Built-ins . . . 129

B The `sys` Module **130**
B.1 Compiling and Loading Programs . . . 130
B.2 Tracing Execution . . . 131
B.2.1 Debugging Commands . . . 131
B.3 Information about the Picat System . . . 132
B.3.1 Statistics . . . 132
B.3.2 Time . . . 134
B.3.3 Other System Information . . . 134
B.4 Garbage Collection . . . 135
B.5 Quitting the Picat System . . . 135

C The `util` Module **136**
C.1 Utilities on Terms . . . 136
C.2 Utilities on Strings and Lists . . . 136
C.3 Utilities on Matrices . . . 137
C.4 Utilities on Lists and Sets . . . 138

D The `ordset` Module **139**

E The `datetime` Module **140**

F Formats **141**
F.1 Formatted Printing . . . 141

G External Language Interface with C **142**
G.1 Term Representation . . . 142
G.2 Fetching Arguments of Picat Calls . . . 142
G.3 Testing Picat Terms . . . 142

G.4 Converting Picat Terms into C 143
G.5 Manipulating and Writing Picat Terms 144
G.6 Building Picat Terms 144
G.7 Registering C-defined Predicates 144

H Tokens **147**

I Grammar **151**

J Operators **157**

K The Library Modules **158**

Index **165**

Chapter 1

Overview

Before we give an overview of the Picat language, let us briefly describe how to use the Picat system. The Picat system provides an interactive programming environment for users to load, debug, and execute programs. Users can start the Picat interpreter with the OS command `picat`.

OSPrompt `picat`

Once the interpreter is started, users can type a command line after the prompt `Picat>`. The `help` command shows how to use commands, and the `halt` command terminates the Picat interpreter. Users can also use the `picat` command to run a program directly as follows:

OSPrompt `picat` *File* *Arg*$_1$ *Arg*$_2$ ... *Arg*$_n$

where *File* (with or without the extension `.pi`) is the main file name of the program. The program must define a predicate named `main/0` or `main/1`.[1] If the command line contains arguments after the file name, then `main/1` is executed. Otherwise, if the file name is not followed by any arguments, then `main/0` is executed. When `main/1` is executed, all of the arguments after the file name are passed to the predicate as a list of strings.

The `picat` command accepts several options, which specify an initial setting for the interpreter. For example, the option `-d` starts the interpreter in debug mode, enabling tracing:

OSPrompt `picat -d`

1.1 Data Types

Picat is a dynamically typed language, in which type checking occurs at runtime. A variable in Picat is a value holder. A variable name is an identifier that begins with a capital letter or an underscore. An *attributed variable* is a variable that has a map of attribute-value pairs attached to it. A variable is free until it is bound to a value. A value in Picat can be *primitive* or *compound*.

A primitive value can be an integer, a real number, or an atom. A character can be represented as a single-character atom. An atom name is either an identifier that begins with a lowercase letter or a single-quoted sequence of characters.

A compound value can be a *list* in the form `[`t_1`,...,`t_n`]` or a *structure* in the form `$`s`(`t_1`,...,`t_n`)` where s stands for a structure name, n is called the *arity* of the structure, and each t_i ($1 \leq i \leq n$) is a *term* which is a variable or a value. The preceding dollar symbol is used to distinguish a structure from a function call. *Strings*, *arrays*, *maps*, *sets*, and *heaps* are special compound values. A *string* is a list of single-character atoms. An *array* takes the form `{`t_1`,...,`t_n`}`, which

[1] This document occasionally uses the notation F/N when referring to predicates and functions. When this notation is used, it refers to the predicate or function that is called F, which has N arguments.

is a special structure with the name `'{}'`. A *map* is a hash-table represented as a structure that contains a set of key-value pairs. A *set* is a special map where only keys are used. A *heap* is a complete binary tree represented as an array. A heap can be a *min-heap* or a *max-heap*.

For each data type, Picat provides a set of built-in predicates and functions. For example, the function `new_struct(`$Name, IntOrList$`)` returns a structure. The function `new_map(`S`)` returns a map that initially contains the pairs in list S, where each pair has the form $Key = Val$. The function `new_set(`S`)` returns a *map set* that initially contains the elements in list S. A *map-set* is a map in which every key is mapped to the atom `not_a_value`. The function `new_array(`$I_1, I_2, \ldots, I_n$`)` returns an n-dimensional array, where each I_i is an integer expression specifying the size of a dimension. An n-dimensional array is a one-dimensional array where the arguments are (n-1)-dimensional arrays.

Example

```
Picat> V1 = X1, V2 = _ab, V3 = _          % variables

Picat> N1 = 12, N2 = 0xf3, N3 = 1.0e8     % numbers

Picat> A1 = x1, A2 = '_AB', A3 = ''       % atoms

Picat> L = [a,b,c,d]                      % a list

Picat> write("hello" ++ "picat")          % strings
[h,e,l,l,o,p,i,c,a,t]

Picat> print("hello" ++ "picat")
hellopicat

Picat> writef("%s", "hello" ++ "picat")   % formatted write
hellopicat

Picat> writef("%-5d %5.2f", 2, 2.0)       % formatted write
2      2.00

Picat> S = $point(1.0, 2.0)               % a structure

Picat> S = new_struct(point, 3)           % create a structure
S = point(_3b0, _3b4, _3b8)

Picat> A = {a,b,c,d}                      % an array

Picat> A = new_array(3)                   % create an array
A = {_3b0,_3b4,_3b8}

Picat> M = new_map([one = 1, two = 2])    % create a map
M =  (map)[two = 2,one = 1]

Picat> M = new_set([one,two,three])       % create a map set
M = (map)[two,one,three]
```

```
Picat> X = 1..2..10                          % ranges
X = [1,3,5,7,9]

Picat> X = 1..5
X = [1,2,3,4,5]
```

Picat allows function calls in arguments. For this reason, it requires structures to be preceded with a dollar symbol in order for them to be treated as data. Without the dollar symbol, the command `S = point(1.0, 2.0)` would call the function `point(1.0, 2.0)` and bind `S` to its return value. In order to ensure safe interpretation of meta-terms in higher-order calls, Picat forbids the creation of terms that contain structures with the name `'.'`, index notations, array comprehensions, list comprehensions, loops, and higher-order calls. While these constructs cannot be preserved as literal syntax structures, their evaluated results can occur in constructed terms.

For each type, Picat provides a set of built-in functions and predicates. The index notation X`[`I`]`, where X is a variable that references a compound value and I is an expression that evaluates to an integer or an integer range of the form $l..u$, is a special function that returns a single component (when I is an integer) or a slice of components (when I is a range) of X. The index of the first element of a list or a structure is 1. In order to facilitate type checking at compile time, Picat does not overload arithmetic operators for other purposes, and requires an index expression to be an integer or an integer range.

A list comprehension, which takes the following form, is a special functional notation for creating lists:

`[`T `:` E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`]`

where T is an expression, each E_i is an iterating pattern, each D_i is an expression that gives a compound value, and the optional conditions $Cond_1$, ..., $Cond_n$ are callable terms. This list comprehension means that for every tuple of values $E_1 \in D_1$, ..., $E_n \in D_n$, if the conditions are true, then the value of T is added into the list.

An array comprehension takes the following form:

`{`T `:` E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`}`

It is the same as:

`to_array([`T `:` E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`])`

The predicate `put(`Map, Key, Val`)` attaches the key-value pair Key=Val to the map Map, where Key is a non-variable term, and Val is any term. The function `get(`Map, Key`)` returns Val of the key-value pair Key=Val attached to Map. The predicate `has_key(`Map, Key`)` succeeds iff Map contains a pair with the given key.

An attributed variable has a map attached to it. The predicate `put_attr(`X, Key, Val`)` attaches the key-value pair Key=Val to X. The function `get_attr(`X, Key`)` returns Val of the key-value pair Key=Val attached to X.

Example

```
Picat> integer(5)
yes
```

```
Picat> real(5)
no

Picat> var(X)
yes

Picat> X = 5, var(X)
no

Picat> 5 != 2+2
yes

Picat> X = to_binary_string(5)
X = ['1','0','1']

Picat> L = [a,b,c,d], X = L[2]
X = b

Picat> L = [(A,I) : A in [a,b], I in 1..2].
L = [(a,1),(a,2),(b,1),(b,2)]

Picat> put_attr(X, one, 1), One = get_attr(X, one)
One = 1

Picat> S = new_struct(point, 3), Name = name(S), Len = length(S)
S = point(_3b0, _3b4, _3b8)
Name = point
Len = 3

Picat> S = new_array(2, 3), S[1, 1] = 11, D2 = length(S[2])
S = {{11,_93a0,_93a4},{_938c,_9390,_9394}}
D2 = 3

Picat> M = new_map(), put(M, one, 1), One = get(M, one)
One = 1

Picat> M = new_set(), put(M, one), has_key(M, one).
```

Picat also allows OOP notations, called *dot notations*, for calling predicates and functions. The notation $A_1.f(A_2,\ldots,A_k)$ is the same as $f(A_1,A_2,\ldots,A_k)$, unless A_1 is an atom, in which case A_1 must be a module qualifier for f. The notation $A.f$, where f is an atom, is the same as the call $A.f()$.

Example

```
Picat> X = 5.to_binary_string()
X = ['1','0','1']

Picat> X = 5.to_binary_string().length
X = 3
```

```
Picat> X.put(one, 1), One = X.get(one)
One = 1

Picat> X = math.pi
X = 3.141592653589793

Picat> S = new_struct(point, 3), Name = S.name, Len = S.length
S = point(_3b0, _3b4, _3b8)
Name = point
Len = 3

Picat> S = new_array(2, 3), S[1, 1] = 11, D2 = S[2].length
S = {{11,_93a0,_93a4},{_938c,_9390,_9394}}
D2 = 3

Picat> M = new_map(), M.put(one, 1), One = M.get(one).
One = 1
```

1.2 Defining Predicates

A predicate call either succeeds or fails, unless an exception occurs. A predicate call can return multiple answers through backtracking. The built-in predicate `true` always succeeds, and the built-in predicate `fail` (or `false`) always fails. A *goal* is made from predicate calls and statements, including conjunction (A, B), disjunction $(A; B)$, negation (not A), if statements, `foreach` loops, and `while` loops.

A predicate is defined with pattern-matching rules. Picat has two types of pattern-matching rules:

- Non-backtrackable rule: $Head, Cond$ `=>` $Body$.
- Backtrackable rule: $Head, Cond$ `?=>` $Body$.

The $Head$ takes the form $p(t_1, \ldots, t_n)$, where p is called the predicate name, and n is called the arity. When $n = 0$, the parentheses can be omitted. The condition $Cond$, which is an optional goal, specifies a condition under which the rule is applicable. For a call C, if C matches $Head$ and $Cond$ succeeds, meaning that the condition evaluates to true, the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites C into $Body$. If the used rule is non-backtrackable, then the rewriting is a commitment, and the program can never backtrack to C. If the used rule is backtrackable, however, the program will backtrack to C once $Body$ fails, meaning that $Body$ will be rewritten back to C, and the next applicable rule will be tried on C.

Example

```
fib(0, F) => F = 1.
fib(1, F) => F = 1.
fib(N, F), N > 1 => fib(N-1, F1), fib(N-2, F2), F = F1+F2.
fib(N, F) => throw $error(wrong_argument, fib, N).
```

A call matches the head `fib(0, F)` if the first argument is 0. The second argument can be anything. For example, for the call `fib(0, 2)`, the first rule is applied, since `fib(0, 2)` matches its head. However, when the body is executed, the call `2 = 1` fails.

The predicate `fib/2` can also be defined using an if-statement as follows:

```
fib(N, F) =>
    if (N = 0; N = 1)
        F = 1
    elseif (N > 1)
        fib(N-1, F1), fib(N-2, F2), F = F1+F2
    else
        throw $error(wrong_argument, fib, N)
    end.
```

An `if` statement takes the form `if` $(Cond)$ $Goal_1$ `else` $Goal_2$ `end`.[2] The `if` statement can contain one or more `elseif` clauses. The `else` part can be omitted. In that case the else part is assumed to be `else true`. The built-in `throw` E throws term E as an exception.

Example

```
member(X, [Y|_]) ?=> X = Y.
member(X, [_|L]) => member(X, L).
```

The pattern `[Y|_]` matches any non-empty list. The backtrackable rule makes a call nondeterministic, and the predicate can be used to retrieve elements from a list one at a time through backtracking.

```
Picat> member(X, [1,2,3])
X = 1;
X = 2;
X =
3;
no
```

After Picat returns an answer, users can type a semicolon immediately after the answer to ask for the next answer. If users only want one answer to be returned from a call, they can use `once` $Call$ to stop backtracking.

The version of `member` that checks if a term occurs in a list can be defined as follows:

```
membchk(X, [X|_]) => true.
membchk(X, [_|L]) => membchk(X, L).
```

The first rule is applicable to a call if the second argument is a list and the first argument of the call is identical to the first element of the list.

Horn Clauses

Picat supports Prolog-style Horn clauses. A Horn clause takes the form $Head$:- $Body$. When $Body$ is `true`, the clause can be written as $Head$. Let $Head$ be $p(t_1, \ldots, t_n)$. The Horn clause can be translated equivalently to the following pattern-matching rule:

$p(V_1, \ldots, V_n)$?=> $V_1 = \$t_1, \ldots, V_n = \$t_n, Body.$

[2] Picat also accepts Prolog-style if-then-else in the form `(If -> Then; Else)`. It will issue a syntax warning if the else-part is missing.

where the dollar symbols indicate that all of the t_i's are terms, rather than function calls.

A predicate definition that consists of Horn clauses can be preceded by an *index declaration* in the form

`index` $(M_{11}, \ldots, M_{1n})$ `...` $(M_{m1}, \ldots, M_{mn})$

where each M_{ij} is either $+$ (meaning indexed) or $-$ (meaning not indexed). When a predicate defined by Horn clauses is not preceded by an index declaration, Picat automatically generates one for the predicate. For each index pattern $(M_{i1}, \ldots, M_{in})$, the compiler generates a version of the predicate that indexes all of the $+$ arguments. An index declaration affects the predicate's efficiency but not its behavior. It is assumed that there is a default version of the predicate that has none of its arguments indexed.

Example

```
index (+, -) (-, +)
edge(a, b).
edge(a, c).
edge(b, c).
edge(c, b).
```

For a predicate of Horn clauses, a matching version of the predicate is selected for a call. If no matching version is available, Picat uses the default version. For example, for the call `edge(X, Y)`, if both `X` and `Y` are free, then the default version is used.

1.3 Defining Functions

A function call always succeeds with a return value if no exception occurs. Functions are defined with non-backtrackable rules in which the head is an equation F=X, where F is the function pattern in the form $f(t_1, \ldots, t_n)$ and X holds the return value. When $n = 0$, the parentheses can be omitted.

Example

```
fib(0) = F => F = 1.
fib(1) = F => F = 1.
fib(N) = F, N > 1 => F = fib(N-1) + fib(N-2).

qsort([]) = L => L = [].
qsort([H|T]) = L => L = qsort([E : E in T, E =< H]) ++ [H] ++
                        qsort([E : E in T, E > H]).
```

A function call never fails and never succeeds more than once. For function calls such as `fib(-1)` or `fib(X)`, Picat raises an exception.

Picat allows inclusion of *function facts* in the form f`(`t_1`, ...,` t_n`)=`Exp in function definitions.

Example

```
fib(0) = 1.
fib(1) = 1.
```

```
fib(N) = F, N > 1 => F = fib(N-1) + fib(N-2).

qsort([]) = [].
qsort([H|T]) =
    qsort([E : E in T, E =< H]) ++ [H] ++
    qsort([E : E in T, E > H]).
```

Function facts are automatically indexed on all of the input arguments, and hence no index declaration is necessary. Note that while a predicate call with no argument does not need parentheses, a function call with no argument must be followed by parentheses, unless the function is module-qualified, as in `math.pi`.

The `fib` function can also be defined as follows:

```
fib(N) = cond((N = 0; N = 1), 1, fib(N-1)+fib(N-2)).
```

The conditional expression returns 1 if the condition `(N = 0; N = 1)` is true, and the value of `fib(N-1)+fib(N-2)` if the condition is false.

1.4 Assignments and Loops

Picat allows assignments in rule bodies. An assignment takes the form LHS:=RHS, where LHS is either a variable or an access of a component of a compound value in the form X`[...]`. When LHS is an access in the form $X[I]$, the component of X at index I is updated. This update is undone if execution backtracks over this assignment.

Example

```
test => X = 0, X := X+1, X := X+2, write(X).
```

In order to handle assignments, Picat creates new variables at compile time. In the above example, at compile time, Picat creates a new variable, say `X1`, to hold the value of `X` after the assignment `X := X+1`. Picat replaces `X` by `X1` on the LHS of the assignment. It also replaces all of the occurrences of `X` to the right of the assignment by `X1`. When encountering `X1 := X1+2`, Picat creates another new variable, say `X2`, to hold the value of `X1` after the assignment, and replaces the remaining occurrences of `X1` by `X2`. When `write(X2)` is executed, the value held in `X2`, which is 3, is printed. This means that the compiler rewrites the above example as follows:

```
test => X = 0, X1 = X+1, X2 = X1+2, write(X2).
```

Picat supports `foreach` and `while` statements for programming repetitions. A `foreach` statement takes the form

```
foreach (E1 in D1, Cond1, ..., En in Dn, Condn)
   Goal
end
```

where each iterator, E_i *in* D_i, can be followed by an optional condition $Cond_i$. Within each iterator, E_i is an iterating pattern, and D_i is an expression that gives a compound value. The `foreach` statement means that $Goal$ is executed for every possible combination of values $E_1 \in D_1, \ldots, E_n \in D_n$ that satisfies the conditions $Cond_1, \ldots, Cond_n$.[3]

A while statement takes the form

[3] The condition `break(`$Cond$`)` terminates the loop when $Cond$ is true.

```
while (Cond)
   Goal
end
```

It repeatedly executes *Goal* as long as *Cond* succeeds. A variant of the while loop in the form of

```
do
   Goal
while (Cond)
```

executes *Goal* one time before testing *Cond*.

A loop statement forms a name scope. Variables that occur only in a loop, but do not occur before the loop in the outer scope, are local to each iteration of the loop. For example, in the following rule:

```
p(A) =>
    foreach (I in 1 .. A.length)
        E = A[I],
        writeln(E)
    end.
```

the variables `I` and `E` are local, and each iteration of the loop has its own values for these variables.

Example

```
write_map(Map) =>
    foreach ((Key = Value) in Map)
        writef("%w = %w\n", Key, Value)
    end.

sum_list(L) = Sum =>    % returns sum(L)
    S = 0,
    foreach (X in L)
        S := S+X
    end,
    Sum = S.

read_list = List =>
    L = [],
    E = read_int(),
    while (E != 0)
        L := [E|L],
        E := read_int()
    end,
    List = L.
```

The function `read_list` reads a sequence of integers into a list, terminating when 0 is read. The loop corresponds to the following sequence of recurrences:

L = []
L_1 = [e_1|L]
L_2 = [e_2|L_1]

$\ldots$
L_n = [e_n|L_{n-1}]
List = L_n

Note that the list of integers is in reverse order. If users want a list in the same order as the input, then the following loop can be used:

```
read_list = List =>
    List = L,
    E = read_int(),
    while (E != 0)
        L = [E|T],
        L := T,
        E := read_int()
    end,
    L = [].
```

This loop corresponds to the following sequence of recurrences:

L = [e_1|L_1]
L_1 = [e_2|L_2]
$\ldots$
L_{n-1} = [e_n|L_n]
L_n = []

Loop statements are compiled into tail-recursive predicates. For example, the second `read_list` function given above is compiled into:

```
read_list = List =>
    List = L,
    E = read_int(),
    p(E, L, Lout),
    Lout = [].

p(0, Lin, Lout) => Lout = Lin.
p(E, Lin, Lout) =>
    Lin = [E|Lin1],
    NE = read_int(),
    p(NE, Lin1, Lout).
```

A list comprehension is first compiled into a `foreach` loop, and then the loop is compiled into a call to a generated tail-recursive predicate. For example, the list comprehension

```
List = [(A,X) : A in [a,b], X in 1..2]
```

is compiled into the following loop:

```
List = L,
foreach (A in [a,b], X in 1..2)
    L = [(A,X)|T],
    L := T
end,
L = [].
```

1.5 Tabling

A predicate defines a relation where the set of facts is implicitly generated by the rules. The process of generating the facts may never end and/or may contain a lot of redundancy. Tabling can prevent infinite loops and redundancy by memorizing calls and their answers. In order to have all calls and answers of a predicate or function tabled, users just need to add the keyword `table` before the first rule.

Example

```
table
fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1) + fib(N-2).
```

When not tabled, the function call `fib(N)` takes exponential time in `N`. When tabled, however, it takes only linear time.

Users can also give table modes to instruct the system on what answers to table. Mode-directed tabling is especially useful for dynamic programming problems. In mode-directed tabling, a plus-sign (+) indicates input, a minus-sign (-) indicates output, `max` indicates that the corresponding variable should be maximized, `min` indicates that the corresponding variable should be minimized, and `nt` indicates that the corresponding argument is not tabled.[4]

Example

```
table(+, +, min)
edit([], [], D) => D = 0.
edit([X|Xs], [X|Ys], D) =>
    edit(Xs, Ys, D).
edit(Xs, [Y|Ys], D) ?=>       % insert
    edit(Xs, Ys, D1),
    D = D1+1.
edit([X|Xs], Ys, D) =>        % delete
    edit(Xs, Ys, D1),
    D = D1+1.
```

For a call `edit(L1, L2, D)`, where `L1` and `L2` are given lists and `D` is a variable, the rules can generate all facts, each of which contains a different edit distance between the two lists. The table mode `table(+, +, min)` tells the system to keep a fact with the minimal edit distance.

A tabled predicate can be preceded by both a table declaration and at most one index declaration if it contains facts. The order of these declarations is not important.

1.6 Modules

A module is a source file with the extension `.pi`. A module begins with a module name declaration and optional import declarations. A module declaration has the form:

`module` *Name*.

[4] An `nt` argument can carry some information that is dependent on the input arguments but useful for the computation.

where $Name$ must be the same as the main file name. A file that does not begin with a module declaration is assumed to belong to the global module, and all of the predicates and functions that are defined in such a file are visible to all modules as well as the top-level of the interpreter.

An import declaration takes the form:

`import` $Name_1$`, ...,` $Name_n$`.`

where each $Name_i$ is a module name. When a module is imported, all of its public predicates and functions will be visible to the importing module. A public predicate or function in a module can also be accessed by preceding it with a module qualifier, as in `m.p()`, but the module still must be imported.

Atoms and structure names do not belong to any module, and are globally visible. In a module, predicates and functions are assumed to be visible both inside and outside of the module, unless their definitions are preceded by the keyword `private`.

Example

```
% in file my_sum.pi
module my_sum.

my_sum(L) = Sum =>
   sum_aux(L, 0, Sum).

private
sum_aux([], Sum0, Sum) => Sum = Sum0.
sum_aux([X|L], Sum0, Sum) => sum_aux(L, X+Sum0, Sum).

% in file test_my_sum.pi
module test_my_sum.
import my_sum.

go =>
   writeln(my_sum([1,2,3,4])).
```

The predicate `sum_aux` is private, and is never visible outside of the module. The following shows a session that uses these modules.

```
Picat> load("test_my_sum")

Picat> go
10
```

The command `load(File)` loads a module file into the system. If the file has not been compiled, then the `load` command compiles the file before loading it. If this module is dependent on other modules, then the other modules are loaded automatically if they are not yet in the system.[5] When a module is loaded, all of its public predicates and functions become visible to the interpreter.

The Picat module system is static, meaning that the binding of normal calls to their definitions takes place at compile time. For higher-order calls, however, Picat may need to search for their definitions at runtime. Several built-in modules are imported by default, including `basic`, `io`,

[5]Dependent modules must be in a path that is specified by the environment variable PICATPATH, unless their source files are located at the importing source file's directory.

math, and sys. For a normal call that is not higher-order in a module, the Picat compiler searches modules for a definition in the following order:

1. The implicitly imported built-in modules in the order from basic, math, io to sys.
2. The enclosing module of the call.
3. The explicitly imported modules in the order that they were imported.
4. The global module.

1.7 Higher-order Calls and Anonymous Definitions

1.7.1 Higher-order Call Conventions

A predicate or function is said to be *higher-order* if it takes calls as arguments. The built-ins call, apply, and find_all are higher-order. The predicate call(S, Arg_1, ..., Arg_n), where S is an atom or a structure naming a callable, calls the predicate named by S with the arguments that are specified in S together with extra arguments Arg_1, ..., Arg_n. The function apply(S, Arg_1, ..., Arg_n) is similar to call, except that apply returns a value. The function findall($Template$, S) returns a list of all possible solutions of call(S) in the form of $Template$. Other higher-order predicates and functions include:

- \+/1 (not)
- call_cleanup/2
- catch/3
- count_all/1
- freeze/2
- maxof/2-3
- maxof_inc/2-3
- minof/2-3
- minof_inc/2-3
- not/1
- once/1
- time/1
- time2/1
- time_out/3

All of these higher-order built-ins are defined in the basic module, except for time/1, time2/1, and time_out/3, which are defined in the sys module. Literal callable terms cannot contain assignments or loops; compiled anonymous bodies can contain both.

The higher-order predicate call call(S, Arg_1, ..., Arg_n) and the function call apply(S, Arg_1, ..., Arg_n), where S is a variable at compile time, can both be written as S(Arg_1, ..., Arg_n). Picat translates this notation into a call or apply invocation depending on the context. Higher-order calls can also be module-qualified: m.S(Arg_1, ..., Arg_n) invokes the function or predicate defined in module m whose symbol is referenced by S. Within a term constructor, a higher-order call whose name is a variable is always treated as a function call.

Example

```
Picat> S = $member(X), S([1,2,3])
X = 1;
X = 2;
X = 3;
no
```

```
Picat> L = findall(X, member(X, [1,2,3])).
L = [1,2,3]

Picat> Z = apply('+', 1, 2)
Z = 3

Picat> F = $'*'(2), Z = F(2)
F = '*'(2)
Z = 4

Picat> F = $'*'(2), Z = $p(F(2))
F = '*'(2)
Z = p(4)
```

1.7.2 Anonymous Definitions

An anonymous definition creates a function or predicate without giving it a name. An anonymous function, also called a lambda expression, has the form

```
([A1,A2,...,An] -> Res, B1,B2,...,Bm)
```

where the bracketed list contains parameter patterns, `Res` is the return expression, and the remaining goals form the body. The body is executed before the return expression is evaluated. If no body is needed, the comma and body can be omitted. An anonymous predicate has the form

```
([A1,A2,...,An] => B1,B2,...,Bm)
```

It has a body but no return expression. Parameters are terms, as in named rule heads, and follow the same pattern-matching conventions. An empty parameter list denotes a definition with no explicit arguments.

Evaluating an anonymous definition creates a callable value, called a *closure*; its body is executed only when it is invoked. Closures use the same variable-call conventions described above. For example:

```
Offset = 10,
Add = ([X] -> X+Offset),
Offset := 20,
Y = Add(5),                    % Y = 15
Print = ([X] => println(X)),
Print(Y)
```

The function closure captures the value `10` of `Offset` when it is created. The later assignment to `Offset` does not change that captured value. The predicate closure prints `15` when invoked.

An anonymous definition forms a name scope, like a `foreach` loop. Variables that have already occurred in the enclosing scope are global to the definition; variables that first occur inside the definition are fresh locals. In the example, `Offset` is a context variable of `Add`, while each parameter `X` is local to its definition. The body can contain the usual goals, including assignments, conditionals, and loops. An anonymous predicate uses a single `=>` rule; no `?=>` form is needed.

The compiler introduces a unique dummy name in the global module and compiles the anonymous definition like a named definition. Its body is compiled in the defining module, preserving that module's call resolution. The closure can consequently be passed to another module and invoked without importing its defining module. Chapter 10 describes anonymous definitions and higher-order call conventions in detail.

1.7.3 Higher-order Routines

Among the higher-order built-ins, `findall` is special in that it forms a name scope like a loop. Local variables that occur in a `findall` call are not visible to subsequent calls in the body or query.

The call `apply` never returns a partially evaluated function. If the number of arguments does not match the required number, then it throws an exception.

Example

```
map(F, []) = [].
map(F, [X|Xs]) = [F(X) | map(F, Xs)].

map2(F, [], []) = [].
map2(F, [X|Xs], [Y|Ys]) = [F(X,Y) | map2(F,Xs,Ys)].

fold(F, Acc, []) = Acc.
fold(F, Acc, [H|T]) = fold(F, F(H, Acc), T).
```

For example, the definitions above accept anonymous functions:

```
Ys = map(([X] -> X*X),[1,2,3]),     % Ys = [1,4,9]
S = fold(([X,A] -> X+A),0,Ys)       % S = 14
```

For named higher-order calls whose targets are unknown at compile time, runtime resolution can require a search of loaded modules. Such calls can incur additional overhead compared with direct calls.

1.8 Constraints

Picat can be used as a modeling and solving language for constraint satisfaction and optimization problems. A constraint program normally poses a problem in three steps: (1) generate variables; (2) generate constraints over the variables; and (3) call `solve` to find a valuation for the variables that satisfies the constraints, and possibly optimizes an objective function. Picat provides four solver modules, including `cp`, `sat`, `smt`, and `mip`.

Example

```
import cp.

go =>
    Vars = [S,E,N,D,M,O,R,Y],  % generate variables
    Vars :: 0..9,
    all_different(Vars),       % generate constraints
    S #!= 0,
    M #!= 0,
    1000*S+100*E+10*N+D+1000*M+100*O+10*R+E
         #= 10000*M+1000*O+100*N+10*E+Y,
    solve(Vars),               % search
    writeln(Vars).
```

In arithmetic constraints, expressions are treated as data, and it is unnecessary to enclose them with dollar-signs.

The loops provided by Picat facilitate modeling of many constraint satisfaction and optimization problems. The following program solves a Sudoku puzzle:

```
import cp.

sudoku =>
    instance(N, A),
    A :: 1..N,
    foreach (Row in 1..N)
        all_different(A[Row])
    end,
    foreach (Col in 1..N)
        all_different([A[Row,Col] : Row in 1..N])
    end,
    M = floor(sqrt(N)),
    foreach (Row in 1..M, Col in 1..M)
        Square = [A[Row1,Col1] :
                    Row1 in (Row-1)*M+1..Row*M,
                    Col1 in (Col-1)*M+1..Col*M],
        all_different(Square)
    end,
    solve(A),
    foreach (I in 1..N) writeln(A[I]) end.

instance(N, A) =>
    N = 9,
    A = {{2,_,_,6,7,_,_,_,_},
         {_,_,6,_,_,_,2,_,1},
         {4,_,_,_,_,_,8,_,_},
         {5,_,_,_,_,9,3,_,_},
         {_,3,_,_,_,_,_,5,_},
         {_,_,2,8,_,_,_,_,7},
         {_,_,1,_,_,_,_,_,4},
         {7,_,8,_,_,_,6,_,_},
         {_,_,_,_,5,3,_,_,8}}.
```

Recall that variables that occur within a loop, and do not occur before the loop in the outer scope, are local to each iteration of the loop. For example, in the third `foreach` statement of the `sudoku` predicate, the variables `Row`, `Col`, and `Square` are local, and each iteration of the loop has its own values for these variables.

1.9 Exceptions

An exception is an event that occurs during the execution of a program that requires special treatment. In Picat, an exception is just a term. The exception `interrupt` is raised when `ctrl-c` is typed during a program's execution. The built-in predicate `throw` $Exception$ throws $Exception$.

All exceptions, including those raised by built-ins and interruptions, can be caught by catchers. A catcher is a call in the form: `catch(`$Goal$`, `$Exception$`, `$Handler$`)` which is equivalent to $Goal$, except when an exception is raised during the execution of $Goal$ that unifies with $Exception$. When such an exception is raised, all of the bindings that have been performed on variables in $Goal$ will be undone, and $Handler$ will be executed to handle the exception.

The call `call_cleanup(`$Goal$`, `$Cleanup$`)` is equivalent to `call(`$Goal$`)`, except that $Cleanup$ is called when $Goal$ succeeds deterministically (i.e., with no remaining choice point), when $Goal$ fails, or when $Goal$ raises an exception.

1.10 Action Rules

Picat provides action rules for describing event-driven actors. An actor is a predicate call that can be delayed, and can be activated later by events. Each time an actor is activated, an action can be executed. A predicate for actors contains at least one *action rule* in the form:

$$Head, Cond, \{Event\} \texttt{=>}\ Body$$

where $Head$ is an actor pattern, $Cond$ is an optional condition, $Event$ is a non-empty set of event patterns separated by `','`, and $Body$ is an action. For an actor and an event, an action rule is said to be *applicable* if the actor matches $Head$ and $Cond$ is true. A predicate for actors cannot contain backtrackable rules.

An event channel is an attributed variable to which actors can be attached, and through which events can be posted to actors. A channel has four ports: `ins`, `bound`, `dom`, and `any`. An event pattern in $Event$ specifies the port to which the actor is attached. The event pattern `ins(`X`)` attaches the actor to the `ins`-port of channel X, and the actor will be activated when X is instantiated. The event pattern `event(`X`, `T`)` attaches the actor to the `dom`-port of channel X. The built-in `post_event(`X`, `T`)` posts an event term T to the `dom`-port of channel X. After an event is posted to a port of a channel, the actors attached to that port are activated. For an activated actor, the system searches for an applicable rule and executes the rule body if it finds one. After execution, the actor is *suspended*, waiting to be activated again by other events. Picat does not provide a built-in for detaching actors from channels. An actor *fails* if no rule is applicable to it when it is activated or the body of the applied rule fails. An actor becomes a *normal call* once a normal non-backtrackable rule is applied to it.

Example

```
echo(X, Flag), var(Flag), {event(X,T)} => writeln(T).
echo(_X, _Flag) => writeln(done).

foo(Flag) => Flag = 1.
```

When a call `echo(X, Flag)` is executed, where `Flag` is a variable, it is attached to the `dom`-port of `X` as an actor. The actor is then suspended, waiting for events posted to the `dom`-port. For this actor definition, the command

```
echo(X, Flag), post_event(X, hello), post_event(X, picat).
```

prints out `hello` followed by `picat`. If the call `foo(Flag)` is inserted before the second call to `post_event`, then `var(Flag)` fails when the actor is activated the second time, causing the second rule to be applied to the actor. Then, the output will be `hello` followed by `done`. Note that events are not handled until a non-inline call is executed. Replacing `foo(Flag)` by `Flag = 1` will result in a different behavior because `Flag = 1` is an inline call.

1.11 Prebuilt Maps

Picat has three kinds of prebuilt maps: *heap* maps, *global* maps, and *table* maps. Prebuilt heap maps are created on the heap immediately after the system is started. The built-in function `get_heap_map(`ID`)` returns the heap map that is associated with ID, where ID must be a ground term. If no heap map is associated with ID, then this function establishes an association between ID and an unused heap map, and returns the map. A heap map is like a normal map. Users use `put` to add key-value pairs into the map. Users use `get` to retrieve a value that is associated with a key in the map. Changes made to a heap map since a choice point are undone when execution backtracks to that choice point. The built-in function `get_heap_map()` returns the heap map that is associated with a system-generated default identifier. There are an unlimited number of prebuilt heap maps.

Global maps are created in the global area when the Picat system is started. The built-in function `get_global_map(`ID`)` returns the global map that is associated with ID, where ID must be a ground term. If no global map is associated with ID, then this function establishes an association between ID and an unused global map, and returns the map. A big difference between a global map and a heap map is that changes to the global map are not undone upon backtracking. When a key-value pair is added into the global map, the variables in the value term are numbered before they are copied to the global area. If the value term contains attributed variables, then the attributes of the variables are not copied, and are therefore lost. When retrieving a value that is associated with a key, the value term in the global area is copied back to the heap after all of the numbered variables are unnumbered. The built-in function `get_global_map()` returns the global map that is associated with a system-generated default identifier. The number of prebuilt global maps is 97, and the system halts if a program requests more than 97 global maps.

Table maps are created in the table area when the Picat system is started. The built-in function `get_table_map(`ID`)` returns the table map that is associated with ID, where ID must be a ground term. If no table map is associated with ID, then this function establishes an association between ID and an unused table map, and returns the map. Like the global map, changes to a table map are not undone upon backtracking. Unlike the global map, however, keys and values are *hash-consed* so that common ground sub-terms are not replicated in the table area. The built-in function `get_table_map()` returns the table map that is associated with a system-generated default identifier. The number of prebuilt table maps is 97, and the system halts if a program requests more than 97 table maps.

The advantage of using prebuilt maps is that data can be accessed everywhere without being passed as arguments, and the disadvantage is that it affects locality of data and thus the readability of programs. In tabled programs, using prebuilt maps is discouraged because it may cause unanticipated effects.

Example

```
go ?=>
    get_heap_map(h1).put(one, 1),
    get_global_map(g1).put(one, 1),
    get_table_map(t1).put(one, 1),
    fail.
go =>
    if (get_heap_map(h1).has_key(one))
       writef("heap map h1 has key%n")
    else
       writef("heap map h1 has no key%n")
```

```
    end,
    if (get_global_map(g1).has_key(one))
       writef("global map g1 has key%n")
    else
       writef("global map g1 has no key%n")
    end,
    if (get_table_map(t1).has_key(one))
       writef("table map t1 has key%n")
    else
       writef("table map t1 has no key%n")
    end.
```

For the call `go`, the output is:

```
heap map h1 has no key
global map g1 has key
table map t1 has key
```

The `fail` call in the first rule causes execution to backtrack to the second rule. After backtracking, the pair added to the heap map by the first rule is lost, but the pair added to the global map and the pair added to the table map remain.

1.12 Programming Exercises

Project Euler (`projecteuler.net`) is an excellent platform for practicing programming and problem solving skills. You can find Picat solutions to some of the problems at `picat-lang.org/projects.html`. Select five problems from the Project Euler problem set for which no solutions have been posted, and write a program in Picat for each of them.

Chapter 2

How to Use the Picat System

2.1 How to Use the Picat Interpreter

The Picat system is written in both C and Picat. The Picat interpreter is provided as a single standalone executable file, named `picat.exe` for Windows and `picat` for Unix. The Picat interpreter provides an interactive programming environment for users to compile, load, debug, and execute programs. In order to start the Picat interpreter, users first need to open an OS terminal. In Windows, this can be done by selecting `Start -> Run` and typing `cmd` or selecting `Start -> Programs -> Accessories -> Command Prompt`. In order to start the Picat interpreter in any working directory, the environment variable `PATH` must be properly set to contain the directory where the executable is located.

2.1.1 How to Enter and Quit the Picat Interpreter

The Picat interpreter is started with the OS command `picat`.

$OSPrompt$ `picat`

where $OSPrompt$ is the OS prompt. After the interpreter is started, it responds with the prompt `Picat>`, and is ready to accept queries.

In general, the `picat` command takes the following form:

`picat` $Opts$ $PicatMainFileName$ A_1 A_2 ... A_n

where $PicatMainFileName$ can have the extension `.pi` and can contain a path of directories, and $Opts$ is a sequence of the following options:

- `-d`: This option puts Picat in debug mode after Picat is started. In debug mode, execution can be traced, and the stack trace is printed once an error occurs.
- `-g` $InitGoal$: This option makes Picat execute a specified initial query $InitGoal$ rather than the default `main` predicate.
- `-help`: Display help info.
- `-log`: The option `-log` makes the system print log information and warning messages.
- `-path` $Directories$: $Directories$ is a semicolon-separated and double-quoted list of directories. This option sets the value of the environment variable `PICATPATH` before the execution of the program. The Picat system will look for $PicatMainFileName$ and the related modules in these directories.

- `-s` $Size$: This option reserves $Size$ words for the stack and the heap when the system is started.
- `-v`: This option makes Picat print the version number.
- `-version`: This option is the same as `-v`.

Once the interpreter is started, users can type a query after the prompt. For example,

```
Picat> X = 1+1
X = 2
Picat> printf("hello"++" picat")
hello picat
```

The `halt` predicate, or the `exit` predicate, terminates the Picat interpreter. An alternative way to terminate the interpreter is to enter `ctrl-d` (control-d) when the cursor is located at the beginning of an empty line.

2.1.2 How to Use the Command-line Editor

The Picat interpreter uses the `getline` program written by Chris Thewalt. The `getline` program memorizes up to 100 of the most recent queries that the users have typed, and allows users to recall past queries and edit the current query by using Emacs editing commands. The following gives the editing commands:

`ctrl-f`	Move the cursor one position forward.
`ctrl-b`	Move the cursor one position backward.
`ctrl-a`	Move the cursor to the beginning of the line.
`ctrl-e`	Move the cursor to the end of the line.
`ctrl-d`	Delete the character under the cursor.
`ctrl-h`	Delete the character to the left of the cursor.
`ctrl-k`	Delete the characters to the right of the cursor.
`ctrl-u`	Delete the whole line.
`ctrl-p`	Load the previous query in the buffer.
`ctrl-n`	Load the next query in the buffer.

Note that the command `ctrl-d` terminates the interpreter if the line is empty and the cursor is located at the beginning of the line.

2.1.3 How to Compile and Load Programs

A Picat program is stored in one or more text files with the extension name `pi`. A file name is a string of characters. Picat treats both `'/'` and `'\'` as path separators. Nevertheless, since `'\'` is used as the escape character in quoted strings, two consecutive backslashes must be used, as in `"c:\\work\\myfile.pi"`, if `'\'` is used as the separator.

A program file can contain `include` directives of the form:

`include` $Name$.

The `include` directive causes the content of the file named $Name$ to be copied verbatim to where the directive occurs.

For the sake of demonstration we assume the existence of a file named `welcome.pi` in the current working directory that stores the following program:

```
main =>
    print(" Welcome to PICAT's world! \n ").

main(Args) =>
    print(" Welcome to PICAT's world! \n"),
    foreach (Arg in Args)
        printf("%s\n", Arg)
    end.
```

- `cl(`$FileName$`)`: A program first needs to be compiled and loaded into the system before it can be executed. The built-in predicate `cl(`$FileName$`)` compiles and loads the source file named $FileName$`.pi`. Note that if the full path of the file name is not given, then the file is assumed to be in the current working directory. Also note that users do not need to give the extension name. The system compiles and loads not only the source file $FileName$`.pi`, but also all of the module files that are either directly imported or indirectly imported by the source file. The system searches for such dependent files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`. For $FileName$`.pi`, the `cl` command loads the generated bytecodes without creating a bytecode file. For example,

```
Picat> cl("welcome")
Compiling:: welcome.pi
welcome.pi compiled in 4 milliseconds
```

- `cl`: The built-in predicate `cl` (with no argument) compiles and loads a program from the console, ending when the end-of-file character (`ctrl-z` for Windows and `ctrl-d` for Unix) is typed.

- `compile(`$FileName$`)`: The built-in predicate `compile(`$FileName$`)` compiles the file $FileName$`.pi` and all of its dependent module files without loading the generated bytecode files. The destination directory for the bytecode file is the same as the source file's directory. If the Picat interpreter does not have permission to write into the directory in which a source file resides, then this built-in throws an exception. For example,

```
Picat> compile("welcome")
Compiling::welcome.pi
welcome.pi compiled in 4 milliseconds
```

- `load(`$FileName$`)`: The built-in predicate `load(`$FileName$`)` loads the bytecode file $FileName$`.qi` and all of its dependent bytecode files. For $FileName$ and its dependent file names, the system searches for a bytecode file in the directory in which $FileName$`.qi` resides or the directories that are stored in the environment variable `PICATPATH`. If the bytecode file $FileName$`.qi` does not exist but the source file $FileName$`.pi` exists, then this built-in compiles the source file and loads the bytecodes without creating a `qi` file.

```
Picat> load("welcome")
loading...welcome.qi
```

2.1.4 How to Run Programs

After a program is loaded, users can query the program. For each query, the system executes the program, and reports `yes` when the query succeeds and `no` when the query fails. When a query that contains variables succeeds, the system also reports the bindings for the variables. For example,

```
Picat> cl("welcome")
Compiling:: welcome.pi
welcome.pi compiled in 4 milliseconds
loading...

yes

Picat> main
 Welcome to PICAT's world!

yes
```

Users can ask the system to find the next solution by typing `';'` after a solution if the query has multiple solutions. For example,

```
Picat> member(X, [1,2,3])
X = 1;
X = 2;
X = 3;
no
```

Users can force a program to terminate by typing `ctrl-c`, or by letting it execute the built-in predicate `abort`. Note that when the system is engaged in certain tasks, such as garbage collection, users may need to wait for a while in order to see the termination after they type `ctrl-c`.

2.1.5 How to Run Programs Directly

Programs that define the `main/0` predicate or the `main/1` predicate can be run directly as an OS command. For example,

```
$ picat welcome
 Welcome to PICAT's world!

$ picat welcome a b c
 Welcome to PICAT's world!
a
b
c
```

The '`$`' sign is the prompt of the OS. It is assumed that the environment variable `PATH` has been set to contain the directory of the executable `picat` (`picat.exe` for Windows), and the environment variable `PICATPATH` has been set to contain the directory of the `welcome.pi` file or the file is in the current working directory.

2.1.6 Creating Standalone Executables

It is possible to create a script that can be run as a standalone executable. For example, consider the following script `welcome.exe` for Linux:

```
#!/bin/bash
picat welcome.pi
echo " Finished!"
```

Once the environment variables `PATH` and `PICATPATH` are set properly, and the script has execute permission, it can be executed as follows:

```
$ welcome.exe
 Welcome to PICAT's world!
 Finished!
```

2.2 How to Use the Debugger

The Picat system has three execution modes: *non-trace mode*, *trace mode*, and *spy mode*. In trace mode, it is possible to trace the execution of a program, showing every call in every possible stage. In order to trace the execution, the program must be recompiled while the system is in trace mode. In spy mode, it is possible to trace the execution of individual functions and predicates that are *spy points*. When the Picat interpreter is started, it runs in non-trace mode. The predicate `debug` or `trace` changes the mode to trace. The predicate `nodebug` or `notrace` changes the mode to non-trace.

In trace mode, the debugger displays execution traces of queries. An *execution trace* consists of a sequence of call traces. Each *call trace* is a line that consists of a stage, the number of the call, and the information about the call itself. For a function call, there are two possible stages: `Call`, meaning the time at which the function is entered, and `Exit`, meaning the time at which the call is completed with an answer. For a predicate call, there are two additional possible stages: `Redo`, meaning a time at which execution backtracks to the call, and `Fail`, meaning the time at which the call is completed with a failure. The information about a call includes the name of the call, and the arguments. If the call is a function, then the call is followed by = and `?` at the `Call` stage, and followed by = $Value$ at the `Exit` stage, where $Value$ is the return value of the call.

Consider, for example, the following program:

```
p(X) ?=> X = a.
p(X) => X = b.
q(X) ?=> X = 1.
q(X) => X = 2.
```

Assume the program is stored in a file named `myprog.pi`. The following shows a trace for a query:

```
Picat> debug

{Trace mode}
Picat> cl(myprog)

{Trace mode}
Picat> p(X), q(Y)
```

```
   Call: (1) p(_328) ?
   Exit: (1) p(a)
   Call: (2) q(_378) ?
   Exit: (2) q(1)
X = a
Y = 1 ?;
   Redo: (2) q(1) ?
   Exit: (2) q(2)
X = a
Y = 2 ?;
   Redo: (1) p(a) ?
   Exit: (1) p(b)
   Call: (3) q(_378) ?
   Exit: (3) q(1)
X = b
Y = 1 ?;
   Redo: (3) q(1) ?
   Exit: (3) q(2)
X = b
Y = 2 ?;
no
```

In trace mode, the debugger displays every call in every possible stage. Users can set *spy points* so that the debugger only shows information about calls of the symbols on which users have set spy points. Users can use the predicate

`spy` $Name/N$

to set the functor $Name/N$ as a spy point, where the arity N is optional. If the functor is defined in multiple loaded modules, then all these definitions will be treated as spy points. If no arity is given, then any functor of $Name$ is treated as a spy point, regardless of the arity.

After displaying a call trace, if the trace is for stage `Call` or stage `Redo`, then the debugger waits for a command from the users. A command is either a single letter followed by a carriage-return, or just a carriage-return. See Appendix B for the debugging commands.

Chapter 3

Data Types, Operators, and Built-ins

Picat is a dynamically typed language, in which type checking occurs at runtime. A variable gets a type once it is bound to a value. In Picat, variables and values are terms. A value can be *primitive* or *compound*. A primitive value can be an *integer*, a *real number*, or an *atom*. A compound value can be a *list* or a *structure*. *Strings*, *arrays*, *maps*, *sets*, and *heaps* are special compound values. This chapter describes the data types and the built-ins for each data type that are provided by the `basic` module.

Many of the built-ins are given as operators. Table 3.1 shows all of the operators that are provided by Picat. Unless the table specifies otherwise, the operators are left-associative. The as-pattern operator (`@`) and the operators for composing goals, including `not`, `once`, conjunction (`,` and `&&`), and disjunction (`;` and `||`), will be described in Chapter 4 on Predicates and Functions. The constraint operators (the ones that begin with `#`) will be described in Chapter 13 on Constraints. In Picat, no new operators can be defined, and none of the existing operators can be redefined.

The dot operator (`.`) is used in OOP notations for calling predicates and functions. It is also used to qualify calls with a module name. The notation $A_1.f(A_2, \ldots, A_k)$ is the same as $f(A_1, A_2, \ldots, A_k)$, unless A_1 is an atom, in which case A_1 must be a module qualifier for f. If an atom needs to be passed as the first argument to a function or a predicate, then this notation cannot be used. The notation $A.f$, where f is an atom, is the same as the call $A.f()$. For example, the expression S`.name` returns the name of S, and the expression S`.arity` returns the arity of S if S is a structure. Note that the dot operator is left-associative. For example, the expression `X.f().g()` is the same as `g(f(X))`. Also note that functions in dot notation are always evaluated eagerly, and therefore should not be used in term constructs or constraint expressions.

The following functions are provided for all terms:

- `copy_term(`$Term_1$`)` = $Term_2$: This function copies $Term_1$ into $Term_2$. If $Term_1$ is an attributed variable, then $Term_2$ will not contain any of the attributes.

- `copy_term_shallow(`$Term_1$`)` = $Term_2$: This function copies the skeleton of $Term_1$ into $Term_2$. If $Term_1$ is a variable or an atomic value, then it returns a complete copy of $Term_1$, the same as `copy_term(`$Term_1$`)`; if $Term_1$ is a list, then it returns a cons `[`H`|`T`]` where both the car H and the cdr T are free variables; otherwise, it is the same as `new_struct(name(`$Term_1$`), arity(`$Term_1$`))`.

- `hash_code(`$Term$`)` = Int: This function returns the hash code of $Term$. If $Term$ is a variable, then the returned hash code is always 0.

- `to_codes(`$Term$`)` = $List$: This function returns a list of character codes of $Term$.

Table 3.1: Operators in Picat

Precedence	Operators
Highest	`.`, `@`
	`**` (right-associative)
	unary +, unary −, `~`
	`*`, `/`, `//`, `/<`, `/>`, `div`, `mod`, `rem`
	binary +, binary −
	`>>`, `<<`
	`/\`
	`^`
	`\/`
	`..`
	`++` (right-associative)
	`is`, `=`, `!=`, `:=`, `==`, `!==`, `=:=`, `=\=`, `<`, `=<`, `<=`, `>`, `>=`, `::`, `in`, `notin`, `=..` `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, `#>=`, `@<`, `@=<`, `@<=`, `@>`, `@>=`
	`#~`
	`#/\`
	`#^`
	`#\/`
	`#=>` (right-associative)
	`#<=>`
	`not`, `once`, `\+`
	`,` (right-associative), && (right-associative)
Lowest	`;` (right-associative), || (right-associative)

- `to_fstring(`$Format$`, ` $Args\ldots$`) = ` $String$: This function converts the arguments in the $Args\ldots$ parameter into a string, according to the format string $Format$, and returns the string. The number of arguments in $Args\ldots$ cannot exceed 10. Format characters are described in Chapter 11.

- `to_string(`$Term$`) = ` $String$: This function returns a string representation of $Term$.

Other built-ins on terms are given in Sections 3.5 and 3.8.

3.1 Variables

Variables in Picat, like variables in mathematics, are value holders. Unlike variables in imperative languages, Picat variables are not symbolic addresses of memory locations. A variable is said to be *free* if it does not hold any value. A variable is *instantiated* when it is bound to a value. Picat variables are *single-assignment*, which means that after a variable is instantiated to a value, the variable will have the same identity as the value. After execution backtracks over a point where a binding took place, the value that was assigned to a variable will be dropped, and the variable will be turned back into a free variable.

A variable name is an identifier that begins with a capital letter or an underscore. For example, the following are valid variable names:

```
X1   _   _ab
```

The name _ is used for *anonymous variables*. In a program, different occurrences of _ are treated as different variables. So the test `_ == _` is always false.

The following two built-ins are provided to test whether a term is a free variable:

- `var(`$Term$`)`: This predicate is true if $Term$ is a free variable.

- `nonvar(`$Term$`)`: This predicate is true if $Term$ is not a free variable.

An *attributed variable* is a variable that has a map of attribute-value pairs attached to it. The following built-ins are provided for attributed variables:

- `attr_var(`$Term$`)`: This predicate is true if $Term$ is an attributed variable.

- `bool_dvar(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable whose lower bound is 0 and whose upper bound is 1.

- `dvar(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable.

- `dvar_or_int(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable or an integer.

- `get_attr(`X`, `Key`) = `Val: This function returns the Val of the pair Key=Val that is attached to X. It throws an error if X has no attribute named Key.

- `get_attr(`X`, `Key`, `$DefaultVal$`) = `Val: This function returns Val of the pair Key=Val that is attached to X. It returns $DefaultVal$ if X does not have the attribute named Key.

- `put_attr(`X`, `Key`)`: This is the same as `put_attr(`X`, `Key`, not_a_value)`.

- `put_attr(`X`, `Key`, `Val`)`: This predicate attaches the pair Key=Val to X, where Key is a non-variable term, and Val is any term.

3.2 Atoms

An atom is a symbolic constant. An atom name can either be quoted or unquoted. An unquoted name is an identifier that begins with a lowercase letter, followed by an optional string of letters, digits, and underscores. A quoted name is a single-quoted sequence of arbitrary characters. A character can be represented as a single-character atom. For example, the following are valid atom names:

```
x   x_1   '_'   '\\'   'a\'b\n'   '_ab'   '$%'
```

An atom name cannot span more than one line. An atom name cannot contain more than 1000 characters. The backslash character `'\'` is used as the escape character. So, the name `'a\'b\n'` contains four characters: `a`, `'`, `b`, and `\n`.

The following built-ins are provided for atoms:

- `ascii_alpha(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single letter.
- `ascii_alpha_digit(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single letter or digit.
- `ascii_digit(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single digit.
- `ascii_lowercase(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single lowercase letter.
- `ascii_uppercase(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single uppercase letter.
- `atom(`$Term$`)`: This predicate is true if $Term$ is an atom.
- `atom_chars(`Atm`)` = $String$: This function returns a string that contains the characters of the atom Atm. It throws an error if Atm is not an atom.
- `atom_codes(`Atm`)` = $List$: This function returns the list of codes of the characters of the atom Atm. It throws an error if Atm is not an atom.
- `atomic(`$Term$`)`: This predicate is true if $Term$ is an atom or a number.
- `char(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single character.
- `chr(`$Code$`)` = $Char$: This function returns the UTF-8 character of the code point $Code$.
- `digit(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom consists of a single digit.
- `len(`$Atom$`)` = Len: This function returns the number of characters in $Atom$. Note that this function is overloaded in such a way that the argument can also be an array, a list, or a structure.
- `length(`$Atom$`)` = Len: This function is the same as `len(`$Atom$`)`.
- `lowercase(`$Term$`)`: This predicate checks if $Term$ is a lowercase character atom.

- `ord(`$Char$`) = `Int: This function returns the code point of the UTF-8 character $Char$. It throws an error if $Char$ is not a single-character atom.
- `to_atom(`$String$`) = `$Atom$: This function converts $String$ to an atom.
- `uppercase(`$Term$`)`: This predicate checks if $Term$ is an uppercase character atom.

3.3 Numbers

A number can be an integer or a real number. An integer can be a decimal numeral, a binary numeral, an octal numeral, or a hexadecimal numeral. In a numeral, digits can be separated by underscores, but underscore separators are ignored by the tokenizer. For example, the following are valid integers:

`12_345`	a decimal numeral
`0b100`	4 in binary notation
`0o73`	59 in octal notation
`0xf7`	247 in hexadecimal notation

A real number consists of an integer part, a decimal fraction preceded by a decimal point, and an optional exponent. For example, the following are valid real numbers.

```
12.345   0.123   12.0e-10   0.12E10
```

Table 3.2 gives the meaning of each of the numeric operators in Picat, from the operator with the highest precedence (`**`) to the one with the lowest precedence (`=:=`). Except for the power operator `**`, which is right-associative, all of the arithmetic operators are left-associative.

In addition to the numeric operators, the `basic` module also provides the following built-ins for numbers:

- `between(`$From$`, `To`, `X`)` (nondet): If X is bound to a number, then this predicate determines whether X is between $From$ and To. Otherwise, if X is unbound, then this predicate nondeterministically selects X from the numbers that are between $From$ and To with step 1. It is the same as `member(`X`, [`E` : `E` in `$From$`..`To`])`, but it does not create the list.
- `between(`$From$`, `$Step$`, `To`, `X`)` (nondet): This predicate nondeterministically selects X from the numbers that are between $From$ and To with $Step$, where $From$, To, and $Step$ must be numbers. It is the same as

  ```
  member(X, [E : E in From..Step..To])
  ```

  but it does not create the list.
- `bigint(`$Term$`)`: This predicate is true if $Term$ is a big integer.
- `float(`$Term$`)`: This predicate is true if $Term$ is a real number.
- `int(`$Term$`)`: This predicate is true if $Term$ is an integer.
- `integer(`$Term$`)`: The same as `int(`$Term$`)`.
- `max(`X`, `Y`) = `Val: This function returns the maximum of X and Y, where X and Y are terms.

Table 3.2: Arithmetic Operators

X `**` Y	power
`+`X	same as X
`-`X	sign reversal
`~`X	bitwise complement
X `*` Y	multiplication
X `/` Y	division
X `//` Y	integer division, truncated
X `/>` Y	integer division (ceiling(X / Y))
X `/<` Y	integer division (floor(X / Y))
X `div` Y	integer division, floored
X `mod` Y	modulo, same as X - floor(X div Y) * Y
X `rem` Y	remainder (X - (X // Y) * Y)
X `+` Y	addition
X `-` Y	subtraction
X `>>` Y[1]	right shift
X `<<` Y	left shift
X `/\` Y	bitwise and
X `^` Y	bitwise xor
X `\/` Y	bitwise or
$From$ `..` $Step$ `..` To	A range (list) of numbers with a step
$From$ `..` To	A range (list) of numbers with step 1
X `=:=` Y	pretty much (numerically) equal

- `maxint_small() = `Int: This function returns the maximum integer that is represented in one word. All integers that are greater than this integer are represented as *big integers*.
- `min(`X`, `Y`) = `Val: This function returns the minimum of X and Y, where X and Y are terms.
- `minint_small() = `Int: This function returns the minimum integer that is represented in one word. All integers that are smaller than this integer are represented as *big integers*.
- `number(`$Term$`)`: This predicate is true if $Term$ is a number.
- `number_chars(`Num`) = `$String$: This function returns a list of characters of Num. This function is the same as `to_fstring("%d", `Num`)` if Num is an integer, and is similar to `to_fstring("%g", `Num`)` if Num is a real number.
- `number_codes(`Num`) = `$List$: This function returns a list of codes of the characters of Num. It is the same as `number_chars(`Num`).to_codes()`.
- `real(`$Term$`)`: This predicate is the same as `float(`$Term$`)`.
- `to_binary_string(`Int`) = `$String$: This function returns the binary representation of the integer Int as a string.
- `to_float(`NS`) = `$Real$:[2] This function is the same as NS`*1.0` if NS is a number, and the same as `1.0*parse_term(`NS`)` if NS is a string.
- `to_hex_string(`Int`) = `$String$: This function returns the hexadecimal representation of the integer Int as a string.
- `to_int(`NCS`) = `Int:[3] This function is the same as `truncate(`NCS`)` in the `math` module if NCS is a number, the same as `ord(`NCS`)-ord('0')` if NCS is a digit character, and the same as `to_int(parse_term(`NCS`))` if NCS is a string.
- `to_integer(`NCS`) = `Int: This function is the same as `to_int(`NCS`)`.
- `to_number(`NCS`) = `Num: This function is the same as NCS if NCS is a number, the same as `ord(`NCS`)-ord('0')` if NCS is a digit character, and the same as `to_number(parse_term(`NCS`))` if NCS is a string.
- `to_oct_string(`Int`) = `$String$: This function returns the octal representation of the integer Int as a string.
- `to_radix_string(`Int`, `$Base$`) = `$String$: This function returns the string representing integer Int in base $Base$, where $Base$ must be greater than 1 and less than 37. The call `to_oct_string(`Int`)` is the same as `to_radix_string(`Int`, 8)`.
- `to_real(`NS`) = `$Real$: This function is the same as `to_float(`NS`)`.

The `math` module provides more numeric functions. See Appendix A.

[2] Here, NS is a number or a string.
[3] Here, NCS is a number, a character, or a string.

3.4 Compound Terms

A compound term can be a *list* or a *structure*. Components of compound terms can be accessed with subscripts. Let X be a variable that references a compound value, and let I be an integer expression that evaluates to an integer or an integer range. The index notation X`[`I`]`, where X is a variable referencing a compound value, is a special function that returns the Ith component of X if I is an integer or a slice of components if I is a range in the form of $l..u$, counting from the beginning.[4] Subscripts begin at 1, meaning that X[1] is the first component of X. An index notation can take multiple subscripts. For example, the expression `X[1, 2]` is the same as `T[2]`, where `T` is a temporary variable that references the component that is returned by `X[1]`. The predicate `compound(`$Term$`)` is true if $Term$ is a compound term.

3.4.1 Lists

A list takes the form `[`t_1`,...,`t_n`]`, where each t_i ($1 \leq i \leq n$) is a term. Let L be a list. The expression L`.length`, which is the same as `length(`L`)`, returns the length of L. Note that a list is represented internally as a singly linked list. Also note that the length of a list is not stored in memory; instead, it is recomputed each time that the function `length` is called.

The symbol `'|'` is not an operator, but a separator that separates the first element (so-called *car*) from the rest of the list (so-called *cdr*). The *cons* notation `[`H`|`T`]` can occur in a pattern or in an expression. When it occurs in a pattern, it matches any list in which H matches the car and T matches the cdr. When it occurs in an expression, it builds a list from H and T. The notation `[`A_1`,`A_2`,...,`A_n`|`T`]` is a shorthand for `[`A_1`|[`A_2`|...[`A_n`|`T`]...]`. So `[a,b,c]` is the same as `[a|[b|[c|[]]]]`.

The `basic` module provides the following built-ins on lists, most of which are overloaded for strings (3.4.2) and arrays (see 3.4.4).

- $List_1$ `++` $List_2$ `=` $List$: This function returns the concatenated list of $List_1$ and $List_2$.
- `append(`X`,` Y`,` Z`)` (nondet): This predicate is true if appending Y to X can create Z. This predicate may backtrack if X is not a complete list.[5]
- `append(`W`,` X`,` Y`,` Z`)` (nondet): This predicate is defined as:

```
append(W, X, Y, Z) => append(W, X, WX), append(WX, Y, Z).
```

- `avg(`$List$`)` `=` Val: This function returns the average of all the elements in $List$. This function throws an exception if $List$ is not a list or any of the elements is not a number.
- `delete(`$List$`,` X`)` `=` $ResList$: This function deletes the first occurrence of X from $List$, returning the result in $ResList$. The built-in `!=/2` is used to test if two terms are different. No variables in $List$ or X will be bound after this function call.
- `delete_all(`$List$`,` X`)` `=` $ResList$: This function deletes all occurrences of X from $List$, returning the result in $ResList$. The built-in `!=/2` is used to test if two terms are different.
- `first(`$List$`)` `=` $Term$: This function returns the first element of $List$.

[4] X must reference a list or an array if I is a range.

[5] A list is *complete* if it is empty, or if its tail is complete. For example, `[a,b,c]` and `[X,Y,Z]` are complete, but `[a,b|T]` is not complete if `T` is a variable.

- `flatten(`$List$`) = `$ResList$: This function flattens a list of nested lists into a list. For example, `flatten([[1],[2,[3]]])` returns `[1,2,3]`.

- `head(`$List$`) = `$Term$: This function returns the head of the list $List$. For example, `head([1,2,3])` returns `1`.

- `insert(`$List$`, `$Index$`, `Elm`) = `$ResList$: This function inserts Elm into $List$ at the index $Index$, returning the result in $ResList$. After insertion, the original $List$ is not changed, and $ResList$ is the same as
$List$`.slice(1, `$Index$`-1)++[`Elm`|`$List$`.slice(`$Index$`,`$List$`.length)]`.

- `insert_all(`$List$`, `$Index$`, `$AList$`) = `$ResList$: This function inserts all of the elements in $AList$ into $List$ at the index $Index$, returning the result in $ResList$. After insertion, the original $List$ is not changed, and $ResList$ is the same as
$List$`.slice(1, `$Index$`-1)++`$AList$`++`$List$`.slice(`$Index$`, `$List$`.length)`.

- `insert_ordered(`$List$`, `$Term$`) = `$ResList$: This function inserts $Term$ into the ordered list $List$, such that the resulting list $ResList$ remains sorted.

- `insert_ordered_down(`$List$`, `$Term$`) = `$ResList$: This function inserts $Term$ into the ordered list $List$, such that the resulting list $ResList$ remains in descending order.

- `last(`$List$`) = `$Term$: This function returns the last element of $List$.

- `len(`$List$`) = `Len: This function returns the number of elements in $List$. Note that this function is overloaded in such a way that the argument can also be an atom, an array, or a structure.

- `length(`$List$`) = `Len: This function is the same as `len(`$List$`)`.

- `list(`$Term$`)`: This predicate is true if $Term$ is a list.

- `max(`$List$`) = `Val: This function returns the maximum value that is in $List$, where $List$ is a list of terms.

- `membchk(`$Term$`, `$List$`)`: This predicate is true if $Term$ is an element of $List$.

- `member(`$Term$`, `$List$`)` (nondet): This predicate is true if $Term$ is an element of $List$. When $Term$ is a variable, this predicate may backtrack, instantiating $Term$ to different elements of $List$.

- `min(`$List$`) = `Val: This function returns the minimum value that is in $List$, where $List$ is a list or an array of terms.

- `new_list(`N`) = `$List$: This function creates a new list that has N free variables.

- `new_list(`N`, `$InitVal$`) = `$List$: This function creates a new list that has N elements initialized to $InitVal$.

- `nth(`$Index$`, `$List$`, `$Elem$`)` (nondet): This predicate is true when $Elem$ is the element at index $Index$ in $List$. Counting starts at 1. When $Index$ is a variable, this predicate may backtrack, instantiating $Index$ to a different integer between 1 and `len(`$List$`)`.

- `prod(`$List$`) = `Val: This function returns the product of all of the values in $List$.

- `remove_dups(`$List$`) = `$ResList$: This function removes all duplicate values from $List$, retaining only the first occurrence of each value. The result is returned in $ResList$. Note that an $O(n^2)$ algorithm is used in the implementation. For long lists, `sort_remove_dups` is faster than this function.

- `reverse(`$List$`) = `$ResList$: This function reverses the order of the elements in $List$, returning the result in $ResList$.

- `select(`X`, `$List$`, `$ResList$`)` (nondet): This predicate nondeterministically selects an element X from $List$, and binds $ResList$ to the list after X is removed. On backtracking, it selects the next element.

- `slice(`$List$`, `$From$`, `To`) = `$SList$: This function returns the sliced list of $List$ from index $From$ through index To. $From$ must not be less than 1. It is the same as the index notation $List$[$From$..To].

- `slice(`$List$`, `$From$`) = `$SList$: This function is the same as the following.

  ```
  slice(List, From, List.length)
  ```

- `sort(`$List$`) = `$SList$: This function sorts the elements of $List$ in ascending order, returning the result in $SList$.

- `sort(`$List$`, `$KeyIndex$`) = `$SList$: This function sorts the elements of $List$ by the key index $KeyIndex$ in ascending order, returning the result in $SList$. The elements of $List$ must be compound values and $KeyIndex$ must be a positive integer that does not exceed the length of any of the elements of $List$. In particular, if an element is a list, then $KeyIndex$ must be either 1 or 2. This function is defined as follows:

  ```
  sort(List, KeyIndex) = SList =>
      List1 = [(A, E) : E in List, arg(KeyIndex,E,A)],
      List2 = sort(List1),
      SList = [E : (_,E) in List2].
  ```

- `sort_remove_dups(`$List$`) = `$SList$: This function is the same as the following, but is faster.

  ```
  sort(List).remove_dups()
  ```

- `sort_remove_dups(`$List$`, `$KeyIndex$`) = `$SList$: This function is the same as the following, but is faster.

  ```
  sort(List, KeyIndex).remove_dups()
  ```

- `sort_down(`$List$`) = `$SList$: This function sorts the elements of $List$ in descending order, returning the result in $SList$.

- `sort_down(`$List$`, `$KeyIndex$`) = `$SList$: This function sorts the elements of $List$ by the key index $KeyIndex$ in descending order, returning the result in $SList$.

- `sort_down_remove_dups(`$List$`) = `$SList$: This function is the same as the following, but is faster.

```
sort_down(List).remove_dups()
```

- `sort_down_remove_dups(`$List$`, `$KeyIndex$`) = `$SList$: This function is the same as the following, but is faster.

  ```
  sort_down(List, KeyIndex).remove_dups()
  ```

- `sorted(`$List$`)`: This predicate checks if $List$ is sorted in ascending order.

- `sorted_down(`$List$`)`: This predicate checks if $List$ is sorted in descending order.

- `sum(`$List$`) = `Val: This function returns the sum of all of the values in $List$.

- `tail(`$List$`) = `$Term$: This function returns the tail of the list $List$. For example, the call `tail([1,2,3])` returns `[2,3]`.

- `to_array(`$List$`) = `$Array$: This function converts the list $List$ to an array. The elements of the array are in the same order as the elements of the list.

- `zip(`$List_1$`, `$List_2$`, ..., `$List_n$`) = `$List$: This function makes a list of array tuples. The jth tuple in the list takes the form $\{E_{1j}, \ldots, E_{nj}\}$, where E_{ij} is the jth element in $List_i$. In the current implementation, n can be 2, 3, or 4.

3.4.2 Strings

A *s*tring is represented as a list of single-character atoms. For example, the string `"hello"` is the same as the list `[h,e,l,l,o]`. In addition to the built-ins on lists, the following built-ins are provided for strings:

- `string(`$Term$`)`: This predicate is true if $Term$ is a string.

- `to_lowercase(`$String$`) = `$LString$: This function converts all uppercase alphabetic characters into lowercase characters, returning the result in $LString$.

- `to_uppercase(`$String$`) = `$UString$: This function converts all lowercase alphabetic characters into uppercase characters, returning the result in $UString$.

3.4.3 Structures

A structure takes the form `$`s`(`t_1`, ..., `t_n`)`, where s is an atom, and n is called the *arity* of the structure. The dollar symbol is used to distinguish a structure from a function call. The *functor* of a structure comprises the name and the arity of the structure.

The following types of structures can never denote functions, meaning that they do not need to be preceded by a $ symbol.

Goals:	`(a, b), (a;b), not a, X = Y, X != 100, X > 1`
Constraints:	`X+Y #= 100, X #!= 1`
Arrays:	`{2,3,4}, {P1,P2,P3}`

Picat disallows creation of the following types of structures:

Dot notations:	`math.pi, my_module.f(a)`
Index notations:	`X[1]+2 X[Y[I]]`
Assignments:	`X := Y+Z, X := X+1`
Ranges:	`1..10, 1..2..10`
List comprehensions:	`[X : X in 1..5]`
Array comprehensions:	`{X : X in 1..5}`
If:	`if (X > Y) Z = X else Z = Y end`
Loops:	`foreach (X in L) writeln(X) end`
Higher-order:	`F(X,Y)`

The compiler will report syntax errors when it encounters assignments, if statements, or loops within term constructors, and treat dot notations, index notations, ranges, list & array comprehensions, and higher-order calls as function calls in term constructors.

The following built-ins are provided for structures:

- T `=..` L: The name and the arguments of T comprise the list L. This predicate extracts the name and arguments of T when T is instantiated, and constructs T from the list L when L is instantiated.
- `arg(`I`,` T`,` A`)`: The Ith argument of the term T is A. This predicate is equivalent to: X = T, A = $X[I]$, except when I is 0 or T is a list. When I is 0, the call fails. When T is a list, only two indices, namely 1 and 2, are acceptable. For I = 1, A is unified with the car of the list, and for I = 2, A is unified with the cdr of the list.
- `arity(`$Struct$`)` = $Arity$: This function returns the arity of $Struct$, which must be a structure.
- `functor(`T`,` F`,` N`)`: The principal functor of the term T has the name F and the arity N. This predicate extracts the functor F/N of T when T is instantiated, and constructs a term T of the given functor F/N when T is a variable.
- `len(`$Struct$`)` = $Arity$: This function is the same as `arity(`$Struct$`)`.
- `name(`$Struct$`)` = $Name$: This function returns the name of $Struct$.
- `new_struct(`$Name$`,` $IntOrList$`)` = $Struct$: This function creates a structure that has the name $Name$. If $IntOrList$ is an integer, N, then the structure has N free variable arguments. Otherwise, if $IntOrList$ is a list, then the structure contains the elements in the list.
- `struct(`$Term$`)`: This predicate is true if $Term$ is a structure.
- `to_list(`$Struct$`)` = $List$: This function returns a list of the components of the structure $Struct$.

3.4.4 Arrays

An *array* takes the form `{`t_1`,...,`t_n`}`, which is a special structure with the name `'{}'` and arity n. Note that, unlike a list, an array always has its length stored in memory, so the function `length(`$Array$`)` always takes constant time. Also note that Picat supports constant-time access of array elements, so the index notation `A[I]` takes constant time when I is an integer.

In addition to the built-ins for structures, the following built-ins are provided for arrays:

- `array(`$Term$`)`: This predicate is true if $Term$ is an array.

- `new_array(`D_1`, ..., `D_n`)` = Arr: This function creates an n-dimensional array, where each D_i is an integer expression that specifies the size of a dimension. In the current implementation, n cannot exceed 10.

The following built-ins, which are originally provided for lists (see 3.4.1), are overloaded for arrays:

- $Array_1$ `++` $Array_2$ `=` $Array$
- `avg(`$Array$`)` `=` Val
- `first(`$Array$`)` `=` $Term$
- `last(`$Array$`)` `=` $Term$
- `len(`$Array$`)` `=` Len
- `length(`$Array$`)` `=` Len
- `max(`$Array$`)` `=` Val
- `min(`$Array$`)` `=` Val
- `nth(`$Index$`, `$Array$`, `$Elem$`)` (nondet)
- `prod(`$Array$`)` `=` Val
- `reverse(`$Array$`)` `=` $ResArray$
- `slice(`$Array$`, `$From$`)` `=` $SArray$
- `slice(`$Array$`, `$From$`, `To`)` `=` $SArray$
- `sort(`$Array$`)` `=` $SArray$
- `sort(`$Array$`, `$KeyIndex$`)` `=` $SArray$
- `sort_down(`$Array$`)` `=` $SArray$
- `sort_down(`$Array$`, `$KeyIndex$`)` `=` $SArray$
- `sort_down_remove_dups(`$Array$`)` `=` $SArray$
- `sort_down_remove_dups(`$Array$`, `$KeyIndex$`)` `=` $SArray$
- `sort_remove_dups(`$Array$`)` `=` $SArray$
- `sort_remove_dups(`$Array$`, `$KeyIndex$`)` `=` $SArray$
- `sorted(`$Array$`)`
- `sorted_down(`$Array$`)`
- `sum(`$Array$`)` `=` Val

Note that many of the overloaded built-ins for arrays are not implemented efficiently, but are provided for convenience. For example, `sort(Array)` is implemented as follows:

```
sort(Array) = Array.to_list().sort().to_array().
```

3.4.5 Maps

A *map* is a hash-table that is represented as a structure that contains a set of key-value pairs. The functor of the structure that is used for a map is not important. An implementation may ban access to the name and the arity of the structure of a map. Maps must be created with the built-in function `new_map`, unless they are prebuilt (see Section 1.11). In addition to the built-ins for structures, the following built-ins are provided for maps:

- `clear(`Map`)`: This predicate clears the map Map. It throws an error if Map is not a map.
- `del(`Map`, `Key`)`: This predicate deletes from Map the pair that has the key Key. It throws an error if Map is not a map. It does nothing if Map does not contain a pair with Key.
- `get(`Map`, `Key`) = `Val: This function returns Val of the pair Key=Val in Map. It throws an error if Map does not contain the key Key.
- `get(`Map`, `Key`, `$DefaultVal$`) = `Val: This function returns Val of the pair Key=Val in Map. It returns $DefaultVal$ if Map does not contain Key.
- `has_key(`Map`, `Key`)`: This predicate is true if Map contains a pair with Key.
- `keys(`Map`) = `$List$: This function returns the list of keys of the pairs in Map.
- `map(`$Term$`)`: This predicate is true if $Term$ is a map.
- `map_to_list(`Map`) = `$PairsList$: This function returns a list of Key=Val pairs that constitute Map.
- `new_map() = `Map: This function creates an empty map.
- `new_map(`$IntOrPairsList$`) = `Map: This function creates a map with an initial capacity or an initial list of pairs.
- `new_map(`N`, `$PairsList$`) = `Map: This function creates a map with initial capacity N and initial list of pairs $PairsList$, where each pair has the form Key=Val.
- `put(`Map`, `Key`, `Val`)`: This predicate attaches the pair Key=Val to Map, where Key is a non-variable term, and Val is any term.
- `put(`Map`, `Key`)`: This predicate is the same as `put(`Map`, `Key`, not_a_value)`.
- `size(`Map`) = `$Size$: This function returns the number of pairs in Map.
- `values(`Map`) = `$List$: This function returns the list of values of the pairs in Map.

3.4.6 Sets

A set is a map where every key is associated with the atom `not_a_value`. All of the built-ins for maps can be applied to sets. For example, the built-in predicate `has_key(`Set`, `Elm`)` tests if Elm is in Set. In addition to the built-ins on maps, the following built-ins are provided for sets:

- `new_set() = `Set: This function creates an empty set.
- `new_set(`$IntOrKeysList$`) = `Set: This function creates a set with an initial capacity or an initial list of keys.
- `new_set(`N`, `$KeysList$`) = `Set: This function creates a set with the initial capacity N and the initial list of keys $KeysList$.

3.4.7 Heaps

A heap[6] is a complete binary tree represented as an array. A heap can be a *min-heap* or a *max-heap*. In a min-heap, the value at the root of each subtree is the minimum among all the values in the subtree. In a max-heap, the value at the root of each subtree is the maximum among all the values in the subtree.

- `heap_is_empty(`$Heap$`)`: This predicate is true if $Heap$ is empty.
- `heap_pop(`$Heap$`) = `Elm: This function removes the root element from the heap, and returns the element. As the function updates the heap, it is not pure. The update will be undone when execution backtracks over the call.
- `heap_push(`$Heap$`, `Elm`)`: This predicate pushes Elm into $Heap$ in a way that maintains the heap property. The update to $Heap$ will be undone when execution backtracks over the call.
- `heap_size(`$Heap$`) = `$Size$: This function returns the size of $Heap$.
- `heap_to_list(`$Heap$`) = `$List$: This function returns a list of the elements in $Heap$.
- `heap_top(`$Heap$`) = `Elm: This function returns the element at the root of the heap. If $Heap$ is a min-heap, then the element is guaranteed to be the minimum, and if $Heap$ is a max-heap, then the element is guaranteed to be the maximum.
- `new_max_heap(`$IntOrList$`) = `$Heap$: This function creates a max-heap. If $IntOrList$ is an integer, then it indicates the capacity. Otherwise, if $IntOrList$ is a list, then the max-heap contains the elements in the list in an order that maintains the heap property.
- `new_min_heap(`$IntOrList$`) = `$Heap$: This function creates a min-heap. If $IntOrList$ is an integer, then it indicates the capacity. Otherwise, if $IntOrList$ is a list, then the min-heap contains the elements in the list in an order that maintains the heap property.

Example

```
main =>
    L = [1,3,2,4,5,3,6],
    H = new_min_heap(L),
    N = H.heap_size(),
    S = [H.heap_pop() : _ in 1..N],
    println(S).
```

3.5 Equality Testing, Unification, and Term Comparison

The equality test T_1 `==` T_2 is true if term T_1 and term T_2 are identical. Two variables are identical if they are aliases. Two primitive values are identical if they have the same type and the same internal representation. Two lists are identical if the cars are identical and the cdrs are identical. Two structures are identical if their functors are the same and their components are pairwise identical. The inequality test T_1 `!==` T_2 is the same as `not` T_1 `==` T_2. Note that two terms can be identical even if they are stored in different memory locations. Also note that it takes linear time in the worst case to test whether two terms are identical.

[6]Note that a heap, as a data structure, is different from the heap area, in which data, including heap maps, are stored.

The unification T_1 `=` T_2 is true if term T_1 and term T_2 are already identical, or if they can be made identical by instantiating the variables in the terms. The built-in T_1 `!=` T_2 is true if term T_1 and term T_2 are not unifiable. The predicate `bind_vars(`$Term$`,` Val`)` binds all of the variables in $Term$ to Val.

Example

```
Picat> X = 1
X = 1
Picat> $f(a, b) = $f(a, b)
yes
Picat> [H|T] = [a,b,c]
H = a
T = [b,c]
Picat> $f(X, b) = $f(a, Y)
X = a
Y = b
Picat> bind_vars({X,Y,Z}, a)
Picat> X = $f(X)
```

The last query illustrates the *occurs-check problem*. When binding `X` to `f(X)`, Picat does not check if `X` occurs in `f(X)` for the sake of efficiency. This unification creates a cyclic term, which can never be printed.

When a unification's operands contain attributed variables, the implementation is more complex. When a plain variable is unified with an attributed variable, the plain variable is bound to the attributed variable. When two attributed variables, say Y and O, where Y is younger than O, are unified, Y is bound to O, but Y's attributes are not copied to O. Since garbage collection does not preserve the seniority of terms, the result of the unification of two attributed variables is normally unpredictable.

3.5.1 Numerical Equality

The numerical equality test T_1 `=:=` T_2 is true if term T_1 and term T_2 are pretty much the same numerical value. This means that $T1$ and $T2$ must both be numbers after evaluation. Whereas the test T_1 `==` T_2 fails if one number is an integer and one number is a real number, the test T_1 `=:=` T_2 may succeed if the absolute difference between T_1 and T_2 is less than 0.00000001. Consider the following examples.

Example

```
Picat> 1 == 1.0
no

Picat> 1 =:= 1.0
yes
```

In the first query, 1 is an integer, while 1.0 is a real number, so the equality test fails. However, the second query, which is a numerical equality test, succeeds.

- T_1 `is` T_2: If T_1 is a variable, then the call binds the variable to the result of T_2, which must be a number after evaluation. If T_1 is a non-variable expression, then the call is equivalent to T_1 `=:=` T_2.
- T_1 `=\=` T_2: The expression T_1 is not numerically equal to T_2.

3.5.2 Ordering of Terms

Picat orders terms in the following way:

`var` < `number` < `atom` < `structure` and `array` < `list` and `string`

Variables are ordered by their addresses. Note that the ordering of variables may change after garbage collection. Numbers are ordered by their numerical values. Atoms are ordered lexicographically. Structures are first ordered lexicographically by their names; if their names are the same, then they are ordered by their components. Arrays are ordered as structures with the special name '{}'. Lists and strings are ordered by their elements.

- $Term1$ `@<` $Term2$: The term $Term1$ precedes the term $Term2$ in the standard order. For example, `a @< b` succeeds.
- $Term1$ `@=<` $Term2$: The term $Term1$ either precedes, or is identical to, the term $Term2$ in the standard order. For example, `a @=< b` succeeds.
- $Term1$ `@<=` $Term2$: This is the same as $Term1$ `@=<` $Term2$.
- $Term1$ `@>` $Term2$: The term $Term1$ follows the term $Term2$ in the standard order.
- $Term1$ `@>=` $Term2$: The term $Term1$ either follows, or is identical to, the term $Term2$ in the standard order.

3.6 Expressions

Expressions are made from variables, values, operators, and function calls. Expressions differ from terms in the following ways:

- An expression can contain dot notations, such as `math.pi`.
- An expression can contain index notations, such as `X[I]`.
- An expression can contain ranges, such as `1..2..100`.
- An expression can contain list comprehensions, such as `[X : X in 1..100]`.
- An expression can contain array comprehensions, such as `{X : X in 1..100}`.

A conditional expression, which takes the form `cond(`$Cond$`,` Exp_1`,` Exp_2`)`, is a special kind of function call that returns the value of Exp_1 if the condition $Cond$ is true and the value of Exp_2 if $Cond$ is false.

Note that, except for conditional expressions in which the conditions are made of predicates, no expressions can contain predicates. A predicate is true or false, but never returns any value.

3.7 Higher-order Predicates and Functions

A predicate or function is said to be *higher-order* if it takes calls as arguments. The `basic` module has the following higher-order predicates and functions.

- `apply(`S`, `Arg_1`, ..., `Arg_n`) = `Val: S is an atom or a structure. This function calls the function that is named by S with the arguments that are specified in S, together with extra arguments $Arg_1, \ldots, Arg_n$. This function returns the value that S returns. The `apply` invocation can be written in short as S`(`Arg_1`, ..., `Arg_n`)`.

- `call(`S`, `Arg_1`, ..., `Arg_n`)`: S is an atom or a structure. This predicate calls the predicate that is named by S with the arguments that are specified in S, together with extra arguments Arg_1, ..., Arg_n. The `call` invocation can be written in short as S`(`Arg_1`, ..., `Arg_n`)`.

- `call_cleanup(`$Call$`, `$Cleanup$`)`: This predicate is the same as `call(`$Call$`)`, except that $Cleanup$ is called when $Call$ succeeds deterministically (i.e., with no remaining choice point), when $Call$ fails, or when $Call$ raises an exception.

- `catch(`$Goal$`, `$Exception$`, `$Handler$`)`: This predicate is the same as $Goal$, except when an exception that unifies with $Exception$ is raised during the execution of $Goal$. When such an exception is raised, all of the bindings that have been performed on variables in $Goal$ will be undone, and $Handler$ will be executed to handle the exception.

- `count_all(`$Call$`) = `Int: This function returns the number of all possible instances of `call(`$Call$`)` that are true. For example, `count_all(member(X, [1,2,3]))` returns 3.

- `findall(`$Template$`, `$Call$`) = `$List$: This function returns a list of all possible instances of `call(`$Call$`)` that are true in the form of $Template$. Note that $Template$ is assumed to be a term without function calls, and that $Call$ is assumed to be a predicate call whose arguments can contain function calls. Also note that, like a loop, `findall` forms a name scope. For example, in `findall(f(X), p(X, g(Y)))`, `f(X)` is a term even though it is not preceded with `$`; `g(Y)` is a function call; the variables `X` and `Y` are assumed to be local to `findall` if they do not occur before in the outer scope.

- `find_all(`$Template$`, `$Call$`) = `$List$: This function is the same as the above function.

- `fold(`F`, `Acc`, `$List$`) = `Res: This function combines the elements of $List$ into a single result by repeatedly applying F to each element from left to right and to the updated accumulator. It is defined as follows:

```
fold(F, Acc, []) = Acc.
fold(F, Acc, [H|T]) = fold(F, F(H, Acc), T).
```

- `freeze(`X`, `$Call$`)`: This predicate delays the evaluation of $Call$ until X becomes a non-variable term.

- `map(`$FuncOrList$`, `$ListOrFunc$`) = `$ResList$: This function applies a given function to every element of a given list and returns a list of the results. One of the arguments is a function, and the other is a list. The order of the arguments is not important.

- `map(`$Func$`, `$List1$`, `$List2$`) = `$ResList$: Let

$List1$ = `[`A_1`,...,`A_n`]` and $List2$ = `[`B_1`,...,`B_n`]`.

This function applies the function $Func$ to every pair of elements (A_i, B_i) by calling `apply(`$Func$`, `A_i`, `B_i`)`, and returns a list of the results.

- `maxof(`$Call$`, `$Objective$`)`: This predicate finds a satisfiable instance of $Call$, such that $Objective$ has the maximum value. Here, $Call$ is used as a generator, and $Objective$ is an expression to be maximized. For every satisfiable instance of $Call$, $Objective$ must be a ground expression. For `maxof`, search is restarted with a new bound each time that a better answer is found.

- `maxof(`$Call$`, `$Objective$`, `$ReportCall$`)`: This predicate is the same as `maxof(`$Call$`, `$Objective$`)`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.

- `maxof_inc(`$Call$`, `$Objective$`)`: This predicate is the same as `maxof(`$Call$`, `$Objective$`)`, except that search continues rather than being restarted each time that a better solution is found.

- `maxof_inc(`$Call$`, `$Objective$`, `$ReportCall$`)`: This predicate is the same as `maxof_inc/2`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.

- `minof(`$Call$`, `$Objective$`)`: This predicate finds a satisfiable instance of $Call$, such that $Objective$ has the minimum value.

- `minof(`$Call$`, `$Objective$`, `$ReportCall$`)`: This predicate is the same as `minof(`$Call$`, `$Objective$`)`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.

- `minof_inc(`$Call$`, `$Objective$`)`: This predicate is the same as `minof(`$Call$`, `$Objective$`)`, except that search continues rather than being restarted each time that a better solution is found.

- `minof_inc(`$Call$`, `$Objective$`, `$ReportCall$`)`: This predicate is the same as `minof_inc/2`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.

- `reduce(`$Func$`, `$List$`)` = Res: If $List$ is a list that contains only one element, this function returns the element. If $List$ contains at least two elements, then the first two elements A_1 and A_2 are replaced with `apply(`$Func$`, `A_1`, `A_2`)`. This step is repeatedly applied to the list until the list contains a single element, which is the final value to be returned. The order of the arguments is not important, meaning that the first argument can be a list and the second one can be a function.

- `reduce(`$Func$`, `$List$`, `$InitVal$`)` = Res: This function is the same as `reduce(`$Func$`, [`$InitVal$`|`$List$`])`.

3.8 Other Built-ins in the `basic` Module

- `acyclic_term(`$Term$`)`: This predicate is true if $Term$ is acyclic, meaning that $Term$ does not contain itself.

- `and_to_list(`$Conj$`)` = $List$: This function converts $Conj$ in the form `(`a_1`, ..., `a_n`)` into a list in the form `[`a_1`,...,`a_n`]`.

- `compare_terms(`$Term_1$`, `$Term_2$`) = `Res: This function compares $Term_1$ and $Term_2$. If $Term_1 < Term_2$, then this function returns -1. If $Term_1 == Term_2$, then this function returns 0. Otherwise, $Term_1 > Term_2$, and this function returns 1.

- `different_terms(`$Term_1$`, `$Term_2$`)`: This constraint ensures that $Term_1$ and $Term_2$ are different. This constraint is suspended when the arguments are not sufficiently instantiated.

- `get_global_map() = `Map: This function returns the global map.

- `get_global_map(`ID`) = `Map: This function returns the global map with the identifier ID, which must be a ground term.

- `get_heap_map() = `Map: This function returns the default heap map.

- `get_heap_map(`ID`) = `Map: This function returns the heap map with the identifier ID, which must be a ground term.

- `get_table_map() = `Map: This function returns the default table map. The table map is stored in the table area and both keys and values are hash-consed (i.e., common sub-terms are shared).

- `get_table_map(`ID`) = `Map: This function returns the table map with the identifier ID, which must be a ground term.

- `ground(`$Term$`)`: This predicate is true if $Term$ is ground. A *ground* term does not contain any variables.

- `list_to_and(`$List$`) = `$Conj$: This function converts $List$ in the form `[`a_1`,...,`a_n`]` into a term in the form `(`a_1`, ..., `a_n`)`.

- `number_vars(`$Term$`)`: This predicate numbers the variables in $Term$ by using the integers starting from 0. Different variables receive different numberings, and the occurrences of the same variable all receive the same numbering.

- `number_vars(`$Term$`, `N_0`) = `N_1: This function numbers the variables in $Term$ by using the integers starting from N_0. N_1 is the next integer that is available after $Term$ is numbered.

- `parse_radix_string(`$String$`, `$Base$`) = `Int: This function parses $String$ as an integer in base $Base$ and returns the decimal integer, where $Base$ must be greater than 1 and less than 37. For example, `parse_radix_string("101", 2)` returns 5, which is the same as `parse_term("0b101")`.

- `parse_term(`$String$`, `$Term$`, `$Vars$`)`: This predicate uses the Picat parser to extract a term $Term$ from $String$. $Vars$ is a map that contains pairs each of which has the form $Name$=Var.

- `parse_term(`$String$`) = `$Term$: This function converts $String$ to a term.

- `second(`$Compound$`) = `$Term$: This function returns the second argument of the compound term $Compound$.

- `subsumes(`$Term_1$`, `$Term_2$`)`: This predicate is true if $Term_1$ subsumes $Term_2$.

- `variant(`$Term_1$`, `$Term_2$`)`: This predicate is true if $Term_2$ is a variant of $Term_1$. $Term_1$ and $Term_2$ cannot share variables.
- `vars(`$Term$`) = `$Vars$: This function returns a list of variables that occur in $Term$.

Chapter 4

Predicates and Functions

In Picat, predicates and functions are defined with rules. Each rule ends with a dot (`.`) followed by a whitespace character or the end of the file.

Picat has two types of pattern-matching rules: the *non-backtrackable* rule

$Head, Cond$ `=>` $Body.$

and the *backtrackable* rule

$Head, Cond$ `?=>` $Body.$

Picat also supports Prolog-style Horn clauses and Definite Clause Grammar (DCG) rules for predicate definitions. A Horn clause takes the form:

$Head$ `:-` $Body.$

which can be written as

$Head.$

if $Body$ is `true`. A DCG rule takes the form:

$Head$ `-->` $Body.$

A predicate definition that consists of Horn clauses can be preceded by an *index declaration*, as described in Section 1.2. Picat converts Horn clauses and DCG rules into pattern-matching rules.

4.1 Predicates

A *predicate* defines a relation, and can have zero, one, or multiple answers. Within a predicate, the $Head$ is a *pattern* in the form $p(t_1, \ldots, t_n)$, where p is called the predicate *name*, and n is called the *arity*. When $n = 0$, the parentheses can be omitted. The condition $Cond$, which is an optional goal, specifies a condition under which the rule is applicable. $Cond$ cannot succeed more than once. The compiler converts $Cond$ to `once` $Cond$ if it would otherwise be possible for $Cond$ to succeed more than once.

For a call C, if C matches the pattern $p(t_1, \ldots, t_n)$ and $Cond$ is true, then the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites C into $Body$. If the used rule is non-backtrackable, then the rewriting is a commitment, and the program can never backtrack to C. However, if the used rule is backtrackable, then the program will backtrack to C once $Body$ fails, meaning that $Body$ will be rewritten back to C, and the next applicable rule will be tried on C.

A predicate is said to be *deterministic* if it is defined with non-backtrackable rules only, *nondeterministic* if at least one of its rules is backtrackable, and *globally deterministic* if it is deterministic and all of the predicates in the bodies of the predicate's rules are also globally deterministic. A deterministic predicate that is not globally deterministic can still have more than one answer.

Example

```
append(Xs, Ys, Zs) ?=> Xs = [], Ys = Zs.
append(Xs, Ys, Zs) =>
    Xs = [X|XsR],
    Zs = [X|ZsR],
    append(XsR, Ys, ZsR).

min_max([H], Min, Max) => Min = H, Max = H.
min_max([H|T], Min, Max) =>
    min_max(T, MinT, MaxT),
    Min = min(MinT, H),
    Max = max(MaxT, H).
```

The predicate `append(Xs, Ys, Zs)` is true if the concatenation of `Xs` and `Ys` is `Zs`. It defines a relation among the three arguments, and does not assume directionality of any of the arguments. For example, this predicate can be used to concatenate two lists, as in the call

```
append([a,b], [c,d], L)
```

This predicate can also be used to split a list nondeterministically into two sublists, as in the call `append(L1, L2, [a,b,c,d])`; this predicate can even be called with three free variables, as in the call `append(L1, L2, L3)`.

The predicate `min_max(L, Min, Max)` returns two values through its arguments. It binds `Min` to the minimum of list `L`, and binds `Max` to the maximum of list `L`. This predicate does not backtrack. Note that a call fails if the first argument is not a list. Also note that this predicate consumes linear space. A tail-recursive version of this predicate that consumes constant space will be given below.

4.2 Functions

A *function* is a special kind of predicate that always succeeds with *one* answer. Within a function, the $Head$ is an equation $p(t_1,\ldots,t_n)$=X, where p is called the function *name*, and X is an *expression* that gives the return value. Functions are defined with non-backtrackable rules only.

For a function call C, if C matches the pattern $p(t_1,\ldots,t_n)$ and $Cond$ is true, then the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites the equation C=X' into $(Body,\ X'=X)$, where X' is a newly introduced variable that holds the return value of C.

Picat allows inclusion of *function facts* in the form $p(t_1,\ \ldots,\ t_n)=Exp$ in function definitions. The function fact $p(t_1,\ \ldots,\ t_n)=Exp$ is shorthand for the rule:

$p(t_1,\ \ldots,\ t_n)=X\ \Rightarrow\ X=Exp.$

where X is a new variable.

Although all functions can be defined as predicates, it is preferable to define them as functions for two reasons. Firstly, functions often lead to more compact expressions than predicates, because arguments of function calls can be other function calls. Secondly, functions are easier to debug than predicates, because functions never fail and never return more than one answer.

Example

```
qequation(A, B, C) = (R1, R2),
    D = B*B-4*A*C,
    D >= 0
=>
    NTwoC = -2*C,
    R1 = NTwoC/(B+sqrt(D)),
    R2 = NTwoC/(B-sqrt(D)).

rev([]) = [].
rev([X|Xs]) = rev(Xs)++[X].
```

The function `qequation(A, B, C)` returns the pair of roots of $\texttt{A*X}^2\texttt{+B*X+C = 0}$. If the discriminant `B*B-4*A*C` is negative, then an exception will be thrown.

The function `rev(L)` returns the reversed list of `L`. Note that the function `rev(L)` takes quadratic time and space in the length of `L`. A tail-recursive version that consumes linear time and space will be given below.

4.3 Patterns and Pattern-Matching

The pattern $p(t_1, \ldots, t_n)$ in the head of a rule takes the same form as a structure. Function calls are not allowed in patterns. Also, patterns cannot contain index notations, dot notations, ranges, array comprehensions, or list comprehensions. Pattern matching is used to decide whether a rule is applicable to a call. For a pattern P and a term T, term T matches pattern P if P is identical to T, or if P can be made identical to T by instantiating P's variables. Note that variables in the term do not get instantiated after the pattern matching. If term T is more general than pattern P, then the pattern matching can never succeed.

Unlike calls in many committed-choice languages, calls in Picat are never suspended if they are more general than the head patterns of the rules. A predicate call fails if it does not match the head pattern of any of the rules in the predicate. A function call throws an exception if it does not match the head pattern of any of the rules in the function. For example, for the function call `rev(L)`, where `L` is a variable, Picat will throw the following exception:

```
unresolved_function_call(rev(L)).
```

A pattern can contain *as-patterns* in the form $V\texttt{@}Pattern$, where V is a new variable in the rule, and $Pattern$ is a non-variable term. The as-pattern $V\texttt{@}Pattern$ is the same as $Pattern$ in pattern matching, but after pattern matching succeeds, V is made to reference the term that matched $Pattern$. As-patterns can avoid re-constructing existing terms.

Example

```
merge([], Ys) = Ys.
merge(Xs, []) = Xs.
merge([X|Xs], Ys@[Y|_]) = [X|Zs], X < Y =>
    Zs = merge(Xs, Ys).
merge(Xs, [Y|Ys]) = [Y|merge(Xs,Ys)].
```

In the third rule, the as-pattern `Ys@[Y|_]` binds two variables: `Ys` references the second argument, and `Y` references the car of the argument. The rule can be rewritten as follows without using any as-pattern:

```
merge([X|Xs], [Y|Ys]) = [X|Zs], X < Y =>
    Zs = merge(Xs, [Y|Ys]).
```

Nevertheless, this version is less efficient, because the cons `[Y|Ys]` needs to be re-constructed.

4.4 Goals

In a rule, both the condition and the body are *goals*. Queries that the users give to the interpreter are also goals. A goal can take one of the following forms:

- `true`: This goal is always true.
- `fail`: This goal is always false. When `fail` occurs in a condition, the condition is false, and the rule is never applicable. When `fail` occurs in a body, it causes execution to backtrack.
- `false`: This goal is the same as `fail`.
- $p(t_1, \ldots, t_n)$: This goal is a predicate call. The arguments $t_1, \ldots, t_n$ are evaluated in the given order, and the resulting call is resolved using the rules in the predicate p/n. If the call succeeds, then variables in the call may get instantiated. Many built-in predicates are written in infix notation. For example, `X = Y` is the same as `'='(X, Y)`.
- P`,` Q: This goal is a conjunction of goal P and goal Q. It is resolved by first resolving P, and then resolving Q. The goal is true if both P and Q are true. Note that the order is important: (P, Q) is in general not the same as (Q, P).
- P `&&` Q: This is the same as `(`P`,` Q`)`.
- P`;` Q: This goal is a disjunction of goal P and goal Q. It is resolved by first resolving P. If P is true, then the disjunction is true. If P is false, then Q is resolved. The disjunction is true if Q is true. The disjunction is false if both P and Q are false. Note that a disjunction can succeed more than once. Note also that the order is important: (P; Q) is generally not the same as (Q; P).
- P `||` Q: This is the same as `(`P`;` Q`)`.
- `not` P: This goal is the negation of P. It is false if P is true, and true if P is false. Note a negation goal can never succeed more than once. Also note that no variables can get instantiated, no matter whether the goal is true or false.
- `\+` P: This is the same as `not` P.
- `once` P: This goal is the same as P, but can never succeed more than once.
- `repeat`: This predicate is defined as follows:

  ```
  repeat ?=> true.
  repeat => repeat.
  ```

  The `repeat` predicate is often used to describe failure-driven loops. For example, the query

  ```
  repeat, writeln(a), fail
  ```

repeatedly outputs `'a'` until `ctrl-c` is typed.

- `if`: An if statement takes the form

  `if` ($Cond_1$)
  $Goal_1$
  `elseif` ($Cond_2$)
  $Goal_2$
  ⋮
  `elseif` ($Cond_n$)
  $Goal_n$
  `else`
  $Goal_{else}$
  `end`

  where the `elseif` and `else` clauses are optional. If the `else` clause is missing, then the else goal is assumed to be `true`. For the if statement, Picat finds the first condition $Cond_i$ that is true. If such a condition is found, then the truth value of the if statement is the same as $Goal_i$. If none of the conditions is true, then the truth value of the if statement is the same as $Goal_{else}$. Note that no condition can succeed more than once.[1]

- `throw` $Exception$: This predicate throws the term $Exception$. This predicate will be detailed in Chapter 6 on Exceptions.

- `!`: This special predicate, called a *cut*, is provided for controlling backtracking. A cut in the body of a rule has the effect of removing the choice points, or alternative rules, of the goals to the left of the cut.

- Loops: Picat has three types of loop statements: foreach, while, and do-while. A loop statement is true if and only if every iteration of the loop is true. The details of loops are given in Chapter 5.

4.5 Tail Recursion

A rule is said to be *tail-recursive* if the last call in the body calls the predicate defined by the rule. The *last-call optimization* enables last calls to reuse the stack frame of the head predicate if the frame is not protected by any choice points. This optimization is especially effective for tail recursion, because it converts recursion into iteration. Tail recursion runs faster and consumes less memory than non-tail recursion.

The trick to convert a predicate (or a function) into tail recursion is to define a helper that uses an *accumulator* parameter to accumulate the result. When the base case is reached, the accumulator is returned. At each iteration, the accumulator is updated. Initially, the original predicate (or function) calls the helper with an initial value for the accumulator parameter.

Example

```
min_max([H|T], Min, Max) =>
    min_max_helper([H|T], H, Min, H, Max).
```

[1] If $Cond_i$ is not enclosed in parentheses, then it must be followed by the keyword `then`.

```
min_max_helper([], CMin, Min, CMax, Max) =>
    Min = CMin, Max = CMax.
min_max_helper([H|T], CMin, Min, CMax, Max) =>
    min_max_helper(T, min(CMin, H), Min, max(CMax, H), Max).

rev([]) = [].
rev([X|Xs]) = rev_helper(Xs, [X]).

rev_helper([], R) = R.
rev_helper([X|Xs], R) = rev_helper(Xs, [X|R]).
```

In the helper predicate `min_max_helper(L, CMin, Min, CMax, Max)`, `CMin` and `CMax` are accumulators: `CMin` is the current minimum value, and `CMax` is the current maximum value. When `L` is empty, the accumulators are returned by the unification calls `Min = CMin` and `Max = CMax`. When `L` is a cons `[H|T]`, the accumulators are updated: `CMin` changes to `min(CMin, H)`, and `CMax` changes to `max(CMax, H)`. The helper function `rev_helper(L, R)` follows the same idea: it uses an accumulator list to hold, in reverse order, the elements that have been scanned. When `L` is empty, the accumulator is returned. When `L` is the cons `[X|Xs]`, the accumulator `R` changes to `[X|R]`.

Chapter 5

Assignments and Loops

This chapter discusses variable assignments, loop constructs, and list and array comprehensions in Picat. It describes the *scope* of an assigned variable, indicating where the variable is defined, and where it is not defined. Finally, it shows how assignments, loops, and list comprehensions are related, and how they are compiled.

5.1 Assignments

Picat variables are *single-assignment*, meaning that once a variable is bound to a value, the variable cannot be bound again. In order to simulate imperative language variables, Picat provides the assignment operator `:=`. An assignment takes the form LHS:=RHS, where LHS is either a variable or an access of a compound value in the form X`[...]`. When LHS is an access in the form $X[I]$, the component of X indexed I is updated. This update is undone if execution backtracks over this assignment.

Example

```
test => X = 0, X := X + 1,  X := X + 2, write(X).
```

The compiler needs to give special consideration to the *scope* of a variable. The scope of a variable is the region of the program in which its name refers to that variable.

Consider the `test` example. This example binds `X` to 0. Then, the example tries to bind `X` to `X + 1`. However, `X` is still visible at the assignment, and the preceding `X = 0` has bound it. Since `X` cannot be bound again, the compiler must perform extra operations in order to manage assignments that use the `:=` operator.

In order to handle assignments, Picat creates new variables at compile time. In the `test` example, at compile time, Picat creates a new variable, say `X1`, to hold the value of `X` after the assignment `X := X + 1`. Picat replaces `X` by `X1` on the LHS of the assignment. All occurrences of `X` after the assignment are replaced by `X1`. When encountering `X1 := X1 + 2`, Picat creates another new variable, say `X2`, to hold the value of `X1` after the assignment, and replaces the remaining occurrences of `X1` by `X2`. When `write(X2)` is executed, the value held in `X2`, which is 3, is printed. This means that the compiler rewrites the above example as follows:

```
test => X = 0, X1 = X + 1, X2 = X1 + 2, write(X2).
```

5.1.1 If-Else

This leads to the question: what does the compiler do if the code branches? Consider the following code skeleton.

Example

```
if_ex(Z) =>
    X = 1, Y = 2,
    if (Z > 0)
       X := X * Z
    else
       Y := Y + Z
    end,
    println([X,Y]).
```

The `if_ex` example performs exactly one assignment. At compilation time, the compiler does not know whether or not `Z>0` evaluates to `true`. Therefore, the compiler does not know whether to introduce a new variable for `X` or for `Y`.

Therefore, when an if-else statement contains an assignment, the compiler rewrites the if-else statement as a predicate. For example, the compiler rewrites the above example as follows:

```
if_ex(Z) =>
   X = 1, Y = 2,
   p(X, Xout, Y, Yout, Z),
   println([Xout,Yout]).

p(Xin, Xout, Yin, Yout, Z), Z > 0 =>
   Xout = Xin * Z,
   Yout = Yin.
p(Xin, Xout, Yin, Yout, Z) =>
   Xout = Xin,
   Yout = Yin + Z.
```

One rule is generated for each branch of the if-else statement. For each variable `V` that occurs on the LHS of an assignment statement that is inside of the if-else statement, predicate `p` is passed two arguments, `Vin` and `Vout`. In the above example, `X` and `Y` each occur on the LHS of an assignment statement. Therefore, predicate `p` is passed the parameters `Xin`, `Xout`, `Yin`, and `Yout`.

5.2 Types of Loops

Picat has three types of loop statements for programming repetitions: `foreach`, `while`, and `do-while`.

5.2.1 Foreach Loops

A `foreach` loop has the form:

`foreach (`E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`)`
   Goal
`end`

Each E_i is an *iterating pattern*. Each D_i is an expression that gives a *compound value*. Each $Cond_i$ is an optional *condition* on iterators E_1 through E_i.

Foreach loops can be used to iterate through compound values, as in the following examples.

Example

```
loop_ex1 =>
    L = [17, 3, 41, 25, 8, 1, 6, 40],
    foreach (E in L)
        println(E)
    end.

loop_ex2(Map) =>
    foreach ((Key = Value) in Map)
        writef("%w = %w\n", Key, Value)
    end.
```

The `loop_ex1` example iterates through a list. The `loop_ex2` example iterates through a map, where `Key = Value` is the iterating pattern.

The `loop_ex1` example can also be written, using a *failure-driven loop*, as follows.

Example

```
loop_ex1 =>
    L = [17, 3, 41, 25, 8, 1, 6, 40],
    (    member(E, L),
         println(E),
         fail
    ;
         true
    ).
```

Recall that the range $Start..Step..End$ stands for a list of numbers. Ranges can be used as compound values in iterators.

Example

```
loop_ex3 =>
    foreach (E in 1 .. 2 .. 9)
        println(E)
    end.
```

Also recall that the function `zip(`$List_1$`, `$List_2$`, ..., `$List_n$`)` returns a list of tuples. This function can be used to simultaneously iterate over multiple lists.

Example:

```
loop_ex_parallel =>
    foreach (Pair in zip(1..2, [a,b]))
        println(Pair)
    end.
```

5.2.2 Foreach Loops with Multiple Iterators

Each of the previous examples uses a single iterator. Foreach loops can also contain multiple iterators.

Example:

```
loop_ex4 =>
    L = [2, 3, 5, 10],
    foreach (I in L, J in 1 .. 10, J mod I != 0)
        printf("%d is not a multiple of %d%n", J, I)
    end.
```

If a foreach loop has multiple iterators, then it is compiled into a series of nested foreach loops in which each nested loop has a single iterator. In other words, a foreach loop with multiple iterators executes its goal once for every possible combination of values in the iterators.

The foreach loop in `loop_ex4` is the same as the nested loop:

```
loop_ex5 =>
    L = [2, 3, 5, 10],
    foreach (I in L)
        foreach (J in 1..10)
            if (J mod I != 0)
               printf("%d is not a multiple of %d%n", J, I)
            end
        end
    end.
```

5.2.3 Foreach Loops with the `break` Condition

When the condition `break(`$Cond$`)` occurs to the right of an iterator in a `foreach` loop, it terminates the loop if $Cond$ is true. Note that the loop body is not executed when $Cond$ is true.

The foreach loop in `loop_ex6` prints out the elements of the array `A`, stopping when it encounters a 0.

```
loop_ex6 =>
    A = {2, 3, 5, 0, 10, 4},
    foreach (I in 1..len(A),  break(A[I] == 0))
        println(A[I])
    end.
```

5.2.4 While Loops

A `while` loop has the form:

```
while (Cond)
   Goal
end
```

As long as $Cond$ succeeds, the loop will repeatedly execute $Goal$.

Example:

```
loop_ex7 =>
    I = 1,
    while (I <= 9)
        println(I),
```

```
        I := I + 2
    end.

loop_ex8 =>
    J = 6,
    while (J <= 5)
        println(J),
        J := J + 1
    end.

loop_ex9 =>
    E = read_int(),
    while (E mod 2 == 0; E mod 5 == 0)
        println(E),
        E := read_int()
    end.

loop_ex10 =>
    E = read_int(),
    while (E mod 2 == 0, E mod 5 == 0)
        println(E),
        E := read_int()
    end.
```

The while loop in `loop_ex7` prints all of the odd numbers between 1 and 9. It is similar to the foreach loop

```
foreach (I in 1 .. 2 .. 9)
    println(I)
end.
```

The while loop in `loop_ex8` never executes its goal. `J` begins at 6, so the condition `J <= 5` is never true, meaning that the body of the loop does not execute.

The while loop in `loop_ex9` demonstrates a compound condition. The loop executes as long as the value that is read into `E` is either a multiple of 2 or a multiple of 5.

The while loop in `loop_ex10` also demonstrates a compound condition. Unlike in `loop_ex9`, in which either condition must be true, in `loop_ex10`, both conditions must be true. The loop executes as long as the value that is read into `E` is both a multiple of 2 and a multiple of 5.

5.2.5 Do-while Loops

A `do-while` loop has the form:

`do`
Goal
`while (`*Cond*`)`

A do-while loop is similar to a while loop, except that a do-while loop executes *Goal* one time before testing *Cond*. The following example demonstrates the similarities and differences between do-while loops and while loops.

Example

```
loop_ex11 =>
    J = 6,
    do
        println(J),
        J := J + 1
    while (J <= 5).
```

Unlike `loop_ex8`, `loop_ex11` executes its body once. Although `J` begins at 6, the do-while loop prints `J`, and increments `J` before evaluating the condition `J <= 5`.

5.3 List and Array Comprehensions

A *list comprehension* is a special functional notation for creating lists. List comprehensions have a similar format to foreach loops.

$$\texttt{[}T\ \texttt{:}\ \ E_1\ \texttt{in}\ D_1\texttt{,}\ Cond_1\texttt{,}\ \ldots\texttt{,}\ E_n\ \texttt{in}\ D_n\texttt{,}\ Cond_n\texttt{]}$$

T is an expression. Each E_i is an *iterating pattern*. Each D_i is an expression that gives a *compound value*. Each $Cond_i$ is an optional *condition* on iterators E_1 through E_i.

An array comprehension takes the following form:

$$\texttt{\{}T\ \texttt{:}\ \ E_1\ \texttt{in}\ D_1\texttt{,}\ Cond_1\texttt{,}\ \ldots\texttt{,}\ E_n\ \texttt{in}\ D_n\texttt{,}\ Cond_n\texttt{\}}$$

It is the same as:

$$\texttt{to_array([}T\ \texttt{:}\ \ E_1\ \texttt{in}\ D_1\texttt{,}\ Cond_1\texttt{,}\ \ldots\texttt{,}\ E_n\ \texttt{in}\ D_n\texttt{,}\ Cond_n\texttt{])}$$

Example

```
Picat> L = [(A, I) : A in [a, b], I in 1 .. 2].
L = [(a , 1),(a , 2),(b , 1),(b , 2)]

Picat> L = {(A, I) : A in [a, b], I in 1 .. 2}.
L = {(a , 1),(a , 2),(b , 1),(b , 2)}
```

5.4 Compilation of Loops

Variables that occur in a loop, but do not occur before the loop in the outer scope, are local to each iteration of the loop. For example, in the rule

```
p(A) =>
    foreach (I in 1 .. A.length)
        E = A[I],
        println(E)
    end.
```

the variables `I` and `E` are local, and each iteration of the loop has its own values for these variables.

Consider the example:

Example

```
while_test(N) =>
    I = 1,
    while (I <= N)
        I := I + 1,
        println(I)
    end.
```

In this example, the while loop contains an assignment statement. As mentioned above, at compilation time, Picat creates new variables in order to handle assignments. One new variable is created for each assignment. However, when this example is compiled, the compiler does not know the number of times that the body of the while loop can be executed. This means that the compiler does not know how many times the assignment `I := I + 1` will occur, and the compiler is unable to create new variables for this assignment. In order to solve this problem, the compiler compiles while loops into tail-recursive predicates.

In the `while_test` example, the while loop is compiled into:

```
while_test(N) =>
    I = 1,
    p(I, N).

p(I, N), I <= N =>
    I1 = I + 1,
    println(I1),
    p(I1, N).
p(_, _) => true.
```

Note that the first rule of the predicate `p(I, N)` has the same condition as the while loop. The second rule, which has no condition, terminates the while loop, because the second rule is only executed if `I > N`. The call `p(I1, N)` is the tail-recursive call, with `I1` storing the modified value.

Suppose that a while loop modifies a variable that is then used outside of the while loop. For each modified variable `V` that is used after the while loop, predicate `p` is passed two arguments, `Vin` and `Vout`. Then, a predicate that has the body `true` is not sufficient to terminate the compiled while loop.

The next example demonstrates a loop that has multiple accumulators, and that modifies values which are then used outside of the loop.

Example

```
min_max([H|T], Min, Max) =>
    LMin = H,
    LMax = H,
    foreach (E in T)
        LMin := min(LMin, E),
        LMax := max(LMax, E)
    end,
    Min = LMin,
    Max = LMax.
```

This loop finds the minimum and maximum values of a list. The loop is compiled to:

```
min_max([H|T], Min, Max) =>
    LMin = H,
    LMax = H,
    p(T, LMin, LMin1, LMax, LMax1),
    Min = LMin1,
    Max = LMax1.

p([], MinIn, MinOut, MaxIn, MaxOut) =>
    MinOut = MinIn,
    MaxOut = MaxIn.
p([E|T], MinIn, MinOut, MaxIn, MaxOut) =>
    Min1 = min(MinIn, E),
    Max1 = max(MaxIn, E),
    p(T, Min1, MinOut, Max1, MaxOut).
```

Notice that there are multiple accumulators: `MinIn` and `MaxIn`. Since the `min_max` predicate returns two values, the accumulators each have an *in* variable (`MinIn` and `MaxIn`) and an *out* variable (`MinOut` and `MaxOut`). If the first parameter of predicate `p` is an empty list, then `MinOut` is set to the value of `MinIn`, and `MaxOut` is set to the value of `MaxIn`.

Foreach and do-while loops are compiled in a similar manner to while loops.

Nested Loops

As mentioned above, variables that only occur within a loop are local to each iteration of the loop. In nested loops, variables that are local to the outer loop are global to the inner loop. In other words, if a variable occurs in the outer loop, then the variable is also visible in the inner loop. However, variables that are local to the inner loop are not visible to the outer loop.

For example, consider the nested loops:

```
nested =>
    foreach (I in 1 .. 10)
        printf("Numbers between %d and %d ", I, I * I),
        foreach (J in I .. I * I)
            printf("%d ", J)
        end,
        nl
    end.
```

Variable `I` is local to the outer foreach loop, and is global to the inner foreach loop. Therefore, iterator `J` is able to iterate from `I` to `I * I` in the inner foreach loop. Iterator `J` is local to the inner loop, and does not occur in the outer loop.

Since a foreach loop with `N` iterators is converted into `N` nested foreach loops, the order of the iterators matters.

5.4.1 List Comprehensions

List comprehensions are compiled into foreach loops.

Example

```
comp_ex =>
    L = [(A, X) : A in [a, b], X in 1 .. 2].
```

This list comprehension is compiled to:

```
comp_ex =>
    L = List,
    foreach (A in [a, b], X in 1 .. 2)
        List = [(A, X) | T],
        List := T
    end,
    List = [].
```

Example

```
make_list1 =>
    L = [Y : X in 1..5],
    write(L).

make_list2 =>
    Y = Y,
    L = [Y : X in 1..5],
    write(L).
```

Suppose that a user would like to create a list `[Y, Y, Y, Y, Y]`. The `make_list1` predicate incorrectly attempts to make this list; instead, it outputs a list of 5 different variables since `Y` is local. In order to make all five variables the same, `make_list2` makes variable `Y` global, by adding the line `Y = Y` to globalize `Y`.

Chapter 6

Exceptions

An *exception* is an event that occurs during the execution of a program. An exception requires special treatment. In Picat, an exception is just a term. A built-in exception is an atom or a structure. If it is a structure, the name denotes the *type* of the exception, and the arguments provide other information about the exception, such as the *source*, which is the goal or function that raised the exception.

6.1 Built-in Exceptions

Picat throws many types of exceptions. The following are some of the built-in exceptions:[1]

- `domain_error(`Val`,` $Source$`)`: $Source$ receives a value Val that is unexpected in the domain.
- `existence_error(`$Entity$`,` $Source$`)`: $Source$ tries to use $Entity$, such as a file, a function, or a solver, that does not exist.
- `interrupt`: The execution is interrupted by a signal or `ctrl-c`.
- `io_error(`ENo`,` $EMsg$`,` $Source$`)`: An I/O error with the number ENo and message $EMsg$ occurs in $Source$.
- `load_error(`$FName$`,` $Source$`)`: An error occurs while loading the bytecode file named $FName$. This error is caused by the malformed bytecode file.
- `out_of_bound(`$EIndex$`,` $Source$`)`: $Source$ tries to access an element of a compound value using the index $EIndex$, which is out of bounds. An index is out of bounds if it is less than or equal to zero, or if it is greater than the length of the compound value.
- `out_of_memory(`$Area$`)`: The system runs out of memory while expanding $Area$, which can be: `stack_heap`, `trail`, `program`, `table`, or `findall`.
- `syntax_error(`$String$`,` $Source$`)`: $String$ cannot be parsed into a value that is expected by $Source$. For example, `read_int()` throws this exception if it reads in a string `"a"` rather than an integer.
- `unresolved_function_call(`$FCall$`)`: No rule is applicable to the function call $FCall$.

[1] In the current implementation, exceptions thrown by B-Prolog are not compliant with the documentation.

- `zero_divisor(`$Source$`)`: $Source$ divides a number by zero.
- $Type$`_expected(`$EArg$`, `$Source$`)`: The argument $EArg$ in $Source$ does not have the expected type or value, where $Type$ can be `var`, `nonvar`, `dvar`, `atom`, `integer`, `real`, `number`, `list`, `map`, etc.

6.2 Throwing Exceptions

The built-in predicate `throw(`$Exception$`)` throws $Exception$. After an exception is thrown, the system searches for a handler for the exception. If none is found, then the system displays the exception and aborts the execution of the current query. It also prints the backtrace of the stack if it is in debug mode. For example, for the function call `open("abc.txt")`, the following message will be displayed if there is no file that is named `"abc.txt"`.

```
*** error(existence_error(source_sink, abc.txt), open)
```

6.3 Defining Exception Handlers

All exceptions, including those raised by built-ins and interruptions, can be caught by catchers. A catcher is a call in the form:

```
catch(Goal, Exception, Handler)
```

which is equivalent to `Goal`, except when an exception is raised during the execution of `Goal` that unifies with `Exception`. When such an exception is raised, all of the bindings that have been performed on variables in `Goal` will be undone, and `Handler` will be executed to handle the exception. Note that `Exception` is unified with a renamed copy of the exception before `Handler` is executed. Also note that only exceptions that are raised by a descendant call of `Goal` can be caught.

The call `call_cleanup(Call, Cleanup)` is equivalent to `call(Call)`, except that `Cleanup` is called when `Call` succeeds deterministically (i.e., with no remaining choice point), when `Call` fails, or when `Call` raises an exception.

Chapter 7

Tabling

The Picat system is a term-rewriting system. For a predicate call, Picat selects a matching rule and rewrites the call into the body of the rule. For a function call C, Picat rewrites the equation $C = X$ where X is a variable that holds the return value of C. Due to the existence of recursion in programs, the term-rewriting process may never terminate. Consider, for example, the following program:

```
reach(X, Y) ?=> edge(X, Y).
reach(X, Y) => reach(X, Z), edge(Z, Y).
```

where the predicate `edge` defines a relation, and the predicate `reach` defines the transitive closure of the relation. For a query such as `reach(a, X)`, the program never terminates due to the existence of left-recursion in the second rule. Even if the rule is converted to right-recursion, the query may still not terminate if the graph that is represented by the relation contains cycles.

Another issue with recursion is redundancy. Consider the following problem: *Starting in the top left corner of an $N \times N$ grid, one can either go rightward or downward. How many routes are there through the grid to the bottom right corner?* The following gives a program in Picat for the problem:

```
route(N, N, _Col) = 1.
route(N, _Row, N) = 1.
route(N, Row, Col) =
    route(N, Row+1, Col)+route(N, Row, Col+1).
```

The function call `route(20, 1, 1)` returns the number of routes through a 20×20 grid. The function call `route(N, 1, 1)` takes exponential time in `N`, because the same function calls are repeatedly spawned during the execution, and are repeatedly resolved each time that they are spawned.

7.1 Table Declarations

Tabling is a memoization technique that can prevent infinite loops and redundancy. The idea of tabling is to memoize the answers to subgoals and use the answers to resolve their variant descendants. In Picat, in order to have all of the calls and answers of a predicate or function tabled, users just need to add the keyword `table` before the first rule.

Example

```
table
```

```
reach(X, Y) ?=> edge(X, Y).
reach(X, Y) => reach(X, Z), edge(Z, Y).

table
route(N, N, _Col) = 1.
route(N, _Row, N) = 1.
route(N, Row, Col) = route(N, Row+1, Col)
                   + route(N, Row, Col+1).
```

With tabling, all queries to the `reach` predicate are guaranteed to terminate, and the function call `route(N, 1, 1)` takes only N^2 time.

For some problems, such as planning problems, it is infeasible to table all answers, because there may be an infinite number of answers. For some other problems, such as those that require the computation of aggregates, it is a waste to table non-contributing answers. Picat allows users to provide table modes to instruct the system about which answers to table. For a tabled predicate, users can give a *table mode declaration* in the form $(M_1, M_2, \ldots, M_n)$, where each M_i is one of the following:

- + : indicates input
- - : indicates output
- `max`: indicates that the corresponding variable should be maximized (only the first best answer is tabled).
- `min`: indicates that the corresponding variable should be minimized (only the first best answer is tabled).
- `mmax`: indicates that the corresponding variable should be maximized (all best answers are tabled).
- `mmin`: indicates that the corresponding variable should be minimized (all best answers are tabled).

The last mode M_n can be `nt`, which indicates that the argument is not tabled. Two types of data can be passed to a tabled predicate as an `nt` argument: (1) global data that are the same for all calls of the predicate, and (2) data that are functionally dependent on the input arguments.

An argument with the mode `max`, `min`, `mmax` or `mmin` is called an *objective* argument. Only one argument can be an objective to be optimized. As an objective argument can be a compound value, this limit is not essential, and users can still specify multiple objective variables to be optimized. When the table mode `max` or `min` is provided, Picat tables only one optimal answer for the same input arguments. When the table mode `mmax` or `mmin` is provided, Picat tables all optimal answers for the same input arguments. Input arguments are assumed to be ground. Output arguments, including objective arguments, are assumed to be variables.

Example

```
table(+, +, -, min)
sp(X, Y, Path, W) ?=>
    Path = [(X,Y)],
    edge(X, Y, W).
sp(X, Y, Path, W) =>
```

```
    Path = [(X,Z)|Path1],
    edge(X, Z, Wxz),
    sp(Z, Y, Path1, W1),
    W = Wxz+W1.
```

The predicate `edge(X, Y, W)` specifies a weighted directed graph, where `W` is the weight of the edge between node `X` and node `Y`. The predicate `sp(X, Y, Path, W)` states that `Path` is a path from `X` to `Y` with the minimum weight `W`. Note that whenever the predicate `sp/4` is called, the first two arguments must always be instantiated. For each pair, the system stores only one path with the minimum weight. When the mode `min` is replaced with `mmin`, the system stores all paths with the minimum weight.

The following program finds a shortest path among those with the minimum weight for each pair of nodes:

```
table (+, +, -, min)
sp(X, Y, Path, WL) ?=>
    Path = [(X,Y)],
    WL = (Wxy, 1),
    edge(X, Y, Wxy).
sp(X, Y, Path, WL) =>
    Path = [(X,Z)|Path1],
    edge(X, Z, Wxz),
    sp(Z, Y, Path1, WL1),
    WL1 = (Wzy, Len1),
    WL = (Wxz+Wzy, Len1+1).
```

For each pair of nodes, the pair of variables `(W, Len)` is minimized, where `W` is the weight, and `Len` is the length of a path. The built-in function `compare_terms(`T_1`, `T_2`)` is used to compare answers. Note that the order is important. If the term were `(Len, W)`, then the program would find a shortest path, breaking a tie by selecting one with the minimum weight.

The tabling system is useful for offering dynamic programming solutions for planning problems. The following shows a tabled program for general planning problems:

```
table (+, -, min)
plan(S, Plan, Len), final(S) => Plan = [], Len = 0.
plan(S, Plan, Len) =>
    action(Action, S, S1),
    plan(S1, Plan1, Len1),
    Plan = [Action|Plan1],
    Len = Len1+1.
```

The predicate `action(Action, S, S1)` selects an action and performs the action on state `S` to generate another state, `S1`. Again, by replacing `min` with `mmin`, the predicate `plan` can be employed to generate all optimal plans.

Example

The program shown in Figure 7.1 solves the Farmer's problem: *The farmer wants to get his goat, wolf, and cabbage to the other side of the river. His boat isn't very big, and it can only carry him and either his goat, his wolf, or his cabbage. If he leaves the goat alone with the cabbage, then the goat will gobble up the cabbage. If he leaves the wolf alone with the goat, then the wolf will gobble up the goat. When the farmer is present, the goat and cabbage are safe from being gobbled up by their predators.*

```
go =>
    S0 = [s,s,s,s],
    plan(S0, Plan, _),
    writeln(Plan).

table (+, -, min)
plan([n,n,n,n], Plan, Len) => Plan = [], Len = 0.
plan(S, Plan, Len) =>
    Plan = [Action|Plan1],
    action(S, S1, Action),
    plan(S1, Plan1, Len1),
    Len = Len1+1.

action([F,F,G,C], S1, Action) ?=>
    Action = farmer_wolf,
    opposite(F, F1),
    S1 = [F1,F1,G,C],
    not unsafe(S1).
action([F,W,F,C], S1, Action) ?=>
    Action = farmer_goat,
    opposite(F, F1),
    S1 = [F1,W,F1,C],
    not unsafe(S1).
action([F,W,G,F], S1, Action) ?=>
    Action = farmer_cabbage,
    opposite(F, F1),
    S1 = [F1,W,G,F1],
    not unsafe(S1).
action([F,W,G,C], S1, Action) ?=>
    Action = farmer_alone,
    opposite(F, F1),
    S1 = [F1,W,G,C],
    not unsafe(S1).

index (+, -) (-, +)
opposite(n, s).
opposite(s, n).

unsafe([F,W,G,_C]), W == G, F !== W => true.
unsafe([F,_W,G,C]), G == C, F !== G => true.
```

Figure 7.1: A program for the Farmer's problem.

7.2 The Tabling Mechanism

The Picat tabling system employs the so-called *linear tabling* mechanism, which computes fixpoints by iteratively evaluating looping subgoals. The system uses a data area, called the *table area*, to store tabled subgoals and their answers. The table area can be initialized with the following built-in predicate:

- `initialize_table`: This predicate initializes the table area.

This predicate clears the table area. It's the user's responsibility to ensure that no data in the table area are referenced by any part of the application.

Linear tabling relies on the following three primitive operations to access and update the table area.

Subgoal lookup and registration: This operation is used when a tabled subgoal is encountered during execution. It looks up the subgoal table to see if there is a variant of the subgoal. If not, it inserts the subgoal (termed a *pioneer* or *generator*) into the subgoal table. It also allocates an answer table for the subgoal and its variants. Initially, the answer table is empty. If the lookup finds that there already is a variant of the subgoal in the table, then the record that is stored in the table is used for the subgoal (called a *consumer*). Generators and consumers are handled differently. In linear tabling, a generator is resolved using rules, and a consumer is resolved using answers; a generator is iterated until the fixpoint is reached, and a consumer fails after it exhausts all of the existing answers.

Answer lookup and registration: This operation is executed when a rule succeeds in generating an answer for a tabled subgoal. If a variant of the answer already exists in the table, then it does nothing; otherwise, it inserts the answer into the answer table for the subgoal, or it tables the answer according to the mode declaration. Picat uses the lazy consumption strategy (also called the local strategy). After an answer is processed, the system backtracks to produce the next answer.

Answer return: When a consumer is encountered, an answer is returned immediately, if an answer exists. On backtracking, the next answer is returned. A generator starts consuming its answers after it has exhausted all of its rules. Under the lazy consumption strategy, a top-most looping generator does not return any answer until it is complete.

Chapter 8

The `planner` Module

The `planner` module provides several predicates for solving planning problems. Given an initial state, a final state, and a set of possible actions, a planning problem is to find a plan that transforms the initial state to the final state. In order to use the `planner` module to solve a planning problem, users have to provide the condition for the final states and the state transition diagram through the following global predicates:

- `final(`S`)`: This predicate succeeds if S is a final state.
- `final(`S`, `$Plan$`, `$Cost$`)`: A final state can be reached from S by the action sequence in $Plan$ with $Cost$. If this predicate is given, then there is no need to define `final/1`. If this predicate is not given, then the system assumes the following definition:

```
final(S, Plan, Cost) => Plan = [], Cost = 0, final(S).
```

- `action(`S`, `$NextS$`, `$Action$`, `$ActionCost$`)`: This predicate encodes the state transition diagram of the planning problem. The state S can be transformed into $NextS$ by performing $Action$. The cost of $Action$ is $ActionCost$. If the plan's length matters, then $ActionCost$ should be 1.

A state is normally a ground term. As all states are tabled during search, it is of paramount importance to find a good representation for states such that terms among states can be shared as much as possible.

In addition to the two required predicates given above, users can optionally provide the following global procedures to assist Picat in searching for plans:

- `heuristic(`S`) = `N: This function returns an estimation of the resource amount needed to transform S into a goal state. This estimation is said to be *admissible* if it never exceeds the real cost. All heuristic estimations must be admissible in order for the planner to find optimal plans. This function is used by the planner to check the heuristic estimation before each state expansion.
- `sequence(`P`, `$Action$`)`: This predicate binds $Action$ to a viable action form based on the current partial plan P. Note that the actions in list P are in reverse order, with the most recent action occurring first in the list, and the first action occurring last in the list. The planner calls `sequence/2` to find an action for expanding the current state before calling `action/4`. For example,

```
sequence([move(R,X,Y)|_], Action) ?=>
```

```
    Action = $move(R, Y, _).
sequence([move(R,_,_)|_], Action) ?=> Action = $jump(R).
sequence([move(R,_,_)|_], Action) => Action = $wait(R).
sequence(_, _) => true.
```

These sequence rules ban robots from moving in an interleaving fashion; a robot must continue to move until it takes the action `jump` or `wait` before another robot can start moving. The last rule `sequence(_, _) => true` is necessary; it permits any action to be taken if the partial plan is empty, or if the most recent action in the partial plan is not `move`.

8.1 Depth-Bounded Search

Depth-bounded search amounts to exploring the search space, taking into account the current available resource amount. A new state is only explored if the available resource amount is non-negative. When depth-bounded search is used, the function `current_resource()` can be used to retrieve the current resource amount. If the heuristic estimate of the cost to travel from the current state to the final state is greater than the available resource amount, then the current state fails.

- `plan(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to a plan that can transform state S to a final state that satisfies the condition given by `final/1` or `final/3`. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.
- `plan(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `plan/4` predicate, except that the plan's cost is not returned.
- `plan(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `plan/4` predicate, except that the limit is assumed to be 268435455.
- `plan(`S`, `$Plan$`)`: This predicate is the same as the `plan/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `best_plan(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate finds an optimal plan by using the following algorithm: It first calls `plan/4` to find a plan of 0 cost. If no plan is found, then it sets the cost limit to 1 or doubles the current limit. Once a plan is found, the algorithm uses binary search to find an optimal plan.
- `best_plan(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan/4` predicate, except that the plan's cost is not returned.
- `best_plan(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan/4` predicate, except that the limit is assumed to be 268435455.
- `best_plan(`S`, `$Plan$`)`: This predicate is the same as the `best_plan/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `best_plan_bb(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to an approximated optimal plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer. The branch-and-bound algorithm is used to find an optimal plan.

- `best_plan_bb(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_bb/4` predicate, except that the plan's cost is not returned.

- `best_plan_bb(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_bb/4` predicate, except that the limit is assumed to be 268435455.

- `best_plan_bb(`S`, `$Plan$`)`: This predicate is the same as the `best_plan_bb/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.

- `best_plan_bin(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to an approximated optimal plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer. The branch-and-bound algorithm is used with binary search to find an optimal plan.

- `best_plan_bin(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_bin/4` predicate, except that the plan's cost is not returned.

- `best_plan_bin(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_bin/4` predicate, except that the limit is assumed to be 268435455.

- `best_plan_bin(`S`, `$Plan$`)`: This predicate is the same as the `best_plan_bin/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.

- `current_resource()`=$Amount$: This function returns the current available resource amount of the current node. If the current execution path was not initiated by one of the calls that performs resource-bounded search, then 268435455 is returned. This function can be used to check the heuristics. If the heuristic estimate of the cost to travel from the current state to a final state is greater than the available resource amount, then the current state can be failed.

- `current_plan()`=$Plan$: This function returns the current plan that has transformed the initial state to the current state. If the current execution path was not initiated by one of the calls that performs resource-bounded search, then [] is returned.

- `current_resource_plan_cost(`$Amount$`, `$Plan$`, `$Cost$`)`: This predicate retrieves the attributes of the current node in the search tree, including the resource amount, the path to the node, and its cost.

- `is_tabled_state(`S`)`: This predicate succeeds if the state S has been explored before and has been tabled.

If depth-bounded search is used, then the table area must be cleared by calling the `initialize_table` predicate before each subsequent call to `plan` or `best_plan`.

8.2 Depth-Unbounded Search

In contrast to depth-bounded search, depth-unbounded search does not take into account the available resource amount. A new state can be explored even if no resource is available for the exploration. The advantage of depth-unbounded search is that failed states are never re-explored.

- `best_plan_nondet(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate is the same as `best_plan(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`, except that it allows multiple best plans to be returned through backtracking.

- `best_plan_nondet(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_nondet/4` predicate, except that the plan's cost is not returned.

- `best_plan_nondet(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_nondet/4` predicate, except that the limit is assumed to be 268435455.

- `best_plan_nondet(`S`, `$Plan$`)`: This predicate is the same as the `best_plan(`S`, `$Plan$`)` predicate, except that it is nondeterministic.

- `plan_unbounded(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to a plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.

- `plan_unbounded(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `plan_unbounded/4` predicate, except that the plan's cost is not returned.

- `plan_unbounded(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `plan_unbounded/4` predicate, except that the limit is assumed to be 268435455.

- `plan_unbounded(`S`, `$Plan$`)`: This predicate is the same as the above predicate, except that the cost is not returned.

- `best_plan_unbounded(`S`, `$Limit$`, `$Plan$`, `$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to an optimal plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.

- `best_plan_unbounded(`S`, `$Limit$`, `$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_unbounded/4` predicate, except that the plan's cost is not returned.

- `best_plan_unbounded(`S`, `$Plan$`, `$Cost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_unbounded/4` predicate, except that the limit is assumed to be 268435455.

- `best_plan_unbounded(`S`, `$Plan$`)`: This predicate is the same as the above predicate, except that the cost is not returned.

```
import planner.

go =>
    S0 = [s,s,s,s],
    best_plan(S0, Plan),
    writeln(Plan).

final([n,n,n,n]) => true.

action([F,F,G,C], S1, Action, ActionCost) ?=>
    Action = farmer_wolf,
    ActionCost = 1,
    opposite(F, F1),
    S1 = [F1,F1,G,C],
    not unsafe(S1).
action([F,W,F,C], S1, Action, ActionCost) ?=>
    Action = farmer_goat,
    ActionCost = 1,
    opposite(F, F1),
    S1 = [F1,W,F1,C],
    not unsafe(S1).
action([F,W,G,F], S1, Action, ActionCost) ?=>
    Action = farmer_cabbage,
    ActionCost = 1,
    opposite(F, F1),
    S1 = [F1,W,G,F1],
    not unsafe(S1).
action([F,W,G,C], S1, Action, ActionCost) =>
    Action = farmer_alone,
    ActionCost = 1,
    opposite(F, F1),
    S1 = [F1,W,G,C],
    not unsafe(S1).

index (+, -) (-, +)
opposite(n, s).
opposite(s, n).

unsafe([F,W,G,_C]), W == G, F !== W => true.
unsafe([F,_W,G,C]), G == C, F !== G => true.
```

Figure 8.1: A program for the Farmer's problem using `planner`.

8.3 Examples

The program shown in Figure 8.1 solves the Farmer's problem by using the `planner` module.

Figure 8.2 gives a program for the 15-puzzle problem. A state is represented as a list of sixteen locations, each of which takes the form `(`R_i`,` C_i`)`, where R_i is a row number and C_i is a column number. The first element in the list gives the position of the empty square, and the remaining elements in the list give the positions of the numbered tiles from 1 to 15. The function `heuristic(Tiles)` returns the Manhattan distance between the current state and the final state.

```
import planner.

main =>
    InitS = [(1,2),(2,2),(4,4),(1,3),
             (1,1),(3,2),(1,4),(2,4),
             (4,2),(3,1),(3,3),(2,3),
             (2,1),(4,1),(4,3),(3,4)],
    best_plan(InitS, Plan),
    foreach (Action in Plan)
       println(Action)
    end.

final(State) => State = [(1,1),(1,2),(1,3),(1,4),
                         (2,1),(2,2),(2,3),(2,4),
                         (3,1),(3,2),(3,3),(3,4),
                         (4,1),(4,2),(4,3),(4,4)].

action([P0@(R0,C0)|Tiles], NextS, Action, Cost) =>
    Cost = 1,
    (R1 = R0-1, R1 >= 1, C1 = C0, Action = up;
     R1 = R0+1, R1 =< 4, C1 = C0, Action = down;
     R1 = R0, C1 = C0-1, C1 >= 1, Action = left;
     R1 = R0, C1 = C0+1, C1 =< 4, Action = right),
    P1 = (R1, C1),
    slide(P0, P1, Tiles, NTiles),
    NextS = [P1|NTiles].

% slide the tile at P1 to the empty square at P0
slide(P0, P1, [P1|Tiles], NTiles) =>
    NTiles = [P0|Tiles].
slide(P0, P1, [Tile|Tiles], NTiles) =>
    NTiles = [Tile|NTilesR],
    slide(P0, P1, Tiles, NTilesR).

heuristic([_|Tiles]) = Dist =>
    final([_|FTiles]),
    Dist = sum([abs(R-FR)+abs(C-FC) :
                {(R,C),(FR,FC)} in zip(Tiles,FTiles)]).
```

Figure 8.2: A program for the 15-puzzle

Chapter 9

Modules

A module is a bundle of predicate and function definitions that are stored in one file. A module forms a namespace. Two definitions can have the same name if they reside in different modules. Because modules avoid name clashes, they are very useful for managing source files of large programs.

9.1 Module and Import Declarations

In Picat, source files must have the extension name `".pi"`. A module is a source file that begins with a module name declaration in the form:

`module` $Name$.

where $Name$ must be the same as the main file name. A file that does not begin with a module declaration is assumed to belong to the default *global* module. The following names are reserved for system modules and should not be used to name user modules: `basic`, `bp`, `cp`, `datetime`, `glb`, `io`, `math`, `mip`, `nn`, `ordset`, `os`, `planner`, `sat`, `smt`, `sys`, and `util`.

In order to use symbols that are defined in another module, users must explicitly import them with an import declaration in the form:

`import` $Name_1$`, ...,` $Name_n$.

where each imported $Name_i$ is a module name. For each imported module, the compiler searches for it first in the importing source file's directory and then in the search path that is specified by the environment variable PICATPATH. If no module is found, the compiler gives an error message. Several modules are imported by default, including `basic`, `math`, `io`, and `sys`.

The import relation is not transitive. Suppose that there are three modules: A, B, and C. If A imports B and B imports C, then A still needs to import C in order to reference C's symbols.

The built-in command `cl("xxx")` compiles the file `xxx.pi` and loads the generated code into the interpreter. The built-in command `load("xxx")` loads the bytecode file `xxx.qi`. It compiles the source file `xxx.pi` only when necessary. The `load` command also imports the public symbols defined in the module to the interpreter. This allows users to use these symbols on the command line without explicitly importing the symbols. If the file `xxx.pi` imports modules, those module files will be compiled and loaded when necessary.

A program file can contain `include` directives of the form:

`include` $Name$.

where $Name$ is a string indicating a file name relative to the program file. The `include` directive causes the content of the file named $Name$ to be copied verbatim to where the directive occurs. The `include` directive allows users to split a large program into several source files. Note that the included source files cannot contain module or import declarations, and no definitions can span multiple files.

9.2 Binding Calls to Definitions

The Picat system has a global symbol table for atoms, a global symbol table for structure names, and a global symbol table for modules. For each module, Picat maintains a symbol table for the public predicate and function symbols defined in the module. Private symbols that are defined in a module are compiled away, and are never stored in the symbol table. While predicate and function symbols can be local to a module, atoms and structures are always global.

The Picat module system is static, meaning that the binding of normal (or non-higher-order) calls to their definitions takes place at compile time. For each call, the compiler first searches the default modules for a definition that has the same name as the call. If no definition is found, then the compiler searches for a definition in the enclosing module. If no definition is found, the compiler searches the imported modules in the order that they were imported. If no definition is found in any of these modules, then the compiler will issue a warning[1], assuming the symbol is defined in the global module.

It is possible for two imported modules to contain different definitions that have the same name. When multiple names match a call, the order of the imported items determines which definition is used. Picat allows users to use qualified names to explicitly select a definition. A module-qualified call is a call preceded by a module name and '.' without intervening whitespace.

Example

```
% qsort.pi
module qsort.

sort([]) = [].
sort([H|T]) =
  sort([E : E in T, E =< H]) ++ [H] ++ sort([E : E in T, E > H]).

% isort.pi
module isort.

sort([]) = [].
sort([H|T]) = insert(H, sort(T)).

private
insert(X, []) = [X].
insert(X, Ys@[Y|_]) = Zs, X =< Y => Zs = [X|Ys].
insert(X, [Y|Ys]) = [Y|insert(X,Ys)].
```

The module `qsort` defines a function named `sort` using quick sort, and the module `isort` defines a function of the same name using insertion sort. In the following session, both modules are used.

[1] A warning is issued instead of an error. This allows users to test incomplete programs with missing definitions.

```
Picat> load("qsort")
Picat> load("isort")
Picat> L = sort([2,1,3])
L = [1,2,3]
Picat> L = qsort.sort([2,1,3])
L = [1,2,3]
Picat> L = isort.sort([2,1,3])
L = [1,2,3]
```

As `sort` is also defined in the `basic` module, which is preloaded, that function is used for the command `L = sort([2,1,3])`.

Module names are just atoms. Consequently, it is possible to bind a variable to a module name. Nevertheless, in a module-qualified call $M.C$, the module name can never be a variable. Recall that the dot notation is also used to access attributes and to call predicates and functions. The notation $M.C$ is treated as a call or an attribute if M is not an atom.

Suppose that users want to define a function named `generic_sort(M, L)` that sorts list `L` using the `sort` function defined in module `M`. Users cannot just call `M.sort(L)`, since `M` is a variable. Users can, however, select a function based on the value held in `M` by using function facts as follows:

```
generic_sort(qsort, L) = qsort.sort(L).
generic_sort(isort, L) = isort.sort(L).
```

9.3 Binding Higher-order Calls

Because Picat forbids variable module qualifiers and terms in dot notations, it is impossible to create module-qualified higher-order terms. For a higher-order call, if the compiler knows the name of the higher-order term, as in `findall(X, ` $member(X, L)$`)`, then it searches for a definition for the name, just like it does for a normal call. However, if the name is unknown, as in `F(X, Y)`, then the compiler generates code to search for a definition. For a higher-order call, the runtime system searches modules in the following order:

1. The implicitly imported built-in modules `basic`, `math`, `io`, and `sys`.
2. The current list of loaded modules that is returned by the built-in `loaded_modules()`, excluding the pre-imported built-in modules.
3. The global module.

For a module-qualified higher-order call such as `m.F(X, Y)`, the runtime system first searches module `m`, followed by the implicitly imported built-in modules and the global module.

As private symbols are compiled away at compile time, higher-order terms can never reference private symbols. Due to the overhead of runtime search, the use of higher-order calls is discouraged.

9.4 Library Modules

Picat comes with a library of standard modules, described in separate chapters. The function `sys.loaded_modules()` returns a list of modules that are currently in the system.

Chapter 10

Higher-order Calls and Anonymous Definitions

A higher-order predicate or function takes a callable value as an argument. Picat supports callable values that name predicates or functions, optionally with arguments already supplied. Anonymous functions and anonymous predicates provide callable values without requiring named definitions.

10.1 Higher-order Call Conventions

The predicate `call(`S`,` Arg_1`, ...,` Arg_n`)` invokes a predicate, while the function `apply(`S`,` Arg_1`, ...,` Arg_n`)` returns the value of a function call. For a named callable, S is an atom or a structure. If S is an atom, it supplies the name of the predicate or function. If S is a structure, its name supplies the callable name and its arguments precede the additional arguments. For example:

```
F = $'*'(2),
Y = apply(F,5)                % Y = 10
```

The resulting function call is `'*'(2,5)`. The dollar symbol constructs a structure instead of evaluating a function call. A bare atom can also name a callable:

```
F = '+',
Y = apply(F,3,4)              % Y = 7
```

When S is a variable at compile time, both call conventions can be written as S`(`Arg_1`, ...,` Arg_n`)`. The context determines whether this notation denotes a predicate call or a function call:

```
P = $member(X),
P([1,2,3]),                   % predicate context: call(P,[1,2,3])
F = $'*'(2),
Y = F(5)                      % function context: apply(F,5)
```

An occurrence used as a goal invokes a predicate; an occurrence used as a value-producing expression invokes a function. The result variable of a function call is not an additional explicit argument.

Within a term constructor, a higher-order call whose name is a variable is also treated as a function call:

```
F = $'*'(2),
T = $p(F(5))                  % T = p(10)
```

Here, the value of `F(5)` becomes an argument of the constructed term. The higher-order call is not preserved as literal syntax in that term.

10.1.1 Arity and Supplying Arguments

The supplied arguments must select a predicate or function of the appropriate arity. In particular, `apply` does not return a partially evaluated function when too few arguments are supplied; an incorrect arity raises an exception. A structure such as `$'*'(2)` explicitly stores a supplied argument. Constructing that structure is different from invoking a function with too few arguments.

An anonymous definition has the explicit arity given by its parameter list. All of those arguments must be supplied when it is invoked. To return a function that takes further arguments, write a nested anonymous function explicitly, as illustrated below.

10.2 Anonymous Functions

An anonymous function, also called a lambda expression, has the form

```
([A1,A2,...,An] -> Res, B1,B2,...,Bm)
```

The outer parentheses are required. The bracketed list contains the parameter patterns, `Res` is the return expression, and the goals following the first comma form the body. The body is executed before the return expression is evaluated. When no body is needed, the function can be written as

```
([A1,A2,...,An] -> Res)
```

Omitting the body is equivalent to using the body `true`.

Evaluating a lambda expression constructs a callable value, called a *closure*; it does not execute the body or evaluate the return expression. No dollar symbol is needed to construct this value. For example:

```
F = ([X,Y] -> X+Y),
Z = F(3,4)                  % Z = 7
```

The invocation `F(3,4)` can also be written as `apply(F,3,4)`. A function with a body can compute its result using intermediate variables:

```
F = ([X,Y] -> Z, T = X*X, Z = T+Y),
R = F(3,4)                  % R = 13
```

The body has the usual goal syntax, including conjunctions, disjunctions, conditionals, and loops. For example:

```
Sum = ([Xs] -> Total,
    Total = 0,
    foreach (X in Xs)
        Total := Total+X
    end),
S = Sum([1,2,3])            % S = 6
```

The function follows the execution conventions of an ordinary named function. It is not a generator of multiple return values.

10.2.1 Parameter Patterns

The parameters are terms, just like the arguments of a named rule head; they are not restricted to variables. Their matching follows the ordinary head pattern-matching conventions. For example:

```
First = ([[X|_]] -> X),
H = First([a,b,c])          % H = a
```

The outer bracketed list contains one parameter, whose pattern is a nonempty list. Matching does not instantiate the caller's argument merely to make a pattern applicable. The same restrictions on head patterns apply here; parameter patterns are not evaluated as function calls.

A parameter list must be a proper list of patterns. Its length determines the arity. An empty list denotes a function with no explicit arguments:

```
F = ([] -> 42),
Y = F()                     % Y = 42
```

10.3 Anonymous Predicates

An anonymous predicate has the form

```
([A1,A2,...,An] => B1,B2,...,Bm)
```

Its parameter list has the same interpretation as that of an anonymous function. The body is a goal, and there is no return expression. For example:

```
Print = ([X] => println(X)),
Print(hello)
```

The invocation `Print(hello)` can also be written as `call(Print,hello)`. An anonymous predicate with no explicit arguments uses `[]` as its parameter list and is invoked with `P()` or `call(P)`.

An anonymous predicate consists of one rule, written with `=>`; there is no anonymous `?=>` form. There is no alternative rule to try on backtracking. This does not require every goal in the body to be globally deterministic: a goal such as `member` can still supply multiple solutions within the body. For example:

```
Choose = ([X] => member(X,[a,b,c])),
L = findall(X,Choose(X))    % L = [a,b,c]
```

Use an anonymous predicate when success, failure, or bindings are the desired result; use an anonymous function when an expression must return a value.

10.4 Context Variables and Nested Definitions

An anonymous definition forms a *name scope*, just like a `foreach` loop. A variable that has occurred before the anonymous definition in the enclosing scope is *global* to the anonymous definition and refers to the existing variable. A variable whose first occurrence is inside the anonymous definition is a fresh *local* variable of that definition. This rule applies to variables in parameter patterns, the return expression, and the body. Here, "global" describes visibility across the scope boundary; it does not mean a module-global variable. A prior occurrence counts even if the variable is still unbound.

Local variables are fresh for each invocation and are not made visible to subsequent goals outside the definition. For example:

```
Offset = 10,
F = ([X] -> R, T = X+Offset, R = T*T),
A = F(1),                    % A = 121
B = F(2)                     % B = 144
```

The variable `Offset` is global to the anonymous definition, while `X`, `R`, and `T` first occur inside it and are local. The two invocations use fresh local variables. Nested anonymous definitions follow the same scope rule: a variable introduced in an outer definition can be global to an inner definition.

An anonymous definition can refer to variables from the enclosing context. A closure carries the values of these context variables at the time the closure is created. Subsequent changes to the context variables do not affect the values captured by that closure. This capture convention applies to both anonymous functions and anonymous predicates. For example:

```
Offset = 10,
AddOffset = ([X] -> X+Offset),
Offset := 20,
Y = AddOffset(5),            % Y = 15
NewAddOffset = ([X] -> X+Offset),
Z = NewAddOffset(5)          % Z = 25
```

The parameter `X` is supplied at invocation, while `Offset` comes from the context in which the closure is constructed. The captured variables are not additional arguments that the caller must supply. The first closure retains the captured value `10`, even after `Offset` is assigned `20`. The second closure is created after the assignment and therefore captures `20`. Invoking either closure does not recapture the current value of `Offset`.

An anonymous function can return another anonymous function:

```
MakeAdder = ([A] -> ([B] -> A+B)),
Add10 = MakeAdder(10),
Y = Add10(5)                  % Y = 15
```

The returned closure retains the context needed to use `A` after the outer invocation finishes. This example uses two explicit invocations rather than an implicit partial application.

10.5 Passing Anonymous Definitions as Arguments

A higher-order routine can invoke an anonymous definition using the same conventions as a named callable. For example:

```
transform(F,[]) = [].
transform(F,[X|Xs]) = [F(X)|transform(F,Xs)].

visit(P,[]) => true.
visit(P,[X|Xs]) =>
    P(X),
    visit(P,Xs).
```

These definitions can be used as follows:

```
Ys = transform(([X] -> X*X),[1,2,3]),
visit(([X] => println(X)),Ys)
```

The transformation returns `[1,4,9]`, and the visit prints those values. Neither routine needs to know whether its argument was constructed from a named callable or an anonymous definition.

10.6 Modules and Call Resolution

For an ordinary named higher-order call whose target is not known at compile time, the runtime system searches the implicitly imported built-in modules `basic`, `math`, `io`, and `sys`, followed by the other loaded modules in the order returned by `loaded_modules()`, and then the global module. If the compiler knows the callable name, it binds the call using the ordinary compile-time resolution conventions.

A variable call can be module-qualified, as in `m.F(X,Y)`. For this form, lookup starts in `m`, followed by the implicitly imported built-in modules and the global module. The module qualifier is an atom, not a variable. The restriction on constructing literal dot-notation terms still applies; qualification is call syntax rather than a module-qualified structure that can be constructed as data. Ordinary named higher-order terms cannot reference private symbols, which are compiled away at compile time.

A closure is associated with its defining code and captured context. Passing it to another module must preserve that association: calls written inside an anonymous body use the defining module's resolution context. The receiving module can invoke the closure without importing the defining module.

10.7 Implementation of Anonymous Definitions

The compiler introduces a unique dummy name for each anonymous definition and compiles the definition just like a named function or predicate. The dummy name belongs to the global module. The body, however, is compiled in the module in which the anonymous definition occurs. Thus, calls in the body are resolved using the defining module's definitions and imports, including its private definitions, just as in an ordinary named definition in that module. The return expression of an anonymous function is compiled in the same defining context.

The closure carries the generated name together with the values of the captured context variables. Conceptually, an anonymous function such as

```
Offset = 10,
F = ([X] -> X+Offset)
```

is replaced by a closure of the form

```
F = $lam999([Offset])
```

and the compiler introduces the function

```
lam999([Offset],X) = X+Offset => true.
```

Here, `lam999` stands for a unique compiler-generated name, and the first argument carries the captured environment. Invoking `F(5)` supplies the explicit argument after that environment argument. An anonymous predicate is translated in the same way, using a generated predicate rather than a function. For example, the anonymous predicate

```
([X] => println(X+Offset))
```

has a corresponding generated definition of the form

```
lam1000([Offset],X) => println(X+Offset).
```

Its closure is conceptually `$lam1000([Offset])`.

Placing the generated name in the global module allows a receiving module to invoke the closure without importing the defining module. Compiling the body in the defining module preserves the meaning of calls within the body. The generated name is unique, so separately compiled anonymous definitions do not share a dummy name. The names and environment lists shown here illustrate the translation; programs should not depend on their particular spelling or representation.

10.8 Compiled Bodies and Callable Terms

Anonymous bodies are compiled code. They can contain assignments, loops, and conditionals, even though these constructs cannot be preserved as literal callable terms for general `call` interpretation. An anonymous predicate is therefore useful for passing such a computation to another routine.

The printed representation and generated name of a closure are implementation details. Programs should construct, pass, and invoke the closure rather than inspect or reconstruct its generated representation. No particular runtime lookup strategy or performance guarantee is implied by the anonymous syntax.

Chapter 11

I/O

Picat has an `io` module for reading input from files and writing output to files. The `io` module is imported by default.

The `io` module contains functions and predicates that read from a file, write to a file, reposition the read/write pointer within a file, redirect input and output, and create temporary files and pipes.

The `io` module uses file descriptors to read input from files, and to write output to files. A *file descriptor* is a structure that encodes file descriptor data, including an index in a file descriptor table that stores information about opened files. The following example reads data from one file, and writes the data into another file.

Example

```
rw =>
     Reader = open("input_file.txt"),
     Writer = open("output_file.txt", write),
     L = read_line(Reader),
     while (L != end_of_file)
            println(Writer, L),
            flush(Writer),
            L := read_line(Reader)
     end,
     close(Reader),
     close(Writer).
```

11.1 Opening a File

There are two functions for opening a file. Both of them are used in the previous example.

- `open(`$Name$`) = `FD: The $Name$ parameter is a file name that is represented as a string. This function opens the file with a default `read` mode.
- `open(`$Name$`, `$Mode$`) = `FD: The $Mode$ parameter is one of the three atoms: `read`, `write`, or `append`. The `read` atom is used for reading from a file; if the file does not exist, or the program tries to write to the file, then the program will throw an error. The `write` atom is used for reading from a file and writing to a file; if the file already exists, then the file will be overwritten. The `append` atom is similar to the write atom; however, if the file already exists, then data will be appended at the end of the pre-existing file.

11.2 Reading from a File

The `io` module has at least one function for reading data into each primitive data type. It also has functions for reading characters, tokens, strings, and bytes. Recall that strings are stored as lists of single-character atoms.

The `read` functions in the `io` module take a file descriptor as the first parameter. This file descriptor is the same descriptor that the `open` function returns. The parameter can be omitted if it is the standard input file `stdin`. Unless specified otherwise explicitly, these functions throw an exception if FD is at the end of the file or the next token at FD is not of an expected type.

- `read_atom(`FD`)` = $Atom$: This function reads a single atom from FD.
- `read_atom()` = $Atom$: This function is the same as `read_atom(stdin)`.
- `read_byte(`FD`)` = Int: This function reads a single byte from FD. It returns -1 if FD is at the end of the file.
- `read_byte()` = Int: This function is the same as `read_byte(stdin)`.
- `read_byte(`FD`, `N`)` = $List$: This function reads up to N bytes from FD. It returns the list of bytes that were read.
- `read_char(`FD`)` = $Char$: This function reads a single UTF-8 character from FD. It returns `end_of_file` if FD is at the end of the file.
- `read_char()` = $Char$: This function is the same as `read_char(stdin)`.
- `read_char(`FD`, `N`)` = $String$: This function reads up to N UTF-8 characters from FD. It returns a string that contains the characters that were read.
- `read_char_code(`FD`)` = Int: This function reads a single UTF-8 character from FD and returns its code point. It returns -1 if FD is at the end of the file.
- `read_char_code()` = Int: This function is the same as `read_char_code(stdin)`.
- `read_char_code(`FD`, `N`)` = $List$: This function reads up to N UTF-8 characters from FD. It returns a list of code points of the characters that were read.
- `read_file_bytes(`$File$`)` = $List$: This function reads an entire byte file into a list.
- `read_file_bytes()` = $List$: This function reads an entire byte file from the console into a list.
- `read_file_chars(`$File$`)` = $String$: This function reads an entire character file into a string.
- `read_file_chars()` = $String$: This function reads an entire character file from the console into a string.
- `read_file_codes(`$File$`)` = $List$: This function reads UTF-8 codes of an entire character file into a list.
- `read_file_codes()` = $List$: This function reads UTF-8 codes of an entire character file from the console into a list.

- `read_file_lines(`$File$`) = `$Lines$: This function reads an entire character file into a list of line strings.
- `read_file_lines() = `$Lines$: This function reads an entire character file from the console into a list of line strings.
- `read_file_terms(`$File$`) = `$Terms$: This function reads an entire text file into a list of terms. In the file, each term must be terminated by '.' followed by at least one whitespace character.
- `read_file_terms() = `$Terms$: This function reads an entire text file from the console into a list of terms.
- `read_int(`FD`) = `Int: This function reads a single integer from FD.
- `read_int() = `Int: This function is the same as `read_int(stdin)`.
- `read_line(`FD`) = `$String$: This function reads a line from FD, excluding the terminating newline ('\r\n' on Windows, and '\n' on Unix). It returns `end_of_file` if FD is already at the end of the file.
- `read_line() = `$String$: This function is the same as `read_line(stdin)`.
- `readln(`FD`) = `$String$: This function does the same thing as `read_line(`FD`)`.
- `readln() = `$String$: This function is the same as `readln(stdin)`.
- `read_number(`FD`) = `Num: This function reads a single number (an integer or a real number) from FD.
- `read_number() = `Num: This function is the same as `read_number(stdin)`.
- `read_picat_token(`FD`, `$TokenType$`, `$TokenValue$`)`: This predicate reads a single Picat token from FD. $TokenType$ is the type and $TokenValue$ is the value of the token. $TokenType$ is one of the following:
  - `atom`
  - `end_of_file`
  - `end_of_rule`
  - `integer`
  - `punctuation`
  - `real`
  - `string`
  - `underscore`
  - `var`
- `read_picat_token(`$TokenType$`, `$TokenValue$`)`: This predicate reads a token from `stdin`.
- `read_picat_token(`FD`) = `$TokenValue$: This function reads a single Picat token from FD and returns the token value.
- `read_picat_token() = `$TokenValue$: This function is the same as the above, except that it reads from `stdin`.
- `read_real(`FD`) = `$Real$: This function reads a single real number from FD.
- `read_real() = `$Real$: This function is the same as `read_real(stdin)`.

- `read_term(`FD`) = `$Term$: This function reads a single Picat term from FD. The term must be followed by a dot '`.`' and at least one whitespace character. This function consumes the dot symbol. The whitespace character is not stored in the returned term. It returns `end_of_file` if FD is at the end of the file.
- `read_term() = `$Term$: This function is the same as `read_term(stdin).`

The bulk-read functions, including `read_byte(`FD`, `N`)`, `read_char(`FD`, `N`)`, and `read_char_code(`FD`, `N`)`, may read fewer than N values. One case occurs when the end of the file is encountered. Another case occurs when reading from a pipe. If a pipe is empty, then the `read` functions wait until data is written to the pipe. As soon as the pipe has data, the `read` functions read the data. If a pipe has fewer than N values when a read occurs, then these three functions will return a list that contains all of the values that are currently in the pipe, without waiting for more values. In order to determine the actual number of elements that were read, after the functions return, use `length(`$List$`)` to check the length of the list that was returned.

The `io` module also has functions that peek at the next value in the file without changing the current file location. This means that the next `read` or `peek` function will return the same value, unless the read/write pointer is repositioned or the file is modified.

- `peek_byte(`FD`) = `Int
- `peek_char(`FD`) = `Val

11.2.1 End of File

The end of a file is detected through the return of the `end_of_file` atom if the read function is `read_line` or `read_char`, the return of the value -1 if the read function is `read_byte`, or an exception if the read function is `read_int`. Instead of checking the returned value or exception, the `at_end_of_stream` predicate can be used to monitor a file descriptor for the end of a file.

- `at_end_of_stream(`FD`)`: The `at_end_of_stream` predicate is demonstrated in the following example.

Example

```
rw =>
     Reader = open("file1.txt"),
     Writer = open("file2.txt", write),
     while (not at_end_of_stream(Reader))
            L := read_line(Reader),
            println(Writer, L),
            flush(Writer)
     end,
     close(Reader),
     close(Writer).
```

The advantage of using the `at_end_of_stream` predicate instead of using the `end_of_file` atom is that `at_end_of_stream` immediately indicates that the end of the file was reached, even if the last `read` function read values into a list. In the first example in this chapter, which used the `end_of_file` atom, an extra call to `read_line` was needed before the end of the file was detected. In the above example, which used `at_end_of_stream`, `read_line` was only called if there was data remaining to be read.

11.3 Writing to a File

The `write` and `print` predicates take a file descriptor as the first parameter. The file descriptor is the same descriptor that the `open` function returns. If the file descriptor is `stdout`, then the parameter can be omitted.

- `write(`FD`, `$Term$`)`: This predicate writes $Term$ to a file. Strings are written as lists of single-character atoms. Atoms are single-quoted when necessary. This predicate does not print a newline, meaning that the next write will begin on the same line.
- `write(`$Term$`)`: This predicate is the same as `write(stdout, `$Term$`)`.
- `write_byte(`FD`, `$Bytes$`)`: This predicate writes a single byte or a list of bytes to a file.
- `write_byte(`$Bytes$`)`: This predicate is the same as `write_byte(stdout, `$Bytes$`)`.
- `write_char(`FD`, `$Chars$`)`: This predicate writes a single character or a list of characters to a file. The characters are not quoted. When writing a single-character atom $Char$, `write_char(`FD`, `$Char$`)` is the same as `print(`FD`, `$Char$`)`, but `write_char` is faster than `print`.
- `write_char(`$Chars$`)`: This predicate is the same as `write_char(stdout, `$Chars$`)`.
- `write_char_code(`FD`, `$Codes$`)`: This predicate writes a single character or a list of characters of the given code or list of codes to a file.
- `write_char_code(`$Codes$`)`: This predicate is the same as the above, except that it writes to `stdout`.
- `writef(`FD`, `$Format$`, `$Args\ldots$`)`: This predicate is used for formatted writing, where the $Format$ parameter contains format characters that indicate how to print each of the arguments in the $Args$ parameter. The number of arguments in $Args\ldots$ cannot exceed 16.
- `writeln(`FD`, `$Term$`)`: This predicate writes $Term$ and a newline, meaning that the next write will begin on the next line.
- `writeln(`$Term$`)`: This predicate is the same as `writeln(stdout, `$Term$`)`.

Note that these predicates write both primitive values and compound values.

The `writef` predicate includes a parameter that specifies the string that is to be formatted. The $Format$ parameter is a string that contains format characters. Format characters take the form `%[flags][width][.precision]specifier`. Only the percent sign and the specifier are mandatory. *Flags* can be used for justification and padding. The *width* is the minimum number of characters that are to be printed. The *precision* is the number of characters that are to be printed after the number's radix point. Note that the width includes all characters, including the radix point and the characters that follow it. The *specifier* indicates the type of data that is to be written. A specifier can be one of the C format specifiers `%%`, `%c`, [1] `%d`, `%e`, `%E`, `%f`, `%g`, `%G`, `%i`, `%o`, `%s`, `%u`, `%x`, and `%X`. In addition, Picat uses the specifier `%n` for newlines, and uses `%w` for terms. For details, see Appendix F.

[1] The specifier `%c` can only be used to print ASCII characters. Use the specifier `%w` to print UTF-8 characters.

Example

```
formatted_print =>
    FD = open("birthday.txt", write),
    Format1 = "Hello, %s.  Happy birthday!  ",
    Format2 =  "You are %d years old today.  ",
    Format3 =  "That is %.2f%% older than you were last year.",
    writef(FD, Format1, "Bob"),
    writef(FD, Format2, 7),
    writef(FD, Format3, ((7.0 - 6.0) / 6.0) * 100),
    close(FD).
```

This writes `Hello, Bob. Happy birthday! You are 7 years old today. That is 16.67% older than you were last year.`

The `io` module also provides the `print`, `printf`, and `println` predicate families.

- `print(`FD`, `$Term$`)`: This predicate prints $Term$ to a file. Unlike the `write` predicates, the `print` predicates do not place quotes around strings and atoms.
- `print(`$Term$`)`: This predicate is the same as `print(stdout, `$Term$`)`.
- `printf(`FD`, `$Format$`, `$Args\ldots$`)`: This predicate is the same as `writef`, except that `printf` uses `print` to display the arguments in the $Args$ parameter, while `writef` uses `write` to display the arguments in the $Args$ parameter. The number of arguments in $Args\ldots$ cannot exceed 16.
- `println(`FD`, `$Term$`)`: This predicate prints $Term$ and a newline.
- `println(`$Term$`)`: This predicate is the same as `println(stdout, `$Term$`)`.

In addition to the above built-ins, the `io` module also provides the following:

- `nl(`FD`)`: This predicate prints a newline to FD.
- `nl()`: This predicate prints a newline to `stdout`.

The following example demonstrates the differences between the `write` and `print` predicates.

Example

```
Picat> write("abc")
[a,b,c]
Picat> write([a,b,c])
[a,b,c]
Picat> write('a@b')
'a@b'
Picat> writef("%w %s%n", [a,b,c], "abc")
[a,b,c] abc
Picat> print("abc")
abc
Picat> print([a,b,c])
abc
```

```
Picat> print('a@b')
a@b
Picat> printf("%w %s%n", [a,b,c], "abc")
abc abc
```

11.4 Flushing and Closing a File

The `io` module has one predicate to flush a file stream, and one predicate to close a file stream.

- `flush(`FD`)`: This predicate causes all buffered data to be written without delay.
- `close(`FD`)`: This predicate causes the file to be closed, releasing the file's resources, and removing the file from the file descriptor table. Any further attempts to write to the file descriptor without calling `open` will cause an error to be thrown.

11.5 Standard File Descriptors

The atoms `stdin`, `stdout`, and `stderr` represent the file descriptors for standard input, standard output, and standard error. These atoms allow the program to use the input and output functions of the `io` module to read from and to write to the three standard streams.

Chapter 12

Event-Driven Actors and Action Rules

Many applications require event-driven computing. For example, an interactive GUI system needs to react to UI events such as mouse clicks on UI components; a Web service provider needs to respond to service requests; a constraint propagator for a constraint needs to react to updates to the domains of the variables in the constraint. Picat provides action rules for describing event-driven actors. An actor is a predicate call that can be delayed and can be activated later by events. Actors communicate with each other through event channels.

12.1 Channels, Ports, and Events

An event channel is an attributed variable to which actors can be attached, and through which events can be posted to actors. A channel has four ports, named `ins`, `bound`, `dom`, and `any`. Many built-ins in Picat post events. When an attributed variable is instantiated, an event is posted to the `ins`-port of the variable. When the lower bound or upper bound of a variable's domain changes, an event is posted to the `bound`-port of the variable. When an inner element E, which is neither the lower nor the upper bound, is excluded from the domain of a variable, E is posted to the `dom`-port of the variable. When an arbitrary element E, which can be the lower or upper bound or an inner element, is excluded from the domain of a variable, E is posted to the `any`-port of the variable. The division of a channel into ports facilitates speedy handling of events. For better performance, the system posts an event to a port only when there are actors attached to the port. For example, if no actor is attached to a domain variable to handle exclusions of domain elements, then these events will never be posted.

The built-in `post_event(`X`, `T`)` posts the event term T to the `dom`-port of the channel variable X.

The following built-ins are used to post events to one of a channel's four ports:

- `post_event_ins(`X`)`: posts an event to the `ins`-port of the channel X.
- `post_event_bound(`X`)`: posts an event to the `bound`-port of the channel X.
- `post_event_dom(`X`, `T`)`: posts the term T to the `dom`-port of the channel X.
- `post_event_any(`X`, `T`)`: posts the event T to the `any`-port of the channel of X.

The call `post_event(`X`, `T`)` is the same as `post_event_dom(`X`, `T`)`. This means that the `dom`-port of a finite domain variable has two uses: posting exclusions of inner elements from the domain, and posting general term events.

12.2 Action Rules

Picat provides *action rules* for describing the behaviors of actors. An action rule takes the following form:

$Head, Cond, \{Event\} => Body$

where $Head$ is an actor pattern, $Cond$ is an optional condition, $Event$ is a non-empty set of event patterns separated by `','`, and $Body$ is an action. For an actor that is activated by an event, an action rule is said to be *applicable* if the actor matches $Head$ and $Cond$ is true. A predicate for actors is defined with action rules and non-backtrackable rules. It cannot contain backtrackable rules.

Unlike rules for a normal predicate or function, in which the conditions can contain any predicates, the conditions of the rules in a predicate for actors must be conjunctions of inline test predicates, such as type-checking built-ins (e.g., `integer(`X`)` and `var(`X`)`) and comparison built-ins (e.g., equality test X `==` Y, disequality test X `!==` Y, and arithmetic comparison $X > Y$). This restriction ensures that no variables in an actor can be changed while the condition is executed.

For an actor that is activated by an event, the system searches the definition sequentially from the top for an applicable rule. If no applicable rule is found, then the actor fails. If an applicable rule is found, the system executes the body of the rule. If the body fails, then the actor also fails. The body cannot succeed more than once. The system enforces this by converting $Body$ into `'once` $Body$`'` if $Body$ contains calls to nondeterministic predicates. If the applied rule is an action rule, then the actor is suspended after the body is executed, meaning that the actor is waiting to be activated again. If the applied rule is a normal non-backtrackable rule, then the actor vanishes after the body is executed. For each activation, only the first applicable rule is applied.

For a call and an action rule '$Head, Cond, \{Event\} => Body$', the call is registered as an actor if the call matches $Head$ and $Cond$ evaluates to `true`. The event pattern $Event$ implicitly specifies the ports to which the actor is attached, and the events that the actor watches. The following event patterns are allowed in $Event$:

- `event(`X`, `T`)`: This is the general event pattern. The actor is attached to the `dom`-ports of the variables in X. The actor will be activated by events posted to the `dom`-ports. T must be a variable that does not occur before `event(`X`, `T`)` in the rule.

- `ins(`X`)`: The actor is attached to the `ins`-ports of the variables in X. The actor will be activated when a variable in X is instantiated.

- `bound(`X`)`: The actor is attached to the `bound`-ports of the variables in X. The actor will be activated when the lower bound or upper bound of the domain of a variable in X changes.

- `dom(`X`)`: The actor is attached to the `dom`-ports of the variables in X. The actor will be activated when an inner value is excluded from the domain of a variable in X. The actor is not interested in what value is actually excluded.

- `dom(`X`, `E`)`: This is the same as `dom(`X`)`, except the actor is interested in the value E that is excluded. E must be a variable that does not occur before `dom(`X`, `E`)` in the rule.

- `dom_any(X)`: The actor is attached to the `any`-ports of the variables in X. The actor will be activated when an arbitrary value, including the lower bound value and the upper bound value, is excluded from the domain of a variable in X. The actor is not interested in what value is actually excluded.

- `dom_any(`X`, `E`)`: This is the same as `dom_any(`X`)`, except the actor is interested in the value E that is actually excluded. E must be a variable that does not occur before `dom_any(`X`, `E`)` in the rule.

In an action rule, multiple event patterns can be specified. After a call is registered as an actor on the channels, it will be suspended, waiting for events, unless the atom `generated` occurs in $Event$, in which case the actor will be suspended after $Body$ is executed.

The system has an event queue. After events are posted, they are added into the queue. Events are not handled until execution enters or exits a non-inline predicate or function. In other words, only non-inline predicates and functions can be interrupted, and inline predicates, such as $X = Y$, and inline functions, such as $X + Y$, are never interrupted by events.

Example

Consider the following action rule:

```
p(X), {event(X,T)} => writeln(T).
```

The following gives a query and its output:

```
Picat> p(X), X.post_event(ping), X.post_event(pong)
ping
pong
```

The call `p(X)` is an actor. After `X.post_event(ping)`, the actor is activated and the body of the action rule is executed, giving the output `ping`. After `X.post_event(pong)`, the actor is activated again, outputting `pong`.

There is no primitive for killing actors or explicitly detaching actors from channels. As described above, an actor never disappears as long as action rules are applied to it. An actor vanishes only when a normal rule is applied to it. Consider the following example.

```
p(X, Flag),
    var(Flag),
    {event(X,T)}
=>
    writeln(T),
    Flag = 1.
p(_, _) => true.
```

An actor defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. When a second event is posted, the action rule is no longer applicable, causing the second rule to be selected.

One question arises here: what happens if a second event is never posted to `X`? In this case, the actor will stay forever. If users want to immediately kill the actor after it is activated once, then users have to define it as follows:

```
p(X, Flag),
    var(Flag),
    {event(X,O),ins(Flag)}
=>
    write(O),
    Flag = 1.
p(_, _) => true.
```

In this way, the actor will be activated again after `Flag` is bound to 1, and will be killed after the second rule is applied to it.

12.3 Lazy Evaluation

The built-in predicate `freeze(X, Goal)` is equivalent to '`once Goal`', but its evaluation is delayed until `X` is bound to a non-variable term. The predicate is defined as follows:

```
freeze(X, Goal), var(X), {ins(X)} => true.
freeze(X, Goal) => call(Goal).
```

For the call `freeze(X, Goal)`, if `X` is a variable, then the call is registered as an actor on the `ins`-port of `X`, and the call is then suspended. Whenever `X` is bound, the event `ins` is posted to the `ins`-port of `X`, which activates the actor `freeze(X, Goal)`. The condition `var(X)` is checked. If true, the actor is suspended again; otherwise, the second rule is executed, causing the actor to vanish after it is rewritten into `once Goal`.

The built-in predicate `different_terms(`T_1`, `T_2`)` is a disequality constraint on terms T_1 and T_2. The constraint fails if the two terms are identical; it succeeds whenever the two terms are found to be different; it is delayed if no decision can be made because the terms are not sufficiently instantiated. The predicate is defined as follows:

```
import cp.

different_terms(X, Y) =>
    different_terms(X, Y, 1).

different_terms(X, Y, B), var(X), {ins(X),ins(Y)} => true.
different_terms(X, Y, B), var(Y), {ins(Y)} => true.
different_terms([X|Xs], [Y|Ys], B) =>
    different_terms(X, Y, B1),
    different_terms(Xs, Ys, B2),
    B #= (B1 #\/ B2).
different_terms(X, Y, B), struct(X), struct(Y) =>
    if (X.name !== Y.name; X.length !== Y.length)
       B = 1
    else
       Bs = new_list(X.length),
       foreach (I in 1 .. X.length)
           different_terms(X[I], Y[I], Bs[I])
       end,
       max(Bs) #= B
    end.
different_terms(X, Y, B), X == Y => B = 0.
different_terms(X, Y, B) => B = 1.
```

The call `different_terms(X, Y, B)` is delayed if either `X` or `Y` is a variable. The delayed call watches `ins(X)` and `ins(Y)` events. Once both `X` and `Y` become non-variable, the action rule becomes inapplicable, and one of the subsequent rules will be applied. If `X` and `Y` are lists, then they are different if the heads are different (`B1`), or if the tails are different (`B2`). This relationship is represented as the Boolean constraint `B #= (B1 #\/ B2)`. If `X` and `Y` are both structures, then they are different if the functor is different, or if any pair of arguments of the structures is different.

12.4 Constraint Propagators

A constraint propagator is an actor that reacts to updates of the domains of the variables in a constraint. The following predicate defines a propagator for maintaining arc consistency on `X` for the constraint `X+Y #= C`:

```
import cp.

x_in_c_y_ac(X, Y, C), var(X), var(Y),
   {dom(Y,Ey)}
    =>
   fd_set_false(X, C-Ey).
x_in_c_y_ac(X, Y, C) => true.
```

Whenever an inner element `Ey` is excluded from the domain of `Y`, this propagator is triggered to exclude `C-Ey`, which is the support of `Ey`, from the domain of `X`. For the constraint `X+Y #= C`, users need to generate two propagators, namely,

`x_in_c_y_ac(X, Y, C)` and `x_in_c_y_ac(Y, X, C)`

to maintain the arc consistency. Note that in addition to these two propagators, users also need to generate propagators for maintaining interval consistency, because `dom(Y, Ey)` only captures exclusions of inner elements, and does not capture bounds. The following propagator maintains interval consistency for the constraint:

```
import cp.

x_add_y_eq_c_ic(X, Y, C), var(X), var(Y),
    {generated,ins(X),ins(Y),bound(X),bound(Y)}
=>
    X :: C-fd_max(Y) .. C-fd_min(Y),
    Y :: C-fd_max(X) .. C-fd_min(X).
x_add_y_eq_c_ic(X, Y, C), var(X) =>
    X = C-Y.
x_add_y_eq_c_ic(X, Y, C) =>
    Y = C-X.
```

When both `X` and `Y` are variables, the propagator `x_add_y_eq_c_ic(X, Y, C)` is activated whenever either `X` or `Y` is instantiated, or whenever the bounds of their domains are updated. The body maintains the interval consistency of the constraint `X+Y #= C`. The body is also executed when the propagator is generated. When either `X` or `Y` becomes non-variable, the propagator becomes a normal call, and vanishes after the variable `X` or `Y` is solved.

Chapter 13

Constraints

Picat provides four solver modules, including `cp`, `sat`, `smt`, and `mip`, for modeling and solving constraint satisfaction and optimization problems (CSPs). All four of these modules implement the same set of constraints on integer-domain variables. The `mip` module also supports real-domain variables. In order to use a solver, users must first import the module. In order to make the symbols defined in a module available to the top level of the interpreter, users can use the built-in `import` to import the module. The `mip` module requires Gurobi (`https://www.gurobi.com/`), Cbc (`https://github.com/coin-or/Cbc`), SCIP (`https://github.com/scipopt/scip`), or GLPK (`https://www.gnu.org/software/glpk/`). The `smt` module requires z3 (`https://github.com/Z3Prover/z3/`) or CVC4 (`https://cvc4.github.io/`). Users can specify, as a solving option, a solver to be used if the module is `mip` or `smt`.

As the four modules have the same interface, this chapter describes the four modules together. Table 13.1 shows the constraint operators that are provided by Picat. Unless it is explicitly specified otherwise, the built-ins that are described in this chapter appear in all four modules. In the built-ins that are presented in this chapter, an integer-domain variable can also be an integer, unless it is explicitly specified to only be a variable.

Table 13.1: Constraint operators in Picat

Precedence	Operators
Highest	`::`, `notin`, `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, `#>=`
	`#~`
	`#/\`
	`#^`
	`#\/`
	`#=>`
Lowest	`#<=>`

A constraint program normally poses a problem in three steps: (1) generate variables; (2) generate constraints over the variables; and (3) call `solve` to find a valuation for the variables that satisfies the constraints and possibly optimizes an objective function.

Example

The program in Figure 13.1 imports the `cp` module in order to solve the N-queens problem. The same program runs with the SAT solver if `sat` is imported, runs with the SMT solver if `smt` is

```
import cp.

queens(N) =>
    Qs = new_array(N),
    Qs :: 1..N,
    foreach (I in 1..N-1, J in I+1..N)
        Qs[I] #!= Qs[J],
        abs(Qs[I]-Qs[J]) #!= J-I
    end,
    solve(Qs),
    writeln(Qs).
```

Figure 13.1: A program for N-queens.

imported, and runs with the LP/MIP solver if `mip` is imported. The predicate `Qs :: 1..N` declares the domains of the variables. The operator `#!=` is used for inequality constraints. In arithmetic constraints, expressions are treated as terms, and it is unnecessary to enclose them with dollar-signs. The predicate `solve(Qs)` calls the solver in order to solve the array of variables `Qs`. For `cp`, `solve([ff], Qs)`, which always selects a variable that has the smallest domain (the so-called *first-fail principle*), can be more efficient than `solve(Qs)`.

13.1 Domain Variables

A domain variable is an attributed variable that has a domain attribute. The Boolean domain is treated as a special integer domain where 1 denotes `true` and 0 denotes `false`. Domain variables are declared with the built-in predicate $Vars$ `::` Exp.

- $Vars$ `::` Exp: This predicate restricts the domain or domains of $Vars$ to Exp. $Vars$ can be either a single variable or a collection of variables.[1] For integer-domain variables, Exp must result in a list of integer values. For real-domain variables for the `mip` module, Exp must be an interval in the form $L..U$, where L and U are real values.

Domain variables, when being created, are usually represented internally by using intervals. An interval representation changes to a bit-vector representation when a hole occurs in the interval. The following built-in predicate can be used to reset the range or access the current range.

- `fd_vector_min_max(`Min`,` Max`)`: When the arguments are integers, this predicate resets the range of bit vectors to `min(Min, -3200)..max(Max, 3200)`; when the arguments are variables, this predicate binds them to the current bounds of the range. The default range is `-3200..3200`, and it can only be expanded.

The following built-ins are provided for domain variables.

- $Vars$ `notin` Exp: This predicate excludes values Exp from the domain or domains of $Vars$, where $Vars$ and Exp are the same as in $Vars$ `::` Exp. This constraint cannot be applied to real-domain variables.

- `fd_degree(`$FDVar$`) =` $Degree$: This function returns the number of propagators that are attached to $FDVar$. This built-in is only provided by `cp`.

[1] A collection can be either a list or an array, where elements can be collections.

- `fd_disjoint(`$FDVar1$`, `$FDVar2$`)`: This predicate is true if $FDVar1$'s domain and $FDVar2$'s domain are disjoint.

- `fd_dom(`$FDVar$`) = `$List$: This function returns the domain of $FDVar$ as a list, where $FDVar$ is an integer-domain variable. If $FDVar$ is an integer, then the returned list contains the integer itself.

- `fd_false(`$FDVar$`, `Elm`)`: This predicate is true if the integer Elm is not an element in the domain of $FDVar$.

- `fd_max(`$FDVar$`) = `Max: This function returns the upper bound of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.

- `fd_min(`$FDVar$`) = `Min: This function returns the lower bound of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.

- `fd_min_max(`$FDVar$`, `Min`, `Max`)`: This predicate binds Min to the lower bound of the domain of $FDVar$, and binds Max to the upper bound of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.

- `fd_next(`$FDVar$`, `Elm`) = `$NextElm$: This function returns the element following Elm in $FDVar$'s domain. It throws an exception if Elm has no next element in $FDVar$'s domain.

- `fd_prev(`$FDVar$`, `Elm`) = `$PrevElm$: This function returns the element preceding Elm in $FDVar$'s domain. It throws an exception if Elm has no previous element in $FDVar$'s domain.

- `fd_set_false(`$FDVar$`, `Elm`)`: This predicate excludes the element Elm from the domain of $FDVar$. If this operation results in a hole in the domain, then the domain changes from an interval representation into a bit-vector representation, no matter how big the domain is. This built-in is only provided by `cp`.

- `fd_size(`$FDVar$`) = `$Size$: This function returns the size of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.

- `fd_true(`$FDVar$`, `Elm`)`: This predicate is true if the integer Elm is an element in the domain of $FDVar$.

- `new_dvar() = `$FDVar$: This function creates a new domain variable with the default domain, which has the bounds `-72057594037927935..72057594037927935` on 64-bit computers and `-268435455..268435455` on 32-bit computers.[2]

13.2 Table Constraints

A *table constraint*, or an *extensional constraint*, over a tuple of variables specifies a set of tuples that are allowed (called *positive*) or disallowed (called *negative*) for the variables. A positive constraint takes the form `table_in(`$DVars$`, `R`)`, where $DVars$ is either a tuple of variables $\{X_1, \ldots, X_n\}$ or a list of tuples of variables, and R is a list of tuples in which each tuple takes the form $\{a_1, \ldots, a_n\}$, where a_i is an integer or the *don't-care* symbol $*$. A negative constraint takes the form `table_notin(`$DVars$`, `R`)`.

[2]Note that incorrect results may be caused by overflows or underflows.

Example

The following example solves a toy crossword puzzle. One variable is used for each cell in the grid, so each slot corresponds to a tuple of variables. Each word is represented as a tuple of integers, and each slot takes on a set of words of the same length as the slot. Recall that the function `ord(`$Char$`)` returns the code of $Char$, and that the function `chr(`$Code$`)` returns the character of $Code$.

```
import cp.

crossword(Vars) =>
    Vars = [X1,X2,X3,X4,X5,X6,X7],
    Words2 = [{ord('I'),ord('N')},
              {ord('I'),ord('F')},
              {ord('A'),ord('S')},
              {ord('G'),ord('O')},
              {ord('T'),ord('O')}],
    Words3 = [{ord('F'),ord('U'),ord('N')},
              {ord('T'),ord('A'),ord('D')},
              {ord('N'),ord('A'),ord('G')},
              {ord('S'),ord('A'),ord('G')}],
    table_in([{X1,X2},{X1,X3},{X5,X7},{X6,X7}], Words2),
    table_in([{X3,X4,X5},{X2,X4,X6}], Words3),
    solve(Vars),
    writeln([chr(Code) : Code in Vars]).
```

13.3 Arithmetic Constraints

An arithmetic constraint takes the form

$Exp1$ Rel $Exp2$

where $Exp1$ and $Exp2$ are arithmetic expressions, and Rel is one of the constraint operators: `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, or `#>=`. The operators `#=<` and `#<=` are the same, meaning less than or equal to. An arithmetic expression is made from integers, variables, arithmetic functions, and constraints. The following arithmetic functions are allowed: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (truncated integer division), `//` (truncated integer division), `count`, `div` (floored integer division), `mod`, `**` (power), `abs`, `min`, `max`, and `sum`. Except for index notations, array comprehensions and list comprehensions, which are interpreted as function calls as in normal expressions, expressions in arithmetic constraints are treated as terms, and it is unnecessary to enclose them with dollar-signs. In addition to the numeric operators, the following functions are allowed in constraints:[3]

- `cond(`$BoolConstr$`,` $ThenExp$`,` $ElseExp$`)`: This expression is the same as $BoolConstr * ThenExp +$ `(1-`$BoolConstr$`)` $* ElseExp$.
- `count(`V`,` $Collection$`)`: The number of times V occurs in $Collection$, where $Collection$ is a nested collection of domain variables.[4]

[3] Dot notation should not be used for functions in constraint expressions, as such functions are always evaluated eagerly.

[4] A nested collection is recursively defined. It can be a variable, a number, a list, or an array. If it is a list or an array, then the elements can be nested collections.

- `max(`$Collection$`)`: The maximum of $Collection$, where $Collection$ is a nested collection of domain variables.
- `max(`$Exp1$`, `$Exp2$`)`: The maximum of $Exp1$ and $Exp2$.
- `min(`$Collection$`)`: The minimum of $Collection$, where $Collection$ is a nested collection of domain variables.
- `min(`$Exp1$`, `$Exp2$`)`: The minimum of $Exp1$ and $Exp2$.
- `prod(`$Collection$`)`: The product of $Collection$, where $Collection$ is a nested collection of domain variables.
- `sum(`$Collection$`)`: The sum of $Collection$, where $Collection$ is a nested collection of domain variables.

When a constraint occurs in an arithmetic expression, it is evaluated to 1 if it is satisfied and 0 if it is not satisfied.

Example

```
import mip.

go =>
    M = {{0,3,2,3,0,0,0,0},
         {0,0,0,0,0,0,5,0},
         {0,1,0,0,0,1,0,0},
         {0,0,2,0,2,0,0,0},
         {0,0,0,0,0,0,0,5},
         {0,4,0,0,2,0,0,1},
         {0,0,0,0,0,2,0,3},
         {0,0,0,0,0,0,0,0}},
    maxflow(M, 1, 8).

maxflow(M, Source, Sink) =>
    N = M.length,
    X = new_array(N, N),
    foreach (I in 1..N, J in 1..N)
        X[I, J] :: 0..M[I, J]
    end,
    foreach (I in 1..N, I != Source, I != Sink)
        sum([X[J,I] : J in 1..N])
          #= sum([X[I,J] : J in 1..N])
    end,
    Total #= sum([X[Source,I] : I in 1..N]),
    Total #= sum([X[I,Sink] : I in 1..N]),
    solve([$max(Total)], X),
    writeln(Total),
    writeln(X).
```

This program uses MIP to solve the maximum integer flow problem. Given the capacity matrix `M` of a directed graph, the start vertex `Source`, and the destination vertex `Sink`, the predicate

`maxflow(M, Source, Sink)` finds a maximum flow from `Source` to `Sink` over the graph. When two vertices are not connected by an arc, the capacity is given as 0. The first foreach loop specifies the domains of the variables. For each variable `X[I, J]`, the domain is restricted to integers between 0 and the capacity, `M[I, J]`. If the capacity is 0, then the variable is immediately instantiated to 0. The next foreach loop posts the conservation constraints. For each vertex `I`, if it is neither the source nor the sink, then its total incoming flow amount

```
sum([X[J,I] : J in 1..N])
```

is equal to the total outgoing flow amount

```
sum([X[I,J] : J in 1..N]).
```

The total flow amount is the total outgoing amount from the source, which is the same as the total incoming amount to the sink.

13.4 Boolean Constraints

A Boolean constraint takes one of the following forms:

#~ $BoolExp$
$BoolExp$ #/\ $BoolExp$
$BoolExp$ #^ $BoolExp$
$BoolExp$ #\/ $BoolExp$
$BoolExp$ #=> $BoolExp$
$BoolExp$ #<=> $BoolExp$

$BoolExp$ is either a Boolean constant (0 or 1), a Boolean variable (an integer-domain variable with the domain [0,1]), an arithmetic constraint, a domain constraint (in the form of Var `::` $Domain$ or Var `notin` $Domain$), or a Boolean constraint. As shown in Table 13.1, the operator #~ has the highest precedence among Boolean constraint operators, and the operator #<=> has the lowest precedence. Note that the Boolean constraint operators have lower precedence than the arithmetic constraint operators. So the constraint

```
X #!= 3 #/\ X#!= 5 #<=> B
```

is interpreted as

```
((X #!= 3) #/\ (X#!= 5)) #<=> B.
```

The Boolean constraint operators are defined as follows.

- #~ $BoolExp$: This constraint is 1 iff $BoolExp$ is equal to 0.
- $BoolExp1$ #/\ $BoolExp2$: This constraint is 1 iff both $BoolExp1$ and $BoolExp2$ are 1.
- $BoolExp1$ #^ $BoolExp2$: This constraint is 1 iff exactly one of $BoolExp1$ and $BoolExp2$ is 1.
- $BoolExp1$ #\/ $BoolExp2$: This constraint is 1 iff $BoolExp1$ or $BoolExp2$ is 1.
- $BoolExp1$ #=> $BoolExp2$: This constraint is 1 iff $BoolExp1$ implies $BoolExp2$.
- $BoolExp1$ #<=> $BoolExp2$: This constraint is 1 iff $BoolExp1$ and $BoolExp2$ are equivalent.

13.5 Global Constraints

A global constraint is a constraint over multiple variables. A global constraint can normally be translated into a set of smaller constraints, such as arithmetic and Boolean constraints. If the `cp` module is used, then global constraints are generally not translated into smaller constraints; rather, they are compiled into special propagators that maintain a certain level of consistency for the constraints. In Picat, constraint propagators are encoded as action rules. If the `sat` module is used, then global constraints are translated into smaller constraints before being translated further into conjunctive normal form. If the `mip` module is used, then global constraints are decomposed into equality and disequality constraints.

Picat provides the following global constraints.

- `acyclic(`Vs`, `Es`)`: This constraint ensures that the *undirected* graph represented by Vs and Es contains no cycles, where Vs is a list of pairs of the form $\{V, B\}$ and Es is a list of triplets of the form $\{V_1, V_2, B\}$. A pair $\{V, B\}$ in Vs, where V is a ground term and B is a Boolean variable, denotes that V is in the graph if and only if $B = 1$. A triplet $\{V_1, V_2, B\}$ denotes that V_1 is connected with V_2 by an edge in the graph if and only if $B = 1$ and both V_1 and V_2 are in the graph. Note that the graph to be constructed is assumed to be undirected. If there exists a triplet $\{V_1, V_2, B\}$ in Es, then the triplet $\{V_2, V_1, B\}$ will be added to Es if it is not specified.

- `acyclic_d(`Vs`, `Es`)`: This constraint ensures that the *directed* graph represented by Vs and Es contains no cycles, where Vs and Es are the same as those in `acyclic(`Vs`, `Es`)`, except that the graph is directed.

- `all_different(`$FDVars$`)`: This constraint ensures that each pair of variables in the list or array $FDVars$ is different. This constraint is compiled into a set of inequality constraints. For each pair of variables $V1$ and $V2$ in $FDVars$, `all_different(`$FDVars$`)` generates the constraint $V1$ `#!=` $V2$.

- `all_different_except_0(`$FDVars$`)`: This constraint is true if all non-zero values in $FDVars$ are different.

- `all_distinct(`$FDVars$`)`: This constraint is the same as `all_different`, but for the `cp` module it maintains a higher level of consistency. For some problems, this constraint is faster and requires fewer backtracks than `all_different`, and, for some other problems, this constraint is slower due to the overhead of consistency checking.

- `assignment(`$FDVars1$`, `$FDVars2$`)`: This constraint ensures that $FDVars2$ is a *dual assignment* of $FDVars1$, i.e., if the ith element of $FDVars1$ is j, then the jth element of $FDVars2$ is i. The constraint can be defined as:

```
assignment(Xs, Ys) =>
    N = Xs.length,
    (var(Ys) -> Ys = new_list(N); true),
    Xs :: 1..N,
    Ys :: 1..N,
    foreach (I in 1..N, J in 1..N)
        Xs[I] #= J #<=> Ys[J] #= I
    end.
```

- `at_least(`N`, `L`, `V`)`: This constraint succeeds if there are at least N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

- `at_most(`N`, `L`, `V`)`: This constraint succeeds if there are at most N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

- `circuit(`$FDVars$`)`: Let $FDVars$ be a list of variables `[`$X_1, X_2, \ldots, X_n$`]`, where each X_i has the domain $1..n$. A valuation $X_1 = v_1$, $X_2 = v_2$, ..., $X_n = v_n$ satisfies the constraint if `1 ->` v_1`, 2 ->` v_2`, ..., n ->` v_n forms a Hamiltonian cycle. This constraint ensures that each variable has a different value, and that the graph that is formed by the assignment does not contain any sub-cycles. For example, for the constraint

  ```
  circuit([X1,X2,X3,X4])
  ```

  `[3,4,2,1]` is a solution, but `[2,1,4,3]` is not, because the graph `1 -> 2, 2 -> 1, 3 -> 4, 4 -> 3` contains two sub-cycles.

- `count(`V`, `$FDVars$`, `Rel`, `N`)`: In this constraint, V and N are integer-domain variables, $FDVars$ is a list of integer-domain variables, and Rel is an arithmetic constraint operator (`#=`, `#!=`, `#>`, `#>=`, `#<`, `#=<`, or `#<=`). Let $Count$ be the number of elements in $FDVars$ that are equal to V. The constraint is true iff $Count\ Rel\ N$ is true. This constraint can be defined as follows:

  ```
  count(V, L, Rel, N) =>
     sum([V #= E : E in L]) #= Count,
     call(Rel, Count, N).
  ```

- `count(`V`, `$FDVars$`, `N`)`: This constraint is the same as `count(`V, $FDVars$, `#=`, N`)`.

- `cumulative(`$Starts$`, `$Durations$`, `$Resources$`, `$Limit$`)`: This constraint is useful for describing and solving scheduling problems. The arguments $Starts$, $Durations$, and $Resources$ are lists of integer-domain variables of the same length, and $Limit$ is an integer-domain variable. Let $Starts$ be `[`S_1`, `S_2`, ..., `S_n`]`, $Durations$ be `[`D_1`, `D_2`, ..., `D_n`]`, and $Resources$ be `[`R_1`, `R_2`, ..., `R_n`]`. For each job i, S_i represents the start time, D_i represents the duration, and R_i represents the units of resources needed. $Limit$ is the limit on the units of resources available at any time. This constraint ensures that the limit cannot be exceeded at any time.

- `decreasing(`L`)`: The sequence (an array or a list) L is in (non-strictly) decreasing order.

- `decreasing_strict(`L`)`: The sequence (an array or a list) L is in strictly decreasing order.

- `diffn(`$RectangleList$`)`: This constraint ensures that no two rectangles in $RectangleList$ overlap with each other. A rectangle in an n-dimensional space is represented by a list of $2 \times n$ elements `[`X_1`, `X_2`, ..., `X_n`, `S_1`, `S_2`, ..., `S_n`]`, where X_i is the starting coordinate of the edge in the ith dimension, and S_i is the size of the edge.

- `disjunctive_tasks(`$Tasks$`)` (cp only): $Tasks$ is a list of terms. Each term has the form `disj_tasks(`S_1`, `D_1`, `S_2`, `D_2`)`, where S_1 and S_2 are two integer-domain variables, and D_1 and D_2 are two positive integers. This constraint is equivalent to posting the disjunctive constraint S_1+D_1 `#=<` S_2 `#\/` S_2+D_2 `#=<` S_1 for each term in $Tasks$; however, the constraint may be more efficient, because it converts the disjunctive tasks into global constraints.

- `element(`I`, `$List$`, `V`)`: This constraint is true if the element at index I in $List$ is V, where I and V are integer-domain variables, and $List$ is a list of integer-domain variables.

- `element0(`I`, `$List$`, `V`)`: This constraint is the same as `element(`I`, `$List$`, `V`)`, except that 0-based, rather than 1-based, indexing is used.

- `exactly(`N`, `L`, `V`)`: This constraint succeeds if there are exactly N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

- `global_cardinality(`$List$`, `$Pairs$`)`: Let $List$ be a list of integer-domain variables `[`X_1`, ..., `X_d`]`, and $Pairs$ be a list of pairs `[`K_1`-`V_1`, ..., `K_n`-`V_n`]`, where each key K_i is a unique integer, and each V_i is an integer-domain variable. The constraint is true if every element of $List$ is equal to some key, and, for each pair K_i-V_i, exactly V_i elements of $List$ are equal to K_i. This constraint can be defined as follows:

```
global_cardinality(List, Pairs) =>
    Keys = [Key : $(Key-_) in Pairs],
    List :: Keys,
    foreach ($Key-V in Pairs)
        sum([B : E in List, B#<=>(E#=Key)]) #= V
    end.
```

- `hcp(`Vs`, `Es`)`: This constraint ensures that the *directed* graph represented by Vs and Es forms a Hamiltonian cycle, where Vs is a list of pairs of the form $\{V, B\}$, and Es is a list of triplets of the form $\{V_1, V_2, B\}$. A pair $\{V, B\}$ in Vs, where V is a ground term and B is a Boolean variable, denotes that V is in the graph if and only if $B = 1$. A triplet $\{V_1, V_2, B\}$ denotes that V_1 is connected to V_2 by an edge in the graph if and only if $B = 1$. The `circuit` and `subcircuit` constraints can be implemented as follows by using `hcp`:

```
circuit(L) =>
    N = len(L),
    L :: 1..N,
    Vs = [{I,1} : I in 1..N],
    Es = [{I,J,B} : I in 1..N,
                    J in fd_dom(L[I]),
                    J !== I,
                    B #<=> L[I] #= J],
    hcp(Vs, Es).

subcircuit(L) =>
    N = len(L),
    L :: 1..N,
    Vs = [{I,B} : I in 1..N,
```

```
                    B #<=> L[I] #!= I],
        Es = [{I,J,B} : I in 1..N,
                        J in fd_dom(L[I]),
                        J !== I,
                        B #<=> L[I] #= J],
        hcp(Vs, Es).
```

- `hcp(`Vs`, `Es`, `K`)`: This constraint is the same as `hcp(`Vs`, `Es`)`, except that it also constrains the number of vertices in the graph to be K.

- `hcp_grid(`A`)`: This constraint ensures that the *grid* graph represented by A, which is a two-dimensional array of Boolean (0/1) variables, forms a Hamiltonian cycle. In a grid graph, each cell is directly connected horizontally and vertically, but not diagonally, to its neighbors. Only cells labeled 1 are considered as vertices of the graph. This constraint is implemented as follows by using `hcp`:

```
hcp_grid(A) =>
    NRows = len(A),
    NCols = len(A[1]),
    Vs = [{(R,C), A[R,C]} :
                R in 1..NRows,
                C in 1..NCols],
    Es = [{(R,C), (R1,C1), _} :
                R in 1..NRows,
                C in 1..NCols,
                (R1,C1) in neibs(A,NRows,NCols,R,C)],
    hcp(Vs, Es).

neibs(A, NRows, NCols, R, C) =
    [(R1,C1) : (R1,C1) in [(R-1,C), (R+1,C),
                           (R,C-1), (R,C+1)],
                R1 >= 1, R1 =< NRows,
                C1 >= 1, C1 =< NCols,
                A[R1,C1] !== 0].
```

- `hcp_grid(`A`, `Es`)`: This constraint is the same as `hcp_grid(`A`)`, except that it also restricts the edges to Es, which consists of triplets of the form $\{V_1, V_2, B\}$. In a triplet in Es, V_1 and V_2 take the form (R, C), where R is a row number and C is a column number, and B is a Boolean variable, which denotes that V_1 is connected to V_2 by an edge in the graph if and only if $B = 1$. If Es is a variable, then it is bound to the edges of the grid graph.

- `hcp_grid(`A`, `Es`, `K`)`: This constraint is the same as `hcp_grid(`A`, `Es`)`, except that it also constrains the number of vertices in the graph to be K.

- `increasing(`L`)`: The sequence (an array or a list) L is in (non-strictly) increasing order.

- `increasing_strict(`L`)`: The sequence (an array or a list) L is in strictly increasing order.

- `lex_le(`L_1`, `L_2`)`: The sequence (an array or a list) L_1 is lexicographically less than or equal to L_2.

- `lex_lt(`L_1`, `L_2`)`: The sequence (an array or a list) L_1 is lexicographically less than L_2.
- `matrix_element(`$Matrix$`, `I`, `J`, `V`)`: This constraint is true if the entry at $<I$, $J>$ in $Matrix$ is V, where I, J, and V are integer-domain variables, and $Matrix$ is a two-dimensional array of integer-domain variables.
- `matrix_element0(`$Matrix$`, `I`, `J`, `V`)`: This constraint is the same as the above, except that it uses 0-based, rather than 1-based, indexing for the entries of the matrix.
- `neqs(`$NeqList$`)` (cp only): $NeqList$ is a list of inequality constraints of the form X `#!=` Y, where X and Y are integer-domain variables. This constraint is equivalent to the conjunction of the inequality constraints in $NeqList$, but it extracts `all_distinct` constraints from the inequality constraints.
- `nvalue(`N`, `$List$`)`: The number of distinct values in $List$ is N, where $List$ is a list of integer-domain variables.
- `path(`Vs`, `Es`, `Src`, `$Dest$`)`: This constraint ensures that the *undirected* graph represented by Vs and Es is a path from Src to $Dest$, where Vs is a list of pairs of the form $\{V, B\}$, Es is a list of triplets of the form $\{V_1, V_2, B\}$, Src is a vertex, and $Dest$ is a vertex or a list of vertices. A pair $\{V, B\}$ in Vs, where V is a ground term representing a vertex and B is a Boolean variable, denotes that V is in the graph if and only if $B = 1$. A triplet $\{V_1, V_2, B\}$ denotes that V_1 is connected with V_2 by an edge in the graph if and only if $B = 1$ and both V_1 and V_2 are in the graph. Note that the graph to be constructed is assumed to be undirected. If there exists a triplet $\{V_1, V_2, B\}$ in Es, then the triplet $\{V_2, V_1, B\}$ will be added to Es if it is not specified.
- `path_d(`Vs`, `Es`, `Src`, `$Dest$`)`: This constraint ensures that the *directed* graph represented by Vs and Es is a path from Src to $Dest$, where Vs and Es are the same as those in `path(`Vs`, `Es`, `Src`, `$Dest$`)`, except that the graph is directed.
- `regular(`L`, `Q`, `S`, `M`, `$Q0$`, `F`)`: Given a finite automaton (DFA or NFA) of Q states numbered 1, 2, ..., Q with input $1..S$, transition matrix M, initial state $Q0$ ($1 \leq Q0 \leq Q$), and a list of accepting states F, this constraint is true if the list L is accepted by the automaton. The transition matrix M represents a mapping from $1..Q \times 1..S$ to $0..Q$, where 0 denotes the *error* state. For a DFA, every entry in M is an integer representing a state, and for an NFA, each entry can be a list of integers representing possible next states.
- `scalar_product(`A`, `X`, `$Product$`)`: The scalar product of A and X is $Product$, where A and X are lists or arrays of integer-domain variables, and $Product$ is an integer-domain variable. A and X must have the same length.
- `scalar_product(`A`, `X`, `Rel`, `$Product$`)`: The scalar product of A and X has the relation Rel with $Product$, where Rel is one of the following operators: `#=`, `#!=`, `#>=`, `#>`, `#=<` (`#<=`), and `#<`.
- `scc(`Vs`, `Es`)`: This constraint ensures that the *undirected* graph represented by Vs and Es is connected, where Vs is a list of pairs of the form $\{V, B\}$, and Es is a list of triplets of the form $\{V_1, V_2, B\}$. A pair $\{V, B\}$ in Vs, where V is a ground term and B is a Boolean variable, denotes that V is in the graph if and only if $B = 1$. A triplet $\{V_1, V_2, B\}$ denotes that V_1 is connected with V_2 by an edge in the graph if and only if $B = 1$ and both V_1 and V_2 are in the graph. Note that the graph to be constructed is assumed to be undirected. If there exists a triplet $\{V_1, V_2, B\}$ in Es, then the triplet $\{V_2, V_1, B\}$ will be added to Es if it is not specified.

- `scc(`Vs`, `Es`, `K`)`: This constraint is the same as `scc(`Vs`, `Es`)`, except that it also constrains the number of vertices in the graph to be K.

- `scc_d(`Vs`, `Es`)`: This constraint ensures that the *directed* graph represented by Vs and Es is strongly connected, where Vs and Es are the same as those in `scc(`Vs`, `Es`)`, except that the graph is directed.

- `scc_d(`Vs`, `Es`, `K`)`: This constraint is the same as `scc_d(`Vs`, `Es`)`, except that it also constrains the number of vertices in the graph to be K.

- `scc_grid(`A`)`: This constraint ensures that the grid graph represented by A, which is a two-dimensional array of Boolean variables, forms a connected undirected graph. In a grid graph, each cell is directly connected horizontally and vertically, but not diagonally, to its neighbors. Only cells labeled 1 are considered as vertices of the graph. This constraint is implemented as follows by using `scc`:

```
scc_grid(A) =>
    NRows = len(A),
    NCols = len(A[1]),
    Vs = [{(R,C), A[R,C]} :
                R in 1..NRows,
                C in 1..NCols],
    Es = [{(R,C), (R1,C1), _} :
                R in 1..NRows,
                C in 1..NCols,
                (R1,C1) in neibs(A,NRows,NCols,R,C),
                (R,C) @< (R1,C1)],
    scc(Vs, Es).

neibs(A, NRows, NCols, R, C) =
    [(R1,C1) : (R1,C1) in [(R-1,C), (R+1,C),
                           (R,C-1), (R,C+1)],
                R1 >= 1, R1 =< NRows,
                C1 >= 1, C1 =< NCols,
                A[R1,C1] !== 0].
```

  Note that there is an edge between each pair of neighboring cells in the resulting graph as long as the cells are in the graph.

- `scc_grid(`A`, `K`)`: This constraint is the same as `scc_grid(`A`)`, except that it also constrains the number of vertices in the graph to be K.

- `serialized(`$Starts$`, `$Durations$`)`: This constraint describes a set of non-overlapping tasks, where $Starts$ and $Durations$ are lists of integer-domain variables, and the lists have the same length. Let Os be a list of 1s that has the same length as $Starts$. This constraint is equivalent to `cumulative(`$Starts$`, `$Durations$`, `Os`, 1)`.

- `subcircuit(`$FDVars$`)`: This constraint is the same as `circuit(`$FDVars$`)`, except that not all of the vertices are required to be in the circuit. If the ith element of $FDVars$ is i, then the vertex i is not part of the circuit.

- `subcircuit_grid(`A`)`: This constraint ensures that the grid graph represented by A, which is a two-dimensional array of Boolean (0/1) variables, forms a Hamiltonian cycle. In a grid graph, each cell is directly connected horizontally and vertically, but not diagonally, to its neighbors. Only non-zero cells are considered as vertices of the graph.

- `subcircuit_grid(`A`, `K`)`: This constraint is the same as `subcircuit_grid(`A`)`, except that it also constrains the number of vertices in the graph to be K.

- `tree(`Vs`, `Es`)`: This constraint ensures that the *undirected* graph represented by Vs and Es is a tree, where Vs is a list of pairs of the form $\{V, B\}$, and Es is a list of triplets of the form $\{V_1, V_2, B\}$. A pair $\{V, B\}$ in Vs, where V is a ground term and B is a Boolean variable, denotes that V is in the tree if and only if $B = 1$. A triplet $\{V_1, V_2, B\}$ denotes that V_1 is connected with V_2 by an edge in the tree if and only if $B = 1$ and both V_1 and V_2 are in the tree. Note that the graph to be constructed is assumed to be undirected. If there exists a triplet $\{V_1, V_2, B\}$ in Es, then the triplet $\{V_2, V_1, B\}$ will be added to Es if it is not specified.

- `tree(`Vs`, `Es`, `K`)`: This constraint is the same as `tree(`Vs`, `Es`)`, except that it also constrains the number of vertices in the tree to be K.

13.6 Bit-Vector Constraints (sat only)

The `sat` module supports bit-vector constraints, which enhance modeling capabilities, improve solving performance, and support large integers that exceed the bounds of domain values. A bit vector is an array representing the binary form of an unsigned integer. Let V be a bit vector of length N. The bit $V[1]$ corresponds to the least significant bit, while $V[N]$ represents the most significant bit. In a bit-vector constraint, a bit-vector argument may also be a binary variable or a non-negative integer. If the argument is a binary variable B, Picat converts it to $\{B\}$. If the argument is a non-negative integer, Picat automatically converts it to its binary representation.

- `bv_add(`A`, `B`, `C`)`: Forces bit-vector C to be the sum of bit-vectors A and B.

- `bv_and(`A`, `B`, `C`)`: Forces bit-vector C to be the bitwise *and* of bit-vectors A and B.

- `bv_div(`A`, `B`, `C`)`: Forces bit-vector C to be the quotient of A divided by B ($C = A \; div \; B$).

- `bv_drop(`A`, `N`) = `Res: Returns a bit vector containing the remaining bits of bit-vector A after dropping the lowest N bits.

- `bv_eq(`A`, `B`)`: Forces equality between bit-vector A and bit-vector B.

- `bv_ge(`A`, `B`)`: Forces bit-vector A to be greater than or equal to bit-vector B ($A \geq B$).

- `bv_gt(`A`, `B`)`: Forces bit-vector A to be greater than bit-vector B ($A > B$).

- `bv_le(`A`, `B`)`: Forces bit-vector B to be greater than or equal to bit-vector A ($B \geq A$). This is the same as `bv_ge(`B`, `A`)`.

- `bv_lt(`A`, `B`)`: Forces bit-vector B to be greater than bit-vector A ($B > A$). This is the same as `bv_gt(`B`, `A`)`.

- `bv_mod(`A`, `B`, `C`)`: Forces bit-vector C to be the remainder of A divided by B ($C = A \; mod \; B$).

- `bv_mul(`A`, `B`, `C`)`: Forces bit-vector C to be the product of multiplying A and B.
- `bv_neq(`A`, `B`)`: Forces inequality between bit-vector A and bit-vector B ($A \neq B$).
- `bv_or(`A`, `B`, `C`)`: Forces bit-vector C to be the bitwise *or* of bit-vectors A and B.
- `bv_pow(`A`, `B`, `C`)`: Forces bit-vector C to be A raised to the power of B ($C = A^B$).
- `bv_sum(`As`, `B`)`: Constrains the sum of the terms in the list As to equal the bit vector B. Each term in As is either a bit vector or a pair `(`Coe`, `V`)` representing $Coe \cdot V$, where Coe is a non-negative integer and V is an integer, a binary variable, or a bit vector.
- `bv_take(`A`, `N`)` = Res: Returns a bit vector containing the lowest N bits of bit-vector A.
- `bv_to_int(`A`)` = Res: Returns an integer represented by bit-vector A. This function throws an exception if A is not ground.
- `bv_xor(`A`, `B`, `C`)`: Forces bit-vector C to be the bitwise *xor* of bit-vectors A and B.
- `int_to_bv(`C`)` = Res: Returns the binary representation of integer C as a bit vector.
- `new_bv(`N`)` = Res: Returns a bit vector of length N.

13.7 Solver Invocation

- `solve(`$Opts$`, `$Vars$`)`: This predicate calls the imported solver to label the variables $Vars$ with values, where $Opts$ is a list of options for the solver. The options will be detailed below. This predicate can backtrack in order to find multiple solutions. The `cp` module allows *incremental labeling* of variables, and some variables that occur in constraints but are not passed to `solve` may remain uninstantiated after a call to `solve`. The user is responsible for having all the variables that need to be instantiated passed to `solve`. In contrast, the `sat`, `mip`, and `smt` modules do not support incremental labeling of variables.
- `solve(`$Vars$`)`: This predicate is the same as `solve([], `$Vars$`)`.
- `indomain(`Var`)`: This predicate is only accepted by `cp`. It is the same as `solve([], [`Var`])`.
- `indomain_down(`Var`)`: This predicate is only accepted by `cp`. It is the same as `solve([down], [`Var`])`.
- `solve_all(`$Opts$`, `$Vars$`)` = $List$: This function returns all the solutions that satisfy the constraints.
- `solve_all(`$Vars$`)` = $List$: This function is the same as `solve_all([], `$Vars$`)`.
- `solve_suspended(`$Opts$`)`: After `solve(`$Vars$`)` has successfully labeled $Vars$, some constraints may remain suspended and not completely checked because not all of the decision variables are included in $Vars$. The `solve_suspended(`$Opts$`)` predicate labels all remaining variables in the suspended constraints. This predicate is only provided by the `cp` module.
- `solve_suspended`: This predicate is the same as `solve_suspended([])`.

13.7.1 Common Solving Options

The following options are shared by the solvers, subject to the restrictions noted below.[5]

- `$limit(`N`)`: Search for up to N solutions.
- `$max(`Var`)`: Maximize the variable Var.
- `$min(`Var`)`: Minimize the variable Var.
- `$report(`$Call$`)`: Execute $Call$ each time a better answer is found while searching for an optimal answer. This option cannot be used if the `mip` module is used.

13.7.2 Solving Options for `cp`

The `cp` module also accepts the following options:

- `backward`: The list of variables is reversed first.
- `constr`: Variables are first ordered by the number of attached constraints.
- `degree`: Variables are first ordered by degree, i.e., the number of connected variables.
- `down`: Values are assigned to variables from the largest to the smallest.
- `ff`: The first-fail principle is used: the leftmost variable with the smallest domain is selected.
- `ffc`: The same as with the two options: `ff` and `constr`.
- `ffd`: The same as with the two options: `ff` and `degree`.
- `forward`: Choose variables in the given order, from left to right.
- `inout`: The variables are reordered in an inside-out fashion. For example, the variable list `[X1,X2,X3,X4,X5]` is rearranged into the list `[X3,X2,X4,X1,X5]`.
- `$label(`$CallName$`)`: This option informs the CP solver that once a variable V is selected, the user-defined call $CallName$`(`V`)` is used to label V, where $CallName$ must be defined in the same module, an imported module, or the global module.
- `leftmost`: The same as `forward`.
- `max`: First, select a variable whose domain has the largest upper bound, breaking ties by selecting a variable with the smallest domain.
- `min`: First, select a variable whose domain has the smallest lower bound, breaking ties by selecting a variable with the smallest domain.
- `rand`: Both variables and values are randomly selected when labeling.
- `rand_var`: Variables are randomly selected when labeling.
- `rand_val`: Values are randomly selected when labeling.
- `reverse_split`: Bisect the variable's domain, excluding the lower half first.
- `split`: Bisect the variable's domain, excluding the upper half first.
- `updown`: Values are assigned to variables from the values that are nearest to the middle of the domain.

[5] Note that the preceding dollar sign indicates that it is followed by a term, not a function call.

13.7.3 Solving Options for `sat`

- `dump`: Dump the CNF code to `stdout`.
- `$dump(`$File$`)`: Dump the CNF code to $File$.
- `$ext(`$Command$`)`: Save the CNF code to a temporary file, then solve it using the external solver specified by the OS command $Command$. For example, `ext("cadical -quiet")`.
- `seq`: Use sequential search to find an optimal answer.
- `split`: Use binary search to find an optimal answer (default).
- `$nvars(`$NVars$`)`: The number of variables in the CNF code is $NVars$.
- `$ncls(`$NCls$`)`: The number of clauses in the CNF code is $NCls$.

13.7.4 Solving Options for `mip`

- `cbc`: Instruct Picat to use the Cbc MIP solver. Picat uses the following command to call the Cbc solver:

  `cbc` $TmpFile$ `-solve -solution` $SolFile$

  where $SolFile$ is a file for the solution, and $TmpFile$ is a file that stores the CPLEX-format constraints. Picat throws `existence_error` if the command `cbc` is not available.

- `dump`: Dump the constraints in CPLEX format to `stdout`.
- `$dump(`$File$`)`: Dump the CPLEX format to $File$.
- `glpk`: Instruct Picat to use the GLPK MIP solver. Picat uses the following command to call the GLPK solver:

  `glpsol -lp -o` $SolFile$ $TmpFile$

  where $SolFile$ is a solution file, and $TmpFile$ is a file that stores the CPLEX-format constraints. Picat throws `existence_error` if the command `glpsol` is not available.

- `gurobi`: Instruct Picat to use the Gurobi MIP solver. Picat uses the following command to call the Gurobi solver:

  `gurobi_cl ResultFile=`$SolFile$ $TmpFile$

  where $SolFile$ is a file for the solution, and $TmpFile$ is a file that stores the CPLEX-format constraints. Picat throws `existence_error` if the command `gurobi_cl` is not available.

- `scip`: Instruct Picat to use the SCIP MIP solver. Picat uses the following command to call the SCIP solver:

  `scip -q -c 'read` $TmpFile$`' -c optimize`
  `-c 'write solution` $SolFileStr$`' -c quit`

  where $SolFile$ is a file for the solution, and $TmpFile$ is a file that stores the CPLEX-format constraints. Picat throws `existence_error` if the command `scip` is not available.

13.7.5 Solving Options for `smt`

- `cvc4`: Instruct Picat to use the CVC4 SMT solver. Picat uses the following command to call the CVC4 solver:

  `cvc4` $TmpFile > SolFile$

  where $TmpFile$ is a file that stores the SMT-LIB2-format constraints, and $SolFile$ is a solution file. Picat throws `existence_error` if the command `cvc4` is not available in the path.

- `dump`: Dump the constraints in SMT-LIB2 format to `stdout`.

- `$dump(`$File$`)`: Dump the SMT-LIB2 format to $File$.

- `$logic(`$Logic$`)`: Instruct the SMT solver to use $Logic$ when solving, where $Logic$ must be an atom or a string, and the specified logic must be available in the SMT solver. The default logic for z3 is "LIA", and the default logic for CVC4 is "NIA".

- `$tmp(`$File$`)`: Dump the SMT-LIB2 format to $File$ rather than the default file "__tmp<pid>.smt2", before calling the smt solver. The name $File$ must be a string or an atom that has the extension name ".smt2". When this file name is specified, the smt solver will save the solution into a file name that has the same main name as $File$ but the extension name ".sol".

- `z3`: Instruct Picat to use the z3 SMT solver. When no SMT solver is specified, Picat first searches for the command `z3`, and when `z3` cannot be found it continues to search for the command `cvc4`.

13.8 Posting and Solving Multiple Constraint Problems

To describe and solve multiple constraint satisfaction problems in Picat, model each problem by creating its variables, specifying their domains, posting its constraints, and calling `solve` to find a satisfying assignment. This process can be repeated using either a failure-driven loop or a normal loop.

A failure-driven loop deliberately fails after processing each problem, causing backtracking to undo its variable bindings and automatically clear the constraint store before the next problem.

A normal loop, such as a `foreach` loop, instead requires an explicit call to `reset_store` between problems to clear the constraint store. Its advantage is that results can be retained across iterations, making it convenient to collect, compare, or aggregate solutions from multiple problems.

- `reset_store`: Reset the constraint store. Memory occupied by constraints from the previous problem may not be reclaimed by garbage collection.

Thus, a failure-driven loop offers automatic cleanup, while a normal loop offers straightforward access to results across problems.

Chapter 14

The nn (Neural Networks) Module

The nn module provides a high-level interface between Picat and the FANN[1] neural networks library, which implements feedforward neural networks.[2] A *feedforward neural network* consists of *neurons* organized in layers from an input layer to an output layer, possibly with a number of hidden layers. A feedforward network represents a function from input to output. Neurons in a layer (except for the input layer) are connected to neurons in the previous layer. The connections have weights. The neurons in the input layer receive the input. The information is propagated forward through the layers until it reaches the output layer, where the output is returned. The information that a neuron receives is determined by the connected predecessor neurons, the weights of the connections, and an *activation function*. The connection weights of a neural network are normally adjusted through training on a given set of input-output pairs, called *training data*. Once a neural network is trained, it can be used to predict the output for a given input.

The following gives an example program which creates a neural network for the xor function, and trains it on a set of data stored in a file:

```
import nn.

main =>
    NN = new_nn({2,3,1}),
    nn_train(NN, "xor.data"),
    nn_save(NN, "xor.net"),
    nn_destroy_all.
```

The function new_nn({2,3,1}) returns a neural network with three layers, where the input layer has 2 neurons, the hidden layer has 3 neurons, and the output layer has 1 neuron. The program does not specify any activation functions used between layers, which means that the default activation function, which is sym_sigmoid, will be used. The predicate nn_train(NN, "xor.data") trains the neural network with the training data stored in the file "xor.data". The user is able to specify an algorithm to be used in the training and several parameters that affect the behavior of the algorithm, such as the maximum number of iterations (called *epochs*), the learning rate, and the error function. This example does not specify a training algorithm or any of the training parameters, which means that the default algorithm, which is rprop, is used with the default setting. The predicate nn_save saves the trained neural network into a file named "xor.net". The predicate nn_destroy_all clears the neural network and the internal data structures used during training.

The text file "xor.data" contains the following training data:

[1] http://leenissen.dk/fann/wp/

[2] The Picat-FANN interface was implemented by Sanders Hernandez.

```
4 2 1
-1 -1
-1
-1 1
1
1 -1
1
1 1
-1
```

The three integers in the first line state, respectively, that the number of input-output pairs is 4, the number of input values is 2, and the number of output values is 1. The remaining lines give the input-output pairs.

The following program performs the same task as the above program, except that it trains the neural network with internal data:

```
import nn.

main =>
    NN = new_nn({2,3,1}),
    nn_train(NN, [({-1,-1}, -1),
                  ({-1,1}, 1),
                  ({1,-1}, 1),
                  ({1,1}, -1)]),
    nn_save(NN, "xor.net"),
    nn_destroy_all.
```

The predicate `nn_train` is overloaded. When the second argument is a file name, Picat reads training data from the file. Otherwise, Picat expects a collection (a list or an array) of input-output pairs.

The following example program illustrates how to use a trained network:

```
import nn.

main =>
    NN = nn_load("xor.net"),
    printf("xor(-1, 1) = %w\n", nn_run(NN, {-1,1})),
    nn_destroy_all.
```

The function `nn_load` loads a neural network. The function `nn_run` uses the network to predict the output for an input.

14.1 Creating, Printing, and Destroying Neural Networks

- `new_nn(`$Layers$`) = `NN: This function creates a fully connected neural network of the structure $Layers$, which is a collection of positive integers indicating the number of neurons in each layer.
- `new_standard_nn(`$Layers$`) = `NN: This function is the same as `new_nn(`$Layers$`)`.

- `new_sparse_nn(`$Layers$`,` $Rate$`)` = NN: This function creates a sparse neural network that has the structure $Layers$ and the connection rate $Rate$. The connection rate determines the sparseness of the network, with 1 indicating that the network is fully connected, and 0 indicating that the network is not connected at all.
- `new_sparse_nn(`$Layers$`)` = NN: This function creates a sparse neural network that has the structure $Layers$ and the connection rate 0.5.
- `nn_print(`NN`)`: This predicate prints the attributes of the neural network NN, including the connections, the weights, the activation functions, and some other parameters.
- `nn_destroy(`NN`)`: This predicate destroys the neural network NN.
- `nn_destroy_all`: This predicate destroys all the neural networks and the internal data structures.

14.2 Activation Functions

An activation function for a neuron determines how information is propagated to it from its predecessor neurons. When a new neural network is created, it uses the default activation function `sym_sigmoid` for all of its non-input neurons. The following predicates can be utilized to set activation functions.

- `nn_set_activation_function_layer(`NN`,` $Func$`,` $Layer$`)`: This predicate sets the activation function to $Func$ for all the neurons in layer $Layer$ in the neural network NN. Let the number of layers in NN be n. Since the first layer is numbered 1, $Layer$ must satisfy $2 \leq Layer \leq n$. The following activation functions are available:
  - `linear`
  - `threshold`
  - `sym_threshold`: symmetric threshold.
  - `sigmoid`
  - `step_sigmoid`: stepped sigmoid
  - `sym_sigmoid`: symmetric sigmoid
  - `elliot`: an alternative to sigmoid
  - `sym_elliot`: symmetric elliot
  - `gaussian`
  - `sym_gaussian`: symmetric Gaussian
  - `linear_piece`
  - `sym_linear_piece`: symmetric linear piece
  - `sin`
  - `sym_sin`: symmetric sin
  - `cos`
  - `sym_cos`: symmetric cos

  The default activation function is `sym_sigmoid`. Each of these functions has a corresponding name in FANN. Please refer to the FANN documentation for a more detailed description of these functions.

- `nn_set_activation_function_hidden(`NN`,` $Func$`)`: This predicate sets the activation function to $Func$ for all of the hidden layers in the neural network NN.
- `nn_set_activation_function_output(`NN`,` $Func$`)`: This predicate sets the activation function to $Func$ for the output layer in the neural network NN.
- `nn_set_activation_steepness_layer(`NN`,` $Steepness$`,` $Layer$`)`: This predicate sets the activation steepness for all of the neurons in $Layer$, where $-1 \leq Steepness \leq 1$ and $2 \leq Layer \leq n$ (n is the number of layers in NN). A higher steepness value means more aggressive training. The default steepness is 0.5.
- `nn_set_activation_steepness_hidden(`NN`,` $Steepness$`)`: This predicate sets the activation steepness to $Steepness$ for all of the hidden layers in the neural network NN.
- `nn_set_activation_steepness_output(`NN`,` $Steepness$`)`: This predicate sets the activation steepness to $Steepness$ for the output layer in the neural network NN.

14.3 Training Data

A training dataset can be supplied to FANN either through a text file or a Picat collection. A training dataset file must have the following format:

```
num_train_data num_input num_output
input_data separated by spaces
output_data separated by spaces
.
.
.
input_data separated by spaces
output_data separated by spaces
```

A training dataset stored in a Picat collection must be either a list or an array of input-output pairs. An input-output pair has the form `(`$Input$`,` $Output$`)`, where $Input$ is an array of numbers or a single number, and so is $Output$.

- `nn_train_data_size(`$Data$`) =` $Size$: This function returns the number of input-output pairs in the dataset $Data$, which is either a file name or a Picat collection.
- `nn_train_data_get(`$Data$`,` I`) =` $Pair$: This function returns the Ith pair in the dataset $Data$. Notice that while this function takes linear time when $Data$ is a list, it takes constant time when $Data$ is a file name or an array.[3]
- `nn_train_data_load(`$File$`) =` $Data$: This function loads a dataset stored in $File$ into a Picat array, and returns the array.
- `nn_train_data_save(`$Data$`,` $File$`)`: This predicate saves a dataset in a Picat collection $Data$ into a file named $File$.

[3]The data is loaded into an internal array when the dataset is accessed the first time.

14.4 Training Neural Networks

- `nn_train(`NN`,` $Data$`,` $Opts$`)`: This predicate trains the neural network NN using the dataset $Data$ under the control of training options $Opts$. The following training options are supported:
  - `maxep(`X`)`: X is the maximum number of epochs for training.
  - `report(`X`)`: report progress every X epochs.
  - `derror(`X`)`: X is the desired error in training.
  - `train_func(`X`)`: X is the training algorithm to be used, which is one of the following: `batch`, `inc`, `qprop` (quick prop), `rprop`, and `sprop` (sarprop).
  - `lrate(`X`)`: X is the learning rate.
  - `momentum(`X`)`: X is the learning momentum.
  - `error_func(`X`)`: X is the error function to be used, which is either `linear` or `tanh`.
  - `stop_func(`X`)`: X is the stop function to be used, which is either `bit` or `mse`.
  - `bfl(`X`)`: X is the bit fail limit.
  - `qp_decay(`X`)`: X is the `qprop` decay.
  - `qp_mu(`X`)` : X is the `qprop` mu factor.
  - `rp_increase(`X`)`: X is the `rprop` increase factor.
  - `rp_decrease(`X`)`: X is the `rprop` decrease factor.
  - `rp_deltamin(`X`)`: X is the `rprop` delta min.
  - `rp_deltamax(`X`)`: X is the `rprop` delta max.
  - `rp_deltazero(`X`)`: X is the `rprop` delta zero.
  - `sp_weight(`X`)`: X is the `sarprop` weight decay.
  - `sp_thresh(`X`)`: X is the `sarprop` step error threshold factor.
  - `sp_shift(`X`)`: X is the `sarprop` step error shift.
  - `sp_temp(`X`)`: X is the `sarprop` temperature.
  - `scale(`$InMin$`,` $InMax$`,` $OutMin$`,` $OutMax$`)`: use training data scaling.
  - `inscale(`$InMin$`,` $InMax$`)`: use input data scaling.
  - `outscale(`$OutMin$`,` $OutMax$`)`: use output data scaling.
- `nn_train(`NN`,` $Data$`)`: This predicate is the same as `nn_train(`NN`,` $Data$`, []).`

14.5 Saving and Loading Neural Networks

- `nn_save(`NN`,` $File$`)`: This predicate saves the neural network NN to a file named $File$.
- `nn_load(`$File$`)` $=$ NN: This function creates a neural network from a FANN neural network file named $File$.

14.6 Running Neural Networks

- `nn_run(`NN`,` $Input$`,` $Opts$`)` = $Output$: This function predicts the output for a given input using the trained neural network NN, where $Input$ is an array of numbers or a single number, and $Opts$ is a list of testing options. The supported testing options include:
  - `scaleIn(`X`)`: X indicates whether or not the input is scaled, with -1 meaning descale, 1 meaning scale, and 0 meaning nothing.
  - `scaleOut(`X`)`: X indicates whether or not the output is scaled, with -1 meaning descale, 1 meaning scale, and 0 meaning nothing.
  - `resetMSE`: resets the current mean squared error of the network.
- `nn_run(`NN`,` $Input$`)` = $Output$: This function is the same as `nn_run(`NN`,` $Input$`, []`.

Chapter 15

The os Module

Picat has an os module for manipulating files and directories. In order to use any of the functions or predicates, users must import the module.

15.1 The $Path$ Parameter

Many of the functions and predicates in this module have a $Path$ parameter. This parameter is a string or an atom, representing the path of a file or directory. This path can be an absolute path, from the system's root directory, or a path relative to the current working directory. Different systems use different separator characters to separate directories at different levels of the directory hierarchy. For example, Windows uses '\' and Unix uses '/'. The following function outputs a single character, representing the character that the current system uses as a file separator.

- `separator() =` $Char$

15.2 Directories

The os module includes functions for reading and modifying directories. The following example shows how to list all of the files in a directory tree, using a depth-first directory traversal.

Example

```
import os.

traverse(Dir), directory(Dir) =>
   List = listdir(Dir),
   printf("Inside %s%n", Dir),
   foreach (File in List)
       printf("    %s%n", File)
   end,
   foreach (File in List, File != ".", File != "..")
       FullName = Dir ++ [separator()] ++ File,
       traverse(FullName)
   end.
traverse(_Dir) => true.
```

The following function can be used to read the contents of a directory:

- `listdir(`$Path$`) = `$List$: This function returns a list of all of the files and directories that are contained inside the directory specified by $Path$. If $Path$ is not a directory, then an error is thrown. The returned list contains strings, each of which is the name of a file or directory.

The above example also uses the `directory` predicate, which will be discussed in Section 15.4.

15.2.1 The Current Working Directory

The `os` module includes built-ins for obtaining the program's current working directory, inspecting files in the directory, and changing the directory.

- `cd(`$Path$`)`: This predicate changes the current working directory to the directory specified by $Path$. It fails if the directory does not exist, if the program lacks permission to it, or if $Path$ does not refer to a directory.
- `chdir(`$Path$`)`: This predicate is the same as `cd(`$Path$`)`.
- `cwd() = `$Path$: This function returns the current working directory.
- `dir`: This predicate lists the files in the current working directory.
- `ls`: This predicate is the same as `dir`.
- `pwd() = `$Path$: This function is the same as `cwd()`.

15.3 Modifying Files and Directories

15.3.1 Creation

The `os` module contains a number of predicates for creating new files and directories:

- `mkdir(`$Path$`)`: This predicate creates a new directory at location $Path$. The directory will be created with a default permission list of `[rwu, rwg, ro]`. If the program does not have permission to write to the parent directory of $Path$, this predicate will throw an error. An error will also occur if the parent directory does not exist.
- `rename(`Old`, `New`)`: This renames a file or a directory from Old to New. This predicate will throw an error if Old does not exist. An error will also occur if the program does not have permission to write to Old or New.
- `cp(`$FromPath$`, `$ToPath$`)`: This copies a file from $FromPath$ to $ToPath$. This predicate will throw an error if $FromPath$ does not exist or $FromPath$ is a directory. An error will also occur if the program does not have permission to read from $FromPath$, or if it does not have permission to write to $ToPath$.

15.3.2 Deletion

The `os` module contains a number of predicates for deleting files and directories.

- `rm(`$Path$`)`: This deletes a file. An error will be thrown if the file does not exist, if the program does not have permission to delete the file, or if $Path$ refers to a directory, a hard link, a symbolic link, or a special file type.
- `rmdir(`$Path$`)`: This deletes a directory. An error will be thrown if the directory does not exist, if the program lacks permission to delete it, if it is not empty, or if $Path$ does not refer to a directory.

15.4 Obtaining Information about Files

The `os` module contains a number of functions that retrieve file status information, and predicates that test the type of a file. Unless specified otherwise, these predicates will all throw an error if the program does not have permission to read from $Path$.

- `readable(`$Path$`)`: Is the program allowed to read from the file?
- `writable(`$Path$`)`: Is the program allowed to write to the file?
- `executable(`$Path$`)`: Is the program allowed to execute the file?
- `size(`$Path$`) =` Int: If $Path$ is not a symbolic link, then this function returns the number of bytes contained in the file to which $Path$ refers. If $Path$ is a symbolic link, then this function returns the path size of the symbolic link. Because the function `size/1` is defined in the `basic` module for returning the size of a map, this function requires an explicit module qualifier `os.size(`$Path$`)`.
- `file_base_name(`$Path$`) =` $String$: This function returns a string containing the base name of $Path$. For example, the base name of `"a/b/c.txt"` is `"c.txt"`.
- `file_directory_name(`$Path$`) =` $String$: This function returns a string containing the path of the directory that contains $Path$. For example, the directory name of `"a/b/c.txt"` is `"a/b"`.
- `exists(`$Path$`)`: Is $Path$ an existing file or directory?
- `file(`$Path$`)`: Does $Path$ refer to a regular file? This predicate is true if $Path$ is neither a directory nor a special file, such as a socket or a pipe.
- `file_exists(`$Path$`)`: This tests whether $Path$ exists, and, if it exists, whether $Path$ refers to a regular file.
- `directory(`$Path$`)`: Does $Path$ refer to a directory?

The following example shows how to use a few of the predicates.

Example

```
import os.

test_file(Path) =>
    if (not exists(Path))
        printf("%s does not exist %n", Path)
    elseif (directory(Path))
        println("Directory")
    elseif (file(Path))
        println("File")
    else
        println("Unknown")
    end.
```

15.5 Environment Variables

- `env_exists(`$Name$`)`: This predicate succeeds if $Name$ is an environment variable in the system.
- `getenv(`$Name$`) = `$String$: This function returns the value of the environment variable $Name$ as a string. This function will throw an error if the environment variable $Name$ does not exist.

Appendix A

The `math` Module

Picat provides a `math` module, which has common mathematical constants and functions. The `math` module is imported by default.

A.1 Constants

The `math` module provides two constants.

- `e = 2.71828182845904523536`
- `pi = 3.14159265358979323846`

A.2 Functions

The `math` module contains mathematical functions that serve a number of different purposes. Note that the arguments must all be numbers. If the arguments are not numbers, then Picat will throw an error.

A.2.1 Sign and Absolute Value

The following functions deal with the positivity and negativity of numbers.

- `sign(`X`) =` Val: This function returns 1, 0, or -1 according to whether X is positive, zero, or negative.
- `abs(`X`) =` Val: This function returns the absolute value of X. If $X \geq 0$, then this function returns X. Otherwise, this function returns $-X$.

Example

```
Picat> Val1 = sign(3), Val2 = sign(-3), Val3 = sign(0)
Val1 = 1
Val2 = -1
Val3 = 0
Picat> Val = abs(-3)
Val = 3
```

A.2.2 Rounding and Truncation

The `math` module includes the following functions for converting a real number into the integers that are closest to the number.

- `ceiling(`X`) = `Val: This function returns the closest integer that is greater than or equal to X.
- `floor(`X`) = `Val: This function returns the closest integer that is less than or equal to X.
- `round(`X`) = `Val: This function returns the integer that is closest to X.
- `truncate(`X`) = `Val: This function removes the fractional part from a real number.
- `modf(`X`) = (`$IntVal$`, `$FractVal$`)`: This function splits a real number into its integer part and its fractional part.

Example

```
Picat> Val1 = ceiling(-3.2), Val2 = ceiling(3)
Val1 = -3
Val2 = 3
Picat> Val1 = floor(-3.2), Val2 = floor(3)
Val1 = -4
Val2 = 3
Picat> Val1 = round(-3.2), Val2 = round(-3.5), Val3 = round(3.5)
Val1 = -3
Val2 = -4
Val3 = 4
Picat> Val1 = truncate(-3.2), Val2 = truncate(3)
Val1 = -3
Val2 = 3
Picat> IF = modf(3.2)
IF = (3.0 , 0.2)
```

A.2.3 Exponents, Roots, and Logarithms

The following functions provide exponentiation, root, and logarithmic functions. Note that, in the logarithmic functions, if $X \leq 0$, then an error is thrown.

- `pow(`X`, `Y`) = `Val: This function returns X^Y. It does the same thing as $X * *Y$.
- `pow_mod(`X`, `Y`, `Z`) = `Val: This function returns X^Y `mod` Z. All of the arguments must be integers, and Y must not be negative.
- `exp(`X`) = `Val: This function returns e^X.
- `sqrt(`X`) = `Val: This function returns the square root of X. Note that the `math` module does not support imaginary numbers. Therefore, if $X < 0$, this function throws an error.
- `log(`X`) = `Val: This function returns $log_e(X)$.
- `log10(`X`) = `Val: This function returns $log_{10}(X)$.

- `log2(`X`) = `Val: This function returns $log_2(X)$.
- `log(`B`, `X`) = `Val: This function returns $log_B(X)$.

Example

```
Picat> P1 = pow(2, 5), P2 = exp(2)
P1 = 32
P2 = 7.38906
Picat> S = sqrt(1)
S = 1.0
Picat> E = log(7), T = log10(7), T2 = log2(7), B = log(7, 7)
E = 1.94591
T = 0.845098
T2 = 2.80735
B = 1.0
```

A.2.4 Converting Between Degrees and Radians

The `math` module has two functions to convert between degrees and radians.

- `to_radians(`$Degree$`) = `$Radian$: This function converts from degrees to radians.
- `to_degrees(`$Radian$`) = `$Degree$: This function converts from radians to degrees.

Example

```
Picat> R = to_radians(180)
R = 3.14159
Picat> D = to_degrees(pi())
D = 180.0
```

A.2.5 Trigonometric Functions

The `math` module provides the following trigonometric functions.

- `sin(`X`) = `Val: This function returns the sine of X, where X is given in radians.
- `cos(`X`) = `Val: This function returns the cosine of X, where X is given in radians.
- `tan(`X`) = `Val: This function returns the tangent of X, where X is given in radians.
- `sec(`X`) = `Val: This function returns the secant of X, where X is given in radians. If `cos(`X`)` is 0, then this function throws an error.
- `csc(`X`) = `Val: This function returns the cosecant of X, where X is given in radians. If `sin(`X`)` is 0, then this function throws an error.
- `cot(`X`) = `Val: This function returns the cotangent of X, where X is given in radians. If `tan(`X`)` is 0, then this function throws an error.
- `asin(`X`) = `Val: This function returns the arc sine of X, in radians. The returned value is in the range [`-pi / 2`, `pi / 2`]. X must be in the range $[-1, 1]$; otherwise, this function throws an error.

- `acos(`X`) = `Val: This function returns the arc cosine of X, in radians. The returned value is in the range [0, `pi`]. X must be in the range $[-1, 1]$; otherwise, this function throws an error.
- `atan(`X`) = `Val: This function returns the arc tangent of X, in radians. The returned value is in the range [`-pi / 2`, `pi / 2`].
- `atan2(`Y`, `X`) = `Val: This function returns the arc tangent of Y / X, in radians. X and Y are coordinates. The returned value is in the range [`-pi`, `pi`].
- `asec(`X`) = `Val: This function returns the arc secant of X, in radians. The returned value is in the range [0, `pi`]. X must be in the range $(-\infty, -1]$ or $[1, \infty)$; otherwise, this function throws an error.
- `acsc(`X`) = `Val: This function returns the arc cosecant of X, in radians. The returned value is in the range [`-pi / 2`, `pi / 2`]. X must be in the range $(-\infty, -1]$ or $[1, \infty)$; otherwise, this function throws an error.
- `acot(`X`) = `Val: This function returns the arc cotangent of X, in radians. The returned value is in the range [`-pi / 2`, `pi / 2`].

Example

```
Picat> S = sin(pi()), C = cos(pi()), T = tan(pi())
S = 0.0
C = -1.0
T = 0.0
Picat> S = asin(0), C = acos(0), T = atan(0), T2 = atan2(-10, 10)
S = 0.0
C = 1.5708
T = 0.0
T2 = -0.785398
Picat> S = sec(pi()/4), C = csc(pi()/4), T = cot(pi()/4)
S = 1.41421
C = 1.41421
T = 1.0
Picat> S = asec(2), C = acsc(2), T = acot(0)
S = 1.0472
C = 0.5236
T = 1.5708
```

A.2.6 Hyperbolic Functions

The `math` module provides the following hyperbolic functions.

- `sinh(`X`) = `Val: This function returns the hyperbolic sine of X.
- `cosh(`X`) = `Val: This function returns the hyperbolic cosine of X.
- `tanh(`X`) = `Val: This function returns the hyperbolic tangent of X.
- `sech(`X`) = `Val: This function returns the hyperbolic secant of X.

- `csch(`X`) = `Val: This function returns the hyperbolic cosecant of X. If X is 0, then this function throws an error.
- `coth(`X`) = `Val: This function returns the hyperbolic cotangent of X. If X is 0, then this function throws an error.
- `asinh(`X`) = `Val: This function returns the arc hyperbolic sine of X.
- `acosh(`X`) = `Val: This function returns the arc hyperbolic cosine of X. If $X < 1$, then this function throws an error.
- `atanh(`X`) = `Val: This function returns the arc hyperbolic tangent of X. X must be in the range $(-1, 1)$; otherwise, this function throws an error.
- `asech(`X`) = `Val: This function returns the arc hyperbolic secant of X. X must be in the range $(0, 1]$; otherwise, this function throws an error.
- `acsch(`X`) = `Val: This function returns the arc hyperbolic cosecant of X. If X is 0, then this function throws an error.
- `acoth(`X`) = `Val: This function returns the arc hyperbolic cotangent of X. X must be in the range $(-\infty, -1)$ or $(1, \infty)$; otherwise, this function throws an error.

Example

```
Picat> S = sinh(pi()), C = cosh(pi()), T = tanh(pi())
S = 11.54874
C = 11.59195
T = 0.99627
Picat> S = sech(pi()/4), C = csch(pi()/4), T = coth(pi()/4)
S = 0.75494
C = 1.15118
T = 1.52487
Picat> S = asinh(0), C = acosh(1),  T = atanh(0)
S = 0.0
C = 0.0
T = 0.0
```

A.2.7 Random Numbers

The following functions provide access to a random number generator.

- `random() = `Val: This function returns a random integer.
- `random2() = `Val: This function returns a random integer. It uses an environment-dependent seed.
- `rand_max() = `Val: This function returns the maximum random integer.
- `random(`$Seed$`) = `Val: This function returns a random integer. At the same time, it changes the seed of the random number generator.
- `random(`Low`, `$High$`) = `Val: This function returns a random integer in the range $Low..High$.

- `frand()` = Val: This function returns a random real number between 0.0 and 1.0, inclusive.
- `frand(`Low`,` $High$`)` = Val: This function returns a random real number between Low and $High$, inclusive.

A.2.8 Other Built-ins

- `binom(`N`,` K`)` = Val: This function returns the binomial coefficient $\binom{N}{K}$.
- `even(`N`)`: This predicate is true if N is an even integer.
- `gcd(`A`,` B`)` = Val: This function returns the greatest common divisor of integer A and integer B.
- `odd(`N`)`: This predicate is true if N is an odd integer.
- `prime(`N`)`: This predicate is true if N is a prime number.
- `primes(`N`)` = $List$: This function returns a list of prime numbers that are less than or equal to N.

Appendix B

The `sys` Module

The `sys` module, which is imported by default, contains built-ins that are relevant to the Picat system. The built-ins in the `sys` module perform operations that include compiling programs, tracing execution, and displaying statistics and information about the Picat system.

B.1 Compiling and Loading Programs

The `sys` module includes a number of built-ins for compiling programs and loading them into memory.

- `compile(`$FileName$`)`: This predicate compiles the file $FileName$`.pi` and all of its dependent files without loading the generated bytecode files. The destination directory for the bytecode file is the same as the source file's directory. If the Picat interpreter does not have permission to write into the directory in which a source file resides, then this built-in throws an exception. If $FileName$`.pi` imports modules, then these module files are also compiled. The system searches for these module files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`.

- `compile_bp(`$FileName$`)`: This predicate translates the Picat file $FileName$`.pi` into a B-Prolog file $FileName$`.pl`. If the file is dependent on other Picat files, then those files are compiled using `compile/1`. The destination directory for the B-Prolog file is the same as the source file's directory. If the Picat interpreter does not have permission to write into the directory in which a source file resides, then this built-in throws an exception.

- `load(`$FileName$`)`: This predicate loads the bytecode file $FileName$`.qi` and all of its dependent bytecode files into the system for execution. For $FileName$, the system searches for a bytecode file in the directory specified by $FileName$ or the directories that are stored in the environment variable `PICATPATH`. For the dependent file names, the system searches for a bytecode file in the directory in which $FileName$`.qi` resides or the directories that are stored in the environment variable `PICATPATH`. If the bytecode file $FileName$`.qi` does not exist, but the source file $FileName$`.pi` exists, then this built-in compiles the source file and loads the bytecodes without creating a `qi` file. Note that, for the dependent files, if the bytecode file does not exist, but the source file exists, then the source file will be compiled.

- `cl(`$FileName$`)`: This predicate compiles and loads the source file named $FileName$`.pi`. Note that the extension `.pi` does not need to be given. The system also compiles and loads all of the module files that are either directly imported or indirectly imported by the source

file. The system searches for such dependent files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`.

- `cl`: This predicate compiles and loads a program from the console, ending when the end-of-file character (`ctrl-z` for Windows and `ctrl-d` for Unix) is typed.
- `cl_facts(`$Facts$`)`: This predicate compiles and loads facts into the system. The argument $Facts$ is a list of ground facts.
- `cl_facts(`$Facts$`, `$IndexInfo$`)`: This predicate compiles and loads facts into the system. The argument $Facts$ is a list of ground facts. The argument $IndexInfo$ is a list of indexing information in the form `p(`M_1`, `M_2`, ..., `M_n`)`. Each M_i can either be +, which indicates that the argument is input, or −, which indicates that the argument is output. Facts are indexed on input arguments.
- `cl_facts_table(`$Facts$`)`: This predicate is the same as `cl_facts/1`, except that the facts are all tabled.
- `cl_facts_table(`$Facts$`, `$IndexInfo$`)`: This predicate is the same as `cl_facts/2`, except that the facts are all tabled.

B.2 Tracing Execution

The Picat system has three execution modes: *non-trace mode*, *trace mode*, and *spy mode*. In trace mode, it is possible to trace the execution of a program, showing every call in every possible stage. In order to trace the execution, the program must be recompiled while the system is in trace mode. In spy mode, it is possible to trace the execution of individual functions and predicates. The following predicates are used to switch between non-trace mode and trace mode.

- `trace`: This predicate switches the execution mode to trace mode.
- `notrace`: This predicate switches the execution mode to non-trace mode.
- `debug`: This predicate switches the execution mode to trace mode.
- `nodebug`: This predicate switches the execution mode to non-trace mode.
- `spy(`$Point$`)`: This predicate places a spy point on $Point$, which is a function or a predicate, optionally followed by an arity. The creation of a spy point switches the Picat system to spy mode.
- `nospy`: This predicate removes all spy points, and switches the execution mode to non-trace mode.
- `abort`: This predicate terminates the current program. This can be used in all three execution modes.

B.2.1 Debugging Commands

In trace mode, the system displays a message when a function or a predicate is entered (`Call`), exited (`Exit`), re-entered (`Redo`) or has failed (`Fail`). After a function or a predicate is entered or re-entered, the system waits for a command from the user. A command is a single letter followed by a carriage-return, or may simply be a carriage-return. The following commands are available:

- `+` : create a spy point.
- `-` : remove a spy point.
- `<` : reset the print depth to 10.
- `< i` : reset the print depth to `i`.
- `a` : abort, quit debugging, moving control to the top level.
- `<cr>` : A carriage return causes the system to show the next call trace.
- `c` : creep, show the next call trace.
- `h` : help, display the debugging commands.
- `?` : help, display the debugging commands.
- `l` : leap, be silent until a spy point is encountered.
- `n` : nodebug, prevent the system from displaying debugging messages for the remainder of the program.
- `r` : repeat, continue to creep or leap without intervention.
- `s` : skip, be silent until the call is completed (`Exit` or `Fail`).
- `t` : backtrace, show the backtrace leading to the current call.
- `t i` : backtrace, show the backtrace from the call numbered `i` to the current call.
- `u` : undo what has been done to the current call and redo it.
- `u i` : undo what has been done to the call numbered `i` and redo it.

B.3 Information about the Picat System

The `sys` module contains a number of built-ins that display information about the Picat system. This information includes statistics about the system, including the memory that is used, and the amount of time that it takes to perform a goal.

B.3.1 Statistics

The following built-ins display statistics about the memory that the Picat system uses.

- `statistics`: This predicate displays the number of bytes that are allocated to each data area, and the number of bytes that are already in use.
- `statistics(`Key`,` $Value$`)`: The statistics concerning Key are $Value$. This predicate gives multiple solutions upon backtracking. Keys include `runtime`, `program`, `heap`, `control`, `trail`, `table`, `gc`, `backtracks`, `table_blocks` and `gc_time`. The values for most of the keys are lists of two elements. For the key `runtime`, the first element denotes the amount of time in milliseconds that has elapsed since Picat started, and the second element denotes the amount of time that has elapsed since the previous call to `statistics/2` was executed. For the key `gc`, the number indicates the number of times that the garbage collector has been invoked. For the key `gc_time`, the value indicates the

time spent on garbage collection. For the key `table_blocks`, the value gives the number of blocks that are used for the table area.[1] For the key `backtracks`, the number indicates the number of backtracks that have been done during the labeling of finite domain variables since Picat was started. For the keys `program`, `heap`, `control`, `trail`, and `table`, the first element denotes the size of memory in use, and the second element denotes the size of memory that is still available in the corresponding data area.

- `statistics_all() =` *List*: This function returns a list of lists that are in the form `[Key, Value]`. The list contains all of the keys that `statistics/2` can display, together with their corresponding values.

Example

```
Picat> statistics
       Stack+Heap:    8, 000, 000 bytes
         Stack in use:    1, 156 bytes
         Heap in use:    28, 592 bytes

       Program:       8, 000, 000 bytes
         In use:      1, 448, 436 bytes
         Symbols:         5, 300

       Trail:         4, 000, 000 bytes
         In use:            936 bytes

       Memory manager:
          GC:         Call(0), Time(0 ms)
          Expansions: Stack+Heap(0), Program(0), Trail(0), Table(0)

Picat> statistics(Key, Value)
       Key = runtime
       Value = [359947,66060]?;

       Key = program
       Value = [1451656,6548344]?;

       Key = heap
       Value = [34112,7964524]?;

       Key = control
       Value = [1360,7964524]?;

       Key = trail
       Value = [1496,3998504]?;

       Key = table
       Value = [0,4000000]?;
```

[1] Each block has 1 million words.

```
      Key = table_blocks
      Value = 1?;

      Key = gc
      Value = 0?;

      Key = backtracks
      Value = 0 ?;

      Key = gc_time
      Value = 0

Picat> L = statistics_all()
L = [[runtime, [359947,66060]], [program, [1451656,6548344]],
     [heap, [34112,7964524]], [control, [1360,7964524]],
     [trail, [1496,3998504]], [table, [0,4000000]],
     [table_blocks,1],[gc, 0], [backtracks, 0], [gc_time, 0]]
```

B.3.2 Time

The following predicates display the amount of CPU time that it takes to perform a goal.

- `time(`*Goal*`)`: This predicate calls *Goal*, and reports the number of seconds of CPU time that were consumed by the execution.
- `time2(`*Goal*`)`: This predicate calls *Goal*, and reports the number of seconds of CPU time that were consumed by the execution, and the number of backtracks that have been performed in labeling finite-domain variables during the execution of *Goal*.
- `time_out(`*Goal*`,` *Limit*`,` *Result*`)`: This predicate is logically equivalent to `once` *Goal*, except that it imposes a time limit, in milliseconds, on the evaluation. If *Goal* is not finished when *Limit* expires, then the evaluation will be aborted, and *Result* will be unified with the atom `time_out`. If *Goal* succeeds within the time limit, then *Result* will be unified with the atom `success`. Note that time-out may be delayed or never occur because of the execution of an external C function.

B.3.3 Other System Information

- `command(`*String*`)` = *Int*: This function sends the command *String* to the OS and returns the status that is returned from the OS.
- `help`: This predicate shows how to use the commands that the system accepts.
- `loaded_modules()` = *List*: This function returns a list of the modules that are currently loaded in the Picat system. This list includes library modules and user-defined modules. By default, this function returns `[basic,math,io,sys]`.
- `nolog`: This predicate turns off the logging flag.
- `picat_path()` = *String*: This function returns the directories that are stored in the environment variable `PICATPATH`. If the environment variable `PICATPATH` does not exist, then this function throws an error.

B.4 Garbage Collection

Picat incorporates an incremental garbage collector for the control stack and the heap. The garbage collector is active by default. The `sys` module includes the following predicates for garbage collection.

- `garbage_collect`: This predicate starts the garbage collector.
- `garbage_collect(`$Size$`)`: This predicate calls the garbage collector. If there are fewer than $Size$ words of combined free space on the control stack and heap after garbage collection, then it invokes the memory manager to expand the stack and heap so that there are $Size$ words of combined free space on the control stack and heap.

B.5 Quitting the Picat System

The following predicates can be used to terminate the Picat interpreter.

- `exit`
- `halt`

Appendix C

The `util` Module

The `util` module provides useful utility functions and predicates. This module is expected to be expanded in the future. This module must be imported before use.

C.1 Utilities on Terms

- `find_first_of(`$Term$`, `$Pattern$`) = `$Index$: This function returns the first index at which the argument unifies with $Pattern$. If there is no argument that unifies with $Pattern$, then this function returns -1. $Term$ must be either a list or a structure.
- `find_last_of(`$Term$`, `$Pattern$`) = `$Index$: This function returns the last index at which the argument unifies with $Pattern$. If there is no argument that unifies with $Pattern$, then this function returns -1. $Term$ must be either a list or a structure.
- `replace(`$Term$`, `Old`, `New`) = `$NewTerm$: This function returns a copy of $Term$, replacing all of the occurrences of Old in $Term$ by New.
- `replace_at(`$Term$`, `$Index$`, `New`) = `$NewTerm$: This function returns a copy of $Term$, replacing the argument at $Index$ by New. $Term$ must be a compound term.

C.2 Utilities on Strings and Lists

- `chunks_of(`$List$`, `K`) = `$ListOfLists$: This function splits $List$ into chunks, each of which has length K, and returns the list of the chunks. The last chunk may have fewer than K elements if the length of $List$ is not a multiple of K.
- `drop(`$List$`, `K`) = `$ResList$: This function returns the suffix of $List$ after the first K elements are dropped, or [] if K is greater than the length of $List$.
- `find(`$String$`, `$SubString$`, `$From$`, `To`)`: This predicate searches for an occurrence of $SubString$ in $String$, and binds $From$ to the starting index and To to the ending index. On backtracking, this predicate searches for the next occurrence of $SubString$.
- `find_ignore_case(`$String$`, `$SubString$`, `$From$`, `To`)`: This predicate is the same as `find(`$String$`, `$SubString$`, `$From$`, `To`)`, except that it is case insensitive.
- `join(`$Tokens$`) = `$String$: This function is the same as `join(`$Tokens$`, " ")`.
- `join(`$Tokens$`, `$Separator$`) = `$String$: This function concatenates the tokens $Tokens$ into a string, adding $Separator$, which is a string or an atom, between adjacent tokens.

- `lstrip(`$List$`)` = $ResList$: This function is the same as `lstrip(`$List$`, " \t\n\r")`.
- `lstrip(`$List$`,` $Elms$`)` = $ResList$: This function returns a copy of $List$ with leading elements in $Elms$ removed.
- `power_set(`$List$`)` = $ResLists$: This function returns the power set, i.e., a list of all subsets of a set represented as a list.
- `rstrip(`$List$`)` = $ResList$: This function is the same as `rstrip(`$List$`, " \t\n\r")`.
- `rstrip(`$List$`,` $Elms$`)` = $ResList$: This function returns a copy of $List$ with trailing elements in $Elms$ removed.
- `split(`$String$`)` = $Tokens$: This function uses whitespace characters as split characters. It is the same as `split(`$String$`, " \t\n\r")`.
- `split(`$String$`,` $Separators$`)` = $Tokens$: This function splits $String$ into a list of tokens, using characters in the string $Separators$ as split characters. Recall that a string is a list of characters. A token is a string, so the return value is a list of lists of characters.
- `strip(`$List$`)` = $ResList$: This function is the same as `strip(`$List$`, " \t\n\r")`.
- `strip(`$List$`,` $Elms$`)` = $ResList$: This function returns a copy of $List$ with leading and trailing elements in $Elms$ removed.
- `take(`$List$`,` K`)` = $ResList$: This function returns the prefix of $List$ of length K, or $List$ itself if K is greater than the length of $List$.

C.3 Utilities on Matrices

An *array matrix* is a two-dimensional array. The first dimension gives the number of rows and the second dimension gives the number of columns. A *list matrix* represents a matrix as a list of lists.

- `array_matrix_to_list(`$AMatrix$`)` = $List$: This function converts the array matrix $AMatrix$ to a flat list, row by row.
- `array_matrix_to_list_matrix(`$AMatrix$`)` = $LMatrix$: This function converts the array matrix $AMatrix$ to a list matrix.
- `columns(`$Matrix$`)` = $List$: This function returns the columns of the matrix $Matrix$ as a list.
- `diagonal1(`$Matrix$`)` = $List$: This function returns the primary diagonal of the matrix $Matrix$ as a list.
- `diagonal2(`$Matrix$`)` = $List$: This function returns the secondary diagonal of the matrix $Matrix$ as a list.
- `list_matrix_to_array_matrix(`$LMatrix$`)` = $AMatrix$: This function converts the list matrix $LMatrix$ to an array matrix.
- `matrix_multi(`$MatrixA$`,` $MatrixB$`)` = $MatrixC$: This function returns the product of matrices $MatrixA$ and $MatrixB$. Both $MatrixA$ and $MatrixB$ must be array matrices.

- `rows(`$Matrix$`) = `$List$: This function returns the rows of the matrix $Matrix$ as a list.
- `transpose(`$Matrix$`) = `$Transposed$: This function returns the transpose of the matrix $Matrix$. $Matrix$ can be an array matrix or a list matrix. If $Matrix$ is an array matrix, then the returned transpose is also an array. If $Matrix$ is a list matrix, then the returned transpose is also a list.

C.4 Utilities on Lists and Sets

- `nextto(`$E1$`, `$E2$`, `$List$`)`: This predicate is true if $E2$ follows $E1$ in $List$. This predicate is nondeterministic.
- `permutation(`$List$`, `$Perm$`)`: This predicate generates a permutation $Perm$ of $List$. This predicate is nondeterministic. On backtracking, it generates the next permutation.
- `permutations(`$List$`) = `$ResLists$: This function returns all the permutations of $List$.

Appendix D

The `ordset` Module

An ordered set is represented as a sorted list that does not contain duplicates. The `ordset` module provides useful utility functions and predicates on ordered sets. This module must be imported before use.

- `delete(`$OSet$`,` Elm`)` = $OSet_1$: This function returns a copy of $OSet$ that does not contain the element Elm.
- `disjoint(`$OSet_1$`,` $OSet_2$`)`: This predicate is true when $OSet_1$ and $OSet_2$ have no element in common.
- `insert(`$OSet$`,` Elm`)` = $OSet_1$: This function returns a copy of $OSet$ with the element Elm inserted.
- `intersection(`$OSet_1$`,` $OSet_2$`)` = $OSet_3$: This function returns an ordered set that contains elements which are in both $OSet_1$ and $OSet_2$.
- `new_ordset(`$List$`)` = $OSet$: This function returns an ordered set that contains the elements of $List$.
- `ordset(`$Term$`)`: This predicate is true if $Term$ is an ordered set.
- `subset(`$OSet_1$`,` $OSet_2$`)`: This predicate is true if $OSet_1$ is a subset of $OSet_2$.
- `subtract(`$OSet_1$`,` $OSet_2$`)` = $OSet_3$: This function returns an ordered set that contains all of the elements of $OSet_1$ which are not in $OSet_2$.
- `union(`$OSet_1$`,` $OSet_2$`)` = $OSet_3$: This function returns an ordered set that contains all of the elements which are present in either $OSet_1$ or $OSet_2$.

Appendix E

The `datetime` Module

Picat's `datetime` module provides built-ins for retrieving the date and time. This module must be imported before use.

- `current_datetime() =` $DateTime$: This function returns the current date and time as a structure in the form

  `date_time(`$Year$`,` $Month$`,` Day`,` $Hour$`,` $Minute$`,` $Second$`)`

  where the arguments are all integers, and have the following meanings and ranges.

Argument	**Meaning**	**Range**
$Year$	years since 1900	an integer
$Month$	months since January	0-11
Day	day of the month	1-31
$Hour$	hours since midnight	0-23
$Minute$	minutes after the hour	0-59
$Second$	seconds after the minute	0-60

  In the $Month$ argument, 0 represents January, and 11 represents December. In the $Hour$ argument, 0 represents 12 AM, and 23 represents 11 PM. In the $Second$ argument, the value 60 represents a leap second.

- `current_date() =` $Date$: This function returns the current date as a structure in the form `date(`$Year$`,` $Month$`,` Day`)`, where the arguments have the meanings and ranges that are defined above.

- `current_day() =` $WDay$: This function returns the number of days since Sunday, in the range 0 to 6.

- `current_time() =` $Time$: This function returns the current time as a structure in the form `time(`$Hour$`,` $Minute$`,` $Second$`)`, where the arguments have the meanings and ranges that are defined above.

Appendix F

Formats

F.1 Formatted Printing

The following table shows the specifiers that can be used in format strings for the built-ins `writef`, `printf`, and `to_fstring`.

Specifier	Output
`%%`	Percent Sign
`%c`	Character
`%d`	Signed Decimal Integer
`%e`	Scientific Notation, with Lowercase `e`
`%E`	Scientific Notation, with Uppercase `E`
`%f`	Decimal Real Number
`%g`	Shorter of `%e` and `%f`
`%G`	Shorter of `%E` and `%f`
`%i`	Signed Decimal Integer
`%n`	Platform-independent Newline
`%o`	Unsigned Octal Integer
`%s`	String
`%u`	Unsigned Decimal Integer
`%w`	Term
`%x`	Unsigned Lowercase Hexadecimal Integer
`%X`	Unsigned Uppercase Hexadecimal Integer

Appendix G

External Language Interface with C

Picat has an interface with C, through which Picat programs can call deterministic predicates that are written as functions in C. C programs that use this interface must include the file `"picat.h"` in the directory `Picat/Emulator`. In order to make C-defined predicates available to Picat, users have to recompile Picat's C source code together with the newly added C functions.

G.1 Term Representation

Picat's C interface provides functions for accessing, manipulating, and building Picat terms. In order to understand these functions, users need to know how terms are represented in Picat's virtual machine.

A term is represented by a word that contains a value and a tag. A word has 32 bits or 64 bits, depending on the underlying CPU and OS. The tag in a word distinguishes the type of the term.

The value of a term is an address, except when the term is a small integer (in which case, the value represents the integer itself). The location to which the address points is dependent on the type of the term. In a reference, the address points to the referenced term. An unbound variable is represented by a self-referencing pointer. In an atom, the address points to the record for the atom symbol in the symbol table. In a structure, $f(t_1, \ldots, t_n)$, the address points to a block of $n + 1$ consecutive words, where the first word points to the record for the functor, *f/n*, in the symbol table, and the remaining n words store the components of the structure. Arrays, floating-point numbers, and big integers are represented as special structures. Picat lists are singly linked lists. In a list, *[H|T]*, the address points to a block of two consecutive words, where the first word stores the car, *H*, and the second word stores the cdr, *T*.

G.2 Fetching Arguments of Picat Calls

A C function that defines a Picat predicate takes no arguments. The following function is used in order to fetch arguments in the current Picat call.

- `TERM picat_get_call_arg(int i, int arity)`: Fetch the `i`th argument, where `arity` is the arity of the predicate, and `i` must be an integer between 1 and `arity`. The validity of the arguments is not checked, and an invalid argument may cause fatal errors.

G.3 Testing Picat Terms

The following functions are provided for testing Picat terms. They return `PICAT_TRUE` when they succeed and `PICAT_FALSE` when they fail.

- `int picat_is_var(TERM t)`: Term `t` is a variable.
- `int picat_is_attr_var(TERM t)`: Term `t` is an attributed variable.
- `int picat_is_dvar(TERM t)`: Term `t` is an attributed domain variable.
- `int picat_is_bool_dvar(TERM t)`: Term `t` is an attributed Boolean variable.
- `int picat_is_integer(TERM t)`: Term `t` is an integer.
- `int picat_is_float(TERM t)`: Term `t` is a floating-point number.
- `int picat_is_atom(TERM t)`: Term `t` is an atom.
- `int picat_is_nil(TERM t)`: Term `t` is nil, i.e., the empty list `[]`.
- `int picat_is_list(TERM t)`: Term `t` is a list.
- `int picat_is_string(TERM t)`: Term `t` is a string.
- `int picat_is_structure(TERM t)`: Term `t` is a structure (but not a list).
- `int picat_is_array(TERM t)`: Term `t` is an array.
- `int picat_is_compound(TERM t)`: True if either `picat_is_list(t)` or `picat_is_structure(t)` is true.
- `int picat_is_identical(TERM t1, TERM t2)`: `t1` and `t2` are identical. This function is equivalent to the Picat call `t1 == t2`.
- `int picat_is_unifiable(TERM t1, TERM t2)`: `t1` and `t2` are unifiable. This is equivalent to the Picat call `not(not(t1 = t2))`.

G.4 Converting Picat Terms into C

The following functions convert Picat terms to C. If a Picat term does not have the expected type, then the global C variable `exception`, which is of type `TERM`, is assigned a term. A C program that uses these functions must check `exception` in order to see whether data are converted correctly. The converted data are only correct when `exception` is `(TERM)NULL`.

- `long picat_get_integer(TERM t)`: Convert the Picat integer `t` into C. The term `t` must be an integer; otherwise `exception` is set to `integer_expected` and 0 is returned. Note that precision may be lost if `t` is a big integer.
- `double picat_get_float(TERM t)`: Convert the Picat float `t` into C. The term `t` must be a floating-point number; otherwise `exception` is set to `number_expected`, and 0.0 is returned.
- `char *picat_get_atom_name(TERM t)`: Return a pointer to the string that is the name of atom `t`. The term `t` must be an atom; otherwise, `NULL` is returned after `exception` is set to `atom_expected`.
- `char *picat_get_struct_name(TERM t)`: Return a pointer to the string that is the name of structure `t`. The term `t` must be a structure; otherwise, `exception` is set to `structure_expected`, and `NULL` is returned.

- `int picat_get_struct_arity(TERM t)`: Return the arity of term `t`. The term `t` must be a structure; otherwise, `exception` is set to `structure_expected` and 0 is returned.

G.5 Manipulating and Writing Picat Terms

- `int picat_unify(TERM t1, TERM t2)`: Unify two Picat terms `t1` and `t2`. The result is `PICAT_TRUE` if the unification succeeds, and `PICAT_FALSE` if the unification fails.
- `TERM picat_get_arg(int i, TERM t)`: Return the `i`th argument of term `t`. The term `t` must be compound, and `i` must be an integer that is between 1 and the arity of `t`; otherwise, `exception` is set to `compound_expected`, and the Picat integer `0` is returned.
- `TERM picat_get_car(TERM t)`: Return the car of the list `t`. The term `t` must be a non-empty list; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.
- `TERM picat_get_cdr(TERM t)`: Return the cdr of the list `t`. The term `t` must be a non-empty list; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.
- `void picat_write_term(TERM t)`: Send term `t` to the standard output stream.

G.6 Building Picat Terms

- `TERM picat_build_var()`: Return a free Picat variable.
- `TERM picat_build_integer(long i)`: Return a Picat integer whose value is `i`.
- `TERM picat_build_float(double f)`: Return a Picat float whose value is `f`.
- `TERM picat_build_atom(char *name)`: Return a Picat atom whose name is `name`.
- `TERM picat_build_nil()`: Return an empty Picat list.
- `TERM picat_build_list()`: Return a Picat list whose car and cdr are free variables.
- `TERM picat_build_structure(char *name, int arity)`: Return a Picat structure whose name is `name` and arity is `arity`. The structure's arguments are all free variables.
- `TERM picat_build_array(int n)`: Return a Picat array whose size is `n`. The array's arguments are all free variables.

G.7 Registering C-defined Predicates

The following function registers a predicate that is defined by a C function.

```
insert_cpred(char *name, int arity, int (*func)())
```

The first argument is the predicate name, the second argument is the arity, and the third argument is the name of the function that defines the predicate. The function that defines the predicate takes no arguments. As described above, `picat_get_call_arg(i, arity)` is used to fetch arguments from the Picat call.

For example, the following registers a predicate whose name is `"p"`, and whose arity is 2.

```
extern int p();
insert_cpred("p", 2, p);
```

The C function's name does not need to be the same as the predicate name.

Predicates that are defined in C should be registered after the Picat engine is initialized, and before any call is executed. One good place for registering predicates is the `Cboot()` function in the file `cpreds.c`, which registers all of the C-defined built-ins of Picat. After registration, the predicate can be called. All C-defined predicates must be explicitly called with the module qualifier `bp`, as in `bp.p(a, X)`.

Example

Consider the Picat predicate:

```
p(a, X) => X = $f(a).
p(b, X) => X = [1].
p(c, X) => X = 1.2.
```

where the first argument is given and the second is unknown. The following steps show how to define this predicate in C, and how to make it callable from Picat.

Step 1 . Write a C function to implement the predicate. The following shows a sample:

```
#include "picat.h"

int p(){
  TERM a1, a2, a, b, c, f1, l1, f12;
  char *name_ptr;

  /*   prepare Picat terms */
  a1 = picat_get_call_arg(1, 2);       /* first argument */
  a2 = picat_get_call_arg(2, 2);       /* second argument */
  a = picat_build_atom("a");
  b = picat_build_atom("b");
  c = picat_build_atom("c");
  f1 = picat_build_structure("f", 1);  /* f(a) */
  picat_unify(picat_get_arg(1, f1), a);
  l1 = picat_build_list();             /* [1] */
  picat_unify(picat_get_car(l1), picat_build_integer(1));
  picat_unify(picat_get_cdr(l1), picat_build_nil());
  f12 = picat_build_float(1.2);        /* 1.2 */

  /* code for the rules */
  if (!picat_is_atom(a1))
      return PICAT_FALSE;
```

```
  name_ptr = picat_get_atom_name(a1);
  switch (*name_ptr){
  case 'a':
    return (picat_unify(a1, a) ?
              picat_unify(a2, f1) : PICAT_FALSE);
  case 'b':
    return (picat_unify(a1, b) ?
              picat_unify(a2, l1) : PICAT_FALSE);
  case 'c':
    return (picat_unify(a1, c) ?
              picat_unify(a2, f12) : PICAT_FALSE);
  default: return PICAT_FALSE;
  }
}
```

Step 2 . Insert the following two lines into `Cboot()` in `cpreds.c`:

```
extern int p();
insert_cpred("p", 2, p);
```

Step 3 . Modify the makefile, if necessary, and recompile the system. Now, `p/2` is in the group of built-ins in Picat.

Step 4 . Use `bp.p(...)` to call the predicate.

```
Picat> bp.p(a, X)
X = f(a)
```

Appendix H

Tokens

```
/* Picat lexical rules
      [...] means optional
      {...} means 0, 1, or more occurrences
      "..." means as-is
      /*  ... */ comment

   Tokens to be returned:

      Token-type     lexeme
     ====================
      ATOM           a string of chars of the atom name
      VARIABLE       a string of chars of the variable name
      INTEGER        an integer literal
      FLOAT          a float literal
      STRING         a string of chars
      OPERATOR       a string of chars in the operator
      SEPARATOR      one of "(" ")" "{" "}"  "[" "]"
*/
line_terminator ->
    the LF character, also known as "newline"
    the CR character, also known as "return"
    the CR character followed by the LF character

input_char ->
    utf8_char but not CR or LF

comment ->
    traditional_comment
    end_of_line_comment

traditional_comment ->
    "/*" comment_tail

comment_tail ->
    "*" comment_tail_star
    not_star comment_tail

comment_tail_star ->
    "/"
    "*" comment_tail_star
    not_star_not_slash comment_tail

not_star ->
    input_char but not "*"
    line_terminator

not_star_not_slash ->
    input_char but not "*" or "/"
    line_terminator

end_of_line_comment ->
    "%" {input_char} line_terminator
```

```
white_space ->
    the SP character, also known as "space"
    the HT character, also known as "horizontal tab"
    the FF character, also known as "form feed"
    line_terminator

token ->
    atom_token
    variable_token
    integer_literal
    real_literal
    string_literal
    operator
    separator

atom_token ->
    small_letter {alphanumeric}
    single_quoted_token

variable_token ->
    anonymous_variable
    named_variable

anonymous_variable ->
    "_"

named_variable ->
    "_" alphanumeric {alphanumeric}
    capital_letter {alphanumeric}

alphanumeric ->
    alpha_char
    decimal_digit

alpha_char ->
    "_"
    letter

letter ->
    small_letter
    capital_letter

single_quoted_token ->
    "'" {string_char} "'"

string_literal ->
    "\"" {string_char} "\""

string_char ->
    input_char
    escape_sequence

integer_literal ->
    decimal_numeral
    hex_numeral
    octal_numeral
    binary_numeral

decimal_numeral ->
    decimal_digit [decimal_digits_and_underscores]

decimal_digits_and_underscores ->
    decimal_digit_or_underscore
    decimal_digits_and_underscores decimal_digit_or_underscore

decimal_digit_or_underscore ->
    decimal_digit
    "_"
```

```
hex_numeral ->
    "0x" hex_digits
    "0X" hex_digits

hex_digits ->
    hex_digit [hex_digits_and_underscores]

hex_digits_and_underscores ->
    hex_digit_or_underscore
    hex_digits_and_underscores hex_digit_or_underscore

hex_digit_or_underscore ->
    hex_digit
    "_"

octal_numeral ->
    "0O" octal_digits
    "0o" octal_digits

octal_digits ->
    octal_digit [octal_digits_and_underscores]

octal_digits_and_underscores ->
    octal_digit_or_underscore
    octal_digits_and_underscores octal_digit_or_underscore

octal_digit_or_underscore ->
    octal_digit
    "_"

binary_numeral ->
    "0b" binary_digits
    "0B" binary_digits

binary_digits ->
    binary_digit [binary_digits_and_underscores]

binary_digits_and_underscores ->
    binary_digit_or_underscore
    binary_digits_and_underscores binary_digit_or_underscore

binary_digit_or_underscore ->
    binary_digit
    "_"

real_literal ->
    decimal_numeral "." decimal_numeral [exponent_part]

exponent_part ->
    exponent_indicator signed_integer

exponent_indicator ->
    "e"
    "E"

signed_integer ->
    [sign] decimal_numeral

sign ->
    "+"
    "-"

separator ->
    one of "(" ")" "{" "}"  "[" "]"

operator ->
    one of
      "=" "!=" ">" ">=" "<" "<=" "=<" ".." "!"
      "," ";" ":" "::" "." ". " (dot-whitespace)
      "=>" "?=>" "==" "!==" ":=" "|" "$" "@"
```

```
      ":-" "-->" "->" "&&" "\+" "||" "//" "=:="
      "=\=" "=.." "/\" "\/" "~" "<<" ">>"
      "+" "-" "*" "**" "/" "/>" "/<" "^"
      "#=" "#!=" "#>" "#>=" "#<" "#<=" "#=<"
      "#/\" "#\/" "#~" "#^" "#=>" "#<=>"
      "@>" "@>=" "@<" "@<=" "@=<"

small_letter ->
    one of "a" "b" ... "z"

capital_letter ->
    one of "A" "B" ... "Z"

decimal_digit ->
    one of "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"

hd ->
    hex_digit

hex_digit ->
    one of
       "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
       "a" "b" "c" "d" "e" "f" "A" "B" "C" "D" "E" "F"

octal_digit ->
    one of "0" "1" "2" "3" "4" "5" "6" "7"

binary_digit ->
    one of "0" "1"

escape_sequence ->
    "\""             /* double quote " */
    "\'"             /* single quote ' */
    "\\"             /* backslash \ */
    "\`"             /* back quote ` */
    "\a"             /* alarm */
    "\b"             /* backspace */
    "\f"             /* form feed */
    "\n"             /* line feed */
    "\r"             /* carriage return */
    "\t"             /* horizontal tab */
    "\uhh...h"       /* unicode (utf-8) code point */
    "\v"             /* vertical tab */
```

Appendix I

Grammar

```
/* Picat syntax rules
      [...] means optional
      {...} means 0, 1, or more occurrences
      (a | b) means choice
      "..." means a token
      %... one-line comment
      input tokens:
            atom_token
            integer_literal
            operator
            real_literal
            separator
            string_literal
            variable_token
            eor is "." followed by a whitespace character or eof
*/
program ->
    [module_declaration]
    {import_declaration}
    program_body

program_body ->
    {predicate_definition | function_definition | actor_definition | include_directive}

module_declaration ->
    "module" atom_token eor

import_declaration ->
    import import_item {"," import_item} eor

import_item ->
    atom_token

predicate_definition ->
    {predicate_directive} predicate_rule_or_fact {predicate_rule_or_fact}

function_definition ->
    {function_directive} function_rule_or_fact {function_rule_or_fact}

actor_definition ->
    ["private"] action_rule {(action_rule
                              | nonbacktrackable_predicate_rule)}

function_directive ->
    "private"
    "table"

predicate_directive ->
    "private"
    "table" ["(" table_mode {"," table_mode} ")" ]
    "index" index_declaration {index_declaration}
```

```
include_directive ->
    "include" string_literal eor

index_declaration ->
    "(" index_mode {"," index_mode} ")"

index_mode ->
   "+"
   "-"

table_mode ->
   "+"
   "-"
   "min"
   "max"
   "mmin"
   "mmax"
   "nt"

predicate_rule_or_fact ->
    predicate_rule
    predicate_fact

function_rule_or_fact ->
    function_rule
    function_fact

predicate_rule ->
    head ["," condition] ("=>" | "?=>") body eor
    head (":-" | "-->") body eor

nonbacktrackable_predicate_rule ->
    head ["," condition] "=>" body eor

predicate_fact ->
    head eor

head ->
    atom_token ["(" [term {"," term}] ")"]

function_rule ->
    head "=" expression ["," condition] "=>" body eor

function_fact ->
    head "=" argument eor

action_rule ->
    head ["," condition] "," "{" event_pattern "}"   => body eor

event_pattern ->
    term {',' term}

condition -> goal

body -> goal

argument ->
    negative_goal

goal ->
    disjunctive_goal

disjunctive_goal ->
    disjunctive_goal (";" | "||") conjunctive_goal
    conjunctive_goal "->" disjunctive_goal         % Prolog-style if-then
    conjunctive_goal

conjunctive_goal ->
    conjunctive_goal ("," | "&&") negative_goal
    negative_goal
```

```
negative_goal ->
   "not" negative_goal
   "\+" negative_goal
   "once" negative_goal
   equiv_constr

equiv_constr ->
   equiv_constr "#<=>" impl_constr
   impl_constr

impl_constr ->
   impl_constr "#=>" or_constr
   or_constr

or_constr ->
   or_constr "#\/" xor_constr
   xor_constr

xor_constr ->
   xor_constr "#^" and_constr
   and_constr

and_constr ->
   and_constr "#/\" not_constr
   not_constr

not_constr ->
   "#~" not_constr
   enclosed_goal

enclosed_goal ->
   "if" if_cond goal {"elseif" if_cond goal} ["else" goal] "end"
   "foreach" "(" iterator {"," (iterator | condition)} ")" goal "end"
   "while" "(" goal ")" ["do"] goal "end"
   "do" goal "while" "(" goal ")"
   expression {bin_rel_op expression}

iterator ->
   expression "in" expression

if_cond -> "(" goal ")" ["then"]       % "then" is optional
if_cond -> goal "then"                 % "then" is required if goal is
                                       % not enclosed in parentheses

bin_rel_op ->
    "!="
    "!=="
    "#!="
    "#<"
    "#<="
    "#="
    "#=<"
    "#>"
    "#>="
    "::"
    ":="
    "<"
    "<="
    "="
    "=.."
    "=:="
    "=<"
    "=="
    "=\="
    ">"
    ">="
    "@<"
    "@<="
    "@=<"
```

```
    "@>"
    "@>="
    "in"
    "is"
    "notin"

expression ->
    concat_expression

concat_expression ->
    range_expression ["++" concat_expression]

range_expression ->
   or_expression [".." or_expression [".." or_expression]]

or_expression  ->
    xor_expression
    or_expression "\/" xor_expression

xor_expression ->
    and_expression
    xor_expression "^" and_expression              % bitwise xor

and_expression ->
    shift_expression
    and_expression  "/\" shift_expression

shift_expression ->
    additive_expression
    shift_expression ( "<<" | ">>" ) additive_expression

additive_expression ->
    multiplicative_expression
    additive_expression "+" multiplicative_expression
    additive_expression "-" multiplicative_expression

multiplicative_expression ->
    unary_expression
    multiplicative_expression "*" unary_expression
    multiplicative_expression "/" unary_expression
    multiplicative_expression "//" unary_expression
    multiplicative_expression "/>" unary_expression
    multiplicative_expression "/<" unary_expression
    multiplicative_expression "div" unary_expression
    multiplicative_expression "mod" unary_expression
    multiplicative_expression "rem" unary_expression

unary_expression ->
    power_expression
    "+" unary_expression
    "-" unary_expression
    "~" unary_expression                           % bitwise complement

power_expression ->
    primary_expression ["**" unary_expression]

primary_expression ->
    "(" goal ")"
    anonymous_function
    anonymous_predicate
    variable "[" argument {"," argument} "]"       % subscript notation
    variable "@" term ["@"]                        % as-pattern
    atom_or_call
    integer_literal
    real_literal
    string_literal
    variable_token
    higher_order_call
    list_expression
    array_expression
```

```
    function_call
    term_constructor
    atom_token "." higher_order_call               % module qualified higher-order call
    primary_expression "." atom_or_call            % dot-notation

atom_or_call ->
    atom_token ["(" [argument {"," argument}] ")"]

higher_order_call ->
    variable "(" [argument {"," argument}] ")"     % higher order call variable

list_expression ->
    "[]"
    "[" argument list_expression_suffix "]"

list_expression_suffix ->
    ":" iterator {"," (iterator | condition)}      % list comprehension
    {"," argument} ["|" argument]

array_expression ->
    "{}"
    "{" argument array_expression_suffix "}"

array_expression_suffix ->
    ":" iterator {"," (iterator | condition)}      % array comprehension
    {"," argument}

function_call ->
    [primary_expression "."] atom_token "(" [argument {"," argument}] ")"

/* Anonymous functions and predicates are closure-valued expressions.
   Their outer parentheses and bracketed parameter lists are required.
   An empty parameter list is written as "[]". */

anonymous_function ->
    "(" anonymous_head "->" expression ["," body] ")"

anonymous_predicate ->
    "(" anonymous_head "=>" body ")"

anonymous_head ->
     lambda_parameters

lambda_parameters ->
    "[]"
    "[" term {"," term} "]"

/* In an anonymous function, the first comma following the return expression
   separates that expression from the body. The body has the full goal syntax.
   Omitting the body is equivalent to using the body true.

   A parenthesized form beginning with lambda_parameters followed by "->",
   or "=>" is parsed as a lambda expression, rather than as an ordinary
   parenthesized goal. In particular, "([X] -> E)" is an anonymous function,
   not a Prolog-style if-then.

   Variables in parameter terms introduce bindings local to the lambda.
   Parameter terms follow the same matching rules as named head arguments.
   Other references
   to enclosing lexical bindings are captured; remaining variables are local
   to each invocation. Anonymous occurrences of _ are fresh independently.
   Nested lambdas obey the same lexical scope rules.

   Creating a lambda constructs a closure; it does not execute its body or
   evaluate its return expression. An anonymous function is invoked in a
   function context, and an anonymous predicate in a predicate context, using
   the existing higher_order_call syntax (or apply/call).

   An anonymous predicate has one rule, using =>. A function lambda
   follows ordinary function-rule behavior; its return expression is evaluated
```

```
   after its body. Calls within a lambda retain the defining module context.
   These scope and execution requirements are semantic, not grammar rules. */

term_constructor ->
    "$" goal ["$"]

/* A term has the same form as a goal except that it cannot contain assignments,
   loops or if statements. Note that subscript notations, range expressions, dot
   notations, list comprehensions, anonymous definitions, and higher-order function
   calls are still treated as functions in term constructors */
```

Appendix J

Operators

Precedence	Operators
Highest	`.`, `@`
	`**` (right-associative)
	unary +, unary −, ~
	`*`, `/`, `//`, `/<`, `/>`, `div`, `mod`, `rem`
	binary +, binary −
	`>>`, `<<`
	`/\`
	`^`
	`\/`
	`..`
	++ (right-associative)
	`is`, `=`, `!=`, `:=`, `==`, `!==`, `=:=`, `=\=`, `<`, `=<`, `<=`, `>`, `>=`, `::`, `in`, `notin`, `=..` `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, `#>=`, `@<`, `@=<`, `@<=`, `@>`, `@>=`
	`#~`
	`#/\`
	`#^`
	`#\/`
	`#=>` (right-associative)
	`#<=>`
	`not`, `once`, `\+`
	`,` (right-associative), && (right-associative)
Lowest	`;` (right-associative), || (right-associative)

Appendix K

The Library Modules

Module `basic` (imported by default)

- `\+` *Call*
- *X* `!=` *Y*
- *X* `!==` *Y*
- *X* `:=` *Y*
- *X* `<` *Y*
- *X* `<=` *Y*
- *X* `=` *Y*
- *X* `=<` *Y*
- *X* `=:=` *Y*
- *X* `=\=` *Y*
- *X* `==` *Y*
- *X* `>` *Y*
- *X* `>=` *Y*
- *X* `@<` *Y*
- *X* `@<=` *Y*
- *X* `@=<` *Y*
- *X* `@>` *Y*
- *X* `@>=` *Y*
- *X* `=..` *Y*
- *Term*$_1$ `++` *Term*$_2$ `=` *List*
- `[`*X* `:` *I* `in` *D*`,...]` `=` *List*
- *L* `..` *U* `=` *List*
- *L* `..` *Step* `..` *U* `=` *List*
- `-`*X* `=` *Y*
- `+`*X* `=` *Y*
- *X* `+` *Y* `=` *Z*
- *X* `-` *Y* `=` *Z*
- *X* `*` *Y* `=` *Z*
- *X* `/` *Y* `=` *Z*
- *X* `//` *Y* `=` *Z*
- *X* `div` *Y* `=` *Z*
- *X* `/<` *Y* `=` *Z*
- *X* `/>` *Y* `=` *Z*
- *X* `**` *Y* `=` *Z*
- *X* `mod` *Y* `=` *Z*
- *X* `rem` *Y* `=` *Z*
- `~`*X* `=` *Y*
- *X* `\/` *Y* `=` *Z*
- *X* `/\` *Y* `=` *Z*
- *X* `^` *Y* `=` *Z*
- *X* `<<` *Y* `=` *Z*
- *X* `>>` *Y* `=` *Z*
- *Var*`[`*Index*$_1$`,...,`*Index*$_n$`]`
- *Goal*$_1$`,` *Goal*$_2$
- *Goal*$_1$ `&&` *Goal*$_2$
- *Goal*$_1$`;`*Goal*$_2$
- *Goal*$_1$ `||` *Goal*$_2$
- `acyclic_term(`*Term*`)`
- `and_to_list(`*Conj*`)` `=` *List*
- `append(`*X*`,` *Y*`,` *Z*`)` (nondet)
- `append(`*X*`,` *Y*`,` *Z*`,` *T*`)` (nondet)
- `apply(`*S*`,` *Arg*$_1$`, ...,` *Arg*$_n$`)` `=` *Val*
- `arg(`*I*`,` *T*`,` *A*`)`
- `arity(`*Struct*`)` `=` *Arity*
- `array(`*Term*`)`
- `ascii_alpha(`*Term*`)`
- `ascii_alpha_digit(`*Term*`)`
- `ascii_digit(`*Term*`)`
- `ascii_lowercase(`*Term*`)`
- `ascii_uppercase(`*Term*`)`
- `atom(`*Term*`)`
- `atom_chars(`*Atm*`)` `=` *String*
- `atom_codes(`*Atm*`)` `=` *List*
- `atomic(`*Term*`)`
- `attr_var(`*Term*`)`
- `avg(`*ListOrArray*`)` `=` *Val*
- `between(`*From*`,` *To*`,` *X*`)` (nondet)
- `between(`*From*`,` *Step*`,` *To*`,` *X*`)` (nondet)
- `bigint(`*Term*`)`
- `bind_vars(`*Term*`,` *Val*`)`
- `bool_dvar(`*Term*`)`
- `call(`*S*`,` *Arg*$_1$`, ...,` *Arg*$_n$`)`
- `call_cleanup(`*S*`,` *Cleanup*`)`
- `catch(`*Goal*`,` *Exception*`,` *Handler*`)`
- `char(`*Term*`)`
- `chr(`*Code*`)` `=` *Char*
- `clear(`*Map*`)`
- `compare_terms(`*Term*$_1$`,` *Term*$_2$`)` `=` *Res*
- `compound(`*Term*`)`
- `copy_term(`*Term*$_1$`)` `=` *Term*$_2$
- `copy_term_shallow(`*Term*$_1$`)` `=` *Term*$_2$
- `count_all(`*Call*`)` `=` *Int*
- `del(`*Map*`,` *Key*`)`
- `delete(`*List*`,` *X*`)` `=` *ResList*
- `delete_all(`*List*`,` *X*`)` `=` *ResList*
- `different_terms(`*Term*$_1$`,` *Term*$_2$`)`
- `digit(`*Char*`)`
- `dvar(`*Term*`)`
- `dvar_or_int(`*Term*`)`
- `fail`

- false
- find_all(*Template*, *Call*) = *List*
- findall(*Template*, *Call*) = *List*
- first(*Compound*) = *Term*
- flatten(*List*) = *ResList*
- float(*Term*)
- fold(*F*, *Acc*, *List*) = *Res*
- freeze(*X*, *Goal*)
- functor(*T*, *F*, *N*)
- get(*Map*, *Key*) = *Val*
- get(*Map*, *Key*, *DefaultVal*) = *Val*
- get_attr(*X*, *Key*) = *Val*
- get_attr(*X*, *Key*, *DefaultVal*) = *Val*
- get_global_map(*ID*) = *Map*
- get_global_map() = *Map*
- get_heap_map(*ID*) = *Map*
- get_heap_map() = *Map*
- get_table_map(*ID*) = *Map*
- get_table_map() = *Map*
- ground(*Term*)
- has_key(*Map*, *Key*)
- hash_code(*Term*) = *Int*
- head(*List*) = *Term*
- heap_is_empty(*Heap*)
- heap_pop(*Heap*) = *Elm*
- heap_push(*Heap*, *Elm*)
- heap_size(*Heap*) = *Size*
- heap_to_list(*Heap*) = *List*
- heap_top(*Heap*) = *Elm*
- insert(*List*, *Index*, *Elm*) = *ResList*
- insert_all(*List*, *Index*, *AList*) = *ResList*
- insert_ordered(*List*, *Term*) = *ResList*
- insert_ordered_down(*List*, *Term*) = *ResList*
- int(*Term*)
- integer(*Term*)
- is(T_1, T_2)
- keys(*Map*) = *List*
- last(*Compound*) = *Term*
- len(*Term*) = *Len*
- length(*Term*) = *Len*
- list(*Term*)
- list_to_and(*List*) = *Conj*
- lowercase(*Term*)
- map(*Func*, *List*1, *List*2) = *ResList*
- map(*FuncOrList*, *ListOrFunc*) = *ResList*
- map(*Term*)
- map_to_list(*Map*) = *PairsList*
- max(*ListOrArray*) = *Val*
- max(*X*, *Y*) = *Val*
- maxint_small() = *Int*
- maxof(*Call*, *Objective*)
- maxof(*Call*, *Objective*, *ReportCall*)
- maxof_inc(*Call*, *Objective*)
- maxof_inc(*Call*, *Objective*, *ReportCall*)
- membchk(*Term*, *List*)
- member(*Term*, *List*) (nondet)
- min(*ListOrArray*) = *Val*
- min(*X*, *Y*) = *Val*
- minint_small() = *Int*
- minof(*Call*, *Objective*)
- minof(*Call*, *Objective*, *ReportCall*)
- minof_inc(*Call*, *Objective*)
- minof_inc(*Call*, *Objective*, *ReportCall*)
- name(*Struct*) = *Name*
- new_array(D_1, ..., D_n) = *Arr*
- new_list(*N*) = *List*
- new_list(*N*, *InitVal*) = *List*
- new_map() = *Map*
- new_map(*IntOrPairsList*) = *Map*
- new_map(*N*, *PairsList*) = *Map*
- new_max_heap(*IntOrList*) = *Heap*
- new_min_heap(*IntOrList*) = *Heap*
- new_set() = *Set*
- new_set(*N*, *KeysList*) = *Set*
- new_set(*IntOrKeysList*) = *Set*
- new_struct(*Name*, *IntOrList*) = *Struct*
- nonvar(*Term*)
- not *Call*
- nth(*Index*, *ListOrArray*, *Elem*) (nondet)
- number(*Term*)
- number_chars(*Num*) = *String*
- number_codes(*Num*) = *List*
- number_vars(*Term*)
- number_vars(*Term*, N_0) = N_1
- once *Call*
- ord(*Char*) = *Int*
- parse_radix_string(*String*, *Base*) = *Int*
- parse_term(*String*) = *Term*
- parse_term(*String*, *Term*, *Vars*)
- post_event(*X*, *Event*)
- post_event_any(*X*, *Event*)
- post_event_bound(*X*)
- post_event_dom(*X*, *Event*)
- post_event_ins(*X*)
- prod(*Collection*) = *Val*
- put(*Map*, *Key*)
- put(*Map*, *Key*, *Val*)
- put_attr(*X*, *Key*)
- put_attr(*X*, *Key*, *Val*)
- real(*Term*)
- reduce(*Func*, *List*) = *Res*
- reduce(*Func*, *List*, *InitVal*) = *Res*
- remove_dups(*ListOrArray*) = *ResList*
- repeat (nondet)
- reverse(*ListOrArray*) = *Res*
- second(*Compound*) = *Term*
- select(*X*, *List*, *ResList*) (nondet)
- size(*Map*) = *Size*
- slice(*ListOrArray*, *From*) = *Res*
- slice(*ListOrArray*, *From*, *To*) = *Res*
- sort(*ListOrArray*) = *Sorted*
- sort(*ListOrArray*, *KeyIndex*) = *Sorted*
- sort_down(*ListOrArray*) = *Sorted*
- sort_down(*ListOrArray*, *KeyIndex*) = *Sorted*
- sort_down_remove_dups(*ListOrArray*) = *Sorted*
- sort_down_remove_dups(*ListOrArray*, *KeyIndex*) = *Sorted*
- sort_remove_dups(*ListOrArray*) = *Sorted*

- sort_remove_dups(*ListOrArray*, *KeyIndex*) = *Sorted*
- sorted(*ListOrArray*)
- sorted_down(*ListOrArray*)
- string(*Term*)
- struct(*Term*)
- subsumes($Term_1$, $Term_2$)
- sum(*Collection*) = *Val*
- tail(*List*) = *Term*
- throw(*E*)
- to_array(*List*) = *Array*
- to_atom(*String*) = *Atom*
- to_binary_string(*Int*) = *String*
- to_codes(*Term*) = *List*
- to_float(*NS*) = *Real*
- to_fstring(*Format*, *Args...*) = *String*
- to_hex_string(*Int*) = *String*
- to_int(*NCS*) = *Int*
- to_integer(*NCS*) = *Int*
- to_list(*Struct*) = *List*
- to_lowercase(*String*) = *LString*
- to_number(*NCS*) = *Num*
- to_oct_string(*Int*) = *String*
- to_radix_string(*Int*, *Base*) = *String*
- to_real(*NS*) = *Real*
- to_string(*Term*) = *String*
- to_uppercase(*String*) = *UString*
- true
- uppercase(*Term*)
- values(*Map*) = *List*
- var(*Term*)
- variant($Term_1$, $Term_2$)
- vars(*Term*) = *Vars*
- zip($List_1$, $List_2$) = *List*
- zip($List_1$, $List_2$, $List_3$) = *List*
- zip($List_1$, $List_2$, $List_3$, $List_4$) = *List*

Module `math` (imported by default)

- abs(*X*) = *Val*
- acos(*X*) = *Val*
- acosh(*X*) = *Val*
- acot(*X*) = *Val*
- acoth(*X*) = *Val*
- acsc(*X*) = *Val*
- acsch(*X*) = *Val*
- asec(*X*) = *Val*
- asech(*X*) = *Val*
- asin(*X*) = *Val*
- asinh(*X*) = *Val*
- atan(*X*) = *Val*
- atan2(*Y*, *X*) = *Val*
- atanh(*X*) = *Val*
- binom(*N*, *K*) = *Val*
- ceiling(*X*) = *Val*
- cos(*X*) = *Val*
- cosh(*X*) = *Val*
- cot(*X*) = *Val*
- coth(*X*) = *Val*
- csc(*X*) = *Val*
- csch(*X*) = *Val*
- e() = 2.71828182845904523536
- even(*N*)
- exp(*X*) = *Val*
- floor(*X*) = *Val*
- frand() = *Val*
- frand(*Low*, *High*) = *Val*
- gcd(*A*, *B*) = *Val*
- log(*X*) = *Val*
- log(B, *X*) = *Val*
- log10(*X*) = *Val*
- log2(*X*) = *Val*
- modf(*X*) = (*IntVal*, *FractVal*)
- odd(*N*)
- pi() = 3.14159265358979323846
- pow(*X*, *Y*) = *Val*
- pow_mod(*X*, *Y*, *Z*) = *Val*
- prime(*N*)
- primes(*N*) = *List*
- rand_max() = *Val*
- random() = *Val*
- random(*Low*, *High*) = *Val*
- random(*Seed*) = *Val*
- random2() = *Val*
- round(*X*) = *Val*
- sec(*X*) = *Val*
- sech(*X*) = *Val*
- sign(*X*) = *Val*
- sin(*X*) = *Val*
- sinh(*X*) = *Val*
- sqrt(*X*) = *Val*
- tan(*X*) = *Val*
- tanh(*X*) = *Val*
- to_degrees(*Radian*) = *Degree*
- to_radians(*Degree*) = *Radian*
- truncate(*X*) = *Val*

Module `io` (imported by default)

- `at_end_of_stream(`FD`)`
- `close(`FD`)`
- `flush(`FD`)`
- `nl(`FD`)`
- `nl()`
- `open(`$Name$`) = `FD
- `open(`$Name$`, `$Mode$`) = `FD
- `peek_byte(`FD`) = `Int
- `peek_char(`FD`) = `Val
- `print(`FD`, `$Term$`)`
- `print(`$Term$`)`
- `printf(`FD`, `$Format$`, `$Args...$`)`
- `println(`FD`, `$Term$`)`
- `println(`$Term$`)`
- `read_atom(`FD`) = `$Atom$
- `read_atom() = `$Atom$
- `read_byte(`FD`) = `Int
- `read_byte(`FD`, `N`) = `$List$
- `read_byte() = `Int
- `read_char(`FD`) = `$Char$
- `read_char(`FD`, `N`) = `$String$
- `read_char() = `$Char$
- `read_char_code(`FD`) = `Int
- `read_char_code(`FD`, `N`) = `$List$
- `read_char_code() = `Int
- `read_file_bytes(`$File$`) = `$List$
- `read_file_bytes() = `$List$
- `read_file_chars(`$File$`) = `$String$
- `read_file_chars() = `$String$
- `read_file_codes(`$File$`) = `$List$
- `read_file_codes() = `$List$
- `read_file_lines(`$File$`) = `$Lines$
- `read_file_lines() = `$Lines$
- `read_file_terms(`$File$`) = `$Terms$
- `read_file_terms() = `$Terms$
- `read_int(`FD`) = `Int
- `read_int() = `Int
- `read_line(`FD`) = `$String$
- `read_line() = `$String$
- `read_number(`FD`) = `Num
- `read_number() = `Num
- `read_picat_token(`FD`) = `$TokenValue$
- `read_picat_token(`FD`, `$TokenType$`, `$TokenValue$`)`
- `read_picat_token(`$TokenType$`, `$TokenValue$`)`
- `read_picat_token() = `$TokenValue$
- `read_real(`FD`) = `$Real$
- `read_real() = `$Real$
- `read_term(`FD`) = `$Term$
- `read_term() = `$Term$
- `readln(`FD`) = `$String$
- `readln() = `$String$
- `write(`FD`, `$Term$`)`
- `write(`$Term$`)`
- `write_byte(`$Bytes$`)`
- `write_byte(`FD`, `$Bytes$`)`
- `write_char(`$Chars$`)`
- `write_char(`FD`, `$Chars$`)`
- `write_char_code(`$Codes$`)`
- `write_char_code(`FD`, `$Codes$`)`
- `writef(`FD`, `$Format$`, `$Args...$`)`
- `writeln(`FD`, `$Term$`)`
- `writeln(`$Term$`)`

Module `ordset`

- `delete(`$OSet$`, `Elm`) = `$OSet1$
- `disjoint(`$OSet1$`, `$OSet2$`)`
- `insert(`$OSet$`, `Elm`) = `$OSet1$
- `intersection(`$OSet1$`, `$OSet2$`) = `$OSet3$
- `new_ordset(`$List$`) = `$OSet$
- `ordset(`$Term$`)`
- `subset(`$OSet1$`, `$OSet2$`)`
- `subtract(`$OSet1$`, `$OSet2$`) = `$OSet3$
- `union(`$OSet1$`, `$OSet2$`) = `$OSet3$

Module `os`

- `cd(`$Path$`)`
- `chdir(`$Path$`)`
- `cp(`$FromPath$`, `$ToPath$`)`
- `cwd() = `$Path$
- `directory(`$Path$`)`
- `dir`
- `env_exists(`$Name$`)`
- `executable(`$Path$`)`
- `exists(`$Path$`)`
- `file(`$Path$`)`
- `file_base_name(`$Path$`) = `$String$
- `file_directory_name(`$Path$`) = `$String$
- `file_exists(`$Path$`)`
- `getenv(`$Name$`) = `$String$
- `listdir(`$Path$`) = `$List$
- `ls`
- `mkdir(`$Path$`)`
- `pwd() = `$Path$
- `readable(`$Path$`)`
- `rename(`Old`, `New`)`
- `rm(`$Path$`)`
- `rmdir(`$Path$`)`
- `separator() = `$Char$
- `size(`$Path$`) = `Int
- `writable(`$Path$`)`

Modules `cp`, `mip`, `sat`, and `smt`

- `#~`X
- X `#!=` Y
- X `#/\` Y
- X `#<` Y
- X `#<=` Y
- X `#<=>` Y
- X `#=` Y
- X `#=<` Y
- X `#=>` Y
- X `#>` Y
- X `#>=` Y
- X `#\/` Y
- X `#^` Y
- $Vars$ `::` Exp
- $Vars$ `notin` Exp
- `acyclic(`Vs`,` Es`)`
- `acyclic_d(`Vs`,` Es`)`
- `all_different(`$FDVars$`)`
- `all_different_except_0(`$FDVars$`)`
- `all_distinct(`$FDVars$`)`
- `assignment(`$FDVars1$`,` $FDVars2$`)`
- `at_least(`N`,` L`,` V`)`
- `at_most(`N`,` L`,` V`)`
- `bv_add(`A`,` B`,` C`)` (sat only)
- `bv_and(`A`,` B`,` C`)` (sat only)
- `bv_div(`A`,` B`,` C`)` (sat only)
- `bv_drop(`A`,` N`) =` Res (sat only)
- `bv_eq(`A`,` B`)` (sat only)
- `bv_ge(`A`,` B`)` (sat only)
- `bv_gt(`A`,` B`)` (sat only)
- `bv_le(`A`,` B`)` (sat only)
- `bv_lt(`A`,` B`)` (sat only)
- `bv_mod(`A`,` B`,` C`)` (sat only)
- `bv_mul(`A`,` B`,` C`)` (sat only)
- `bv_neq(`A`,` B`)` (sat only)
- `bv_or(`A`,` B`,` C`)` (sat only)
- `bv_pow(`A`,` B`,` C`)` (sat only)
- `bv_sum(`As`,` B`)` (sat only)
- `bv_take(`A`,` N`) =` Res (sat only)
- `bv_to_int(`A`) =` Res (sat only)
- `bv_xor(`A`,` B`,` C`)` (sat only)
- `circuit(`$FDVars$`)`
- `count(`V`,` $FDVars$`,` N`)`
- `count(`V`,` $FDVars$`,` Rel`,` N`)`
- `cumulative(`Ss`,` Ds`,` Rs`,` $Limit$`)`
- `decreasing(`L`)`
- `decreasing_strict(`L`)`
- `diffn(`$RectangleList$`)`
- `disjunctive_tasks(`$Tasks$`)` (cp only)
- `element(`I`,` $List$`,` V`)`
- `element0(`I`,` $List$`,` V`)`
- `exactly(`N`,` L`,` V`)`
- `fd_degree(`$FDVar$`) =` $Degree$ (cp only)
- `fd_disjoint(`$DVar1$`,` $DVar2$`)`
- `fd_dom(`$FDVar$`) =` $List$
- `fd_false(`$FDVar$`,` Elm`)`
- `fd_max(`$FDVar$`) =` Max
- `fd_min(`$FDVar$`) =` Min
- `fd_min_max(`$FDVar$`,` Min`,` Max`)`
- `fd_next(`$FDVar$`,` Elm`) =` $NextElm$
- `fd_prev(`$FDVar$`,` Elm`) =` $PrevElm$
- `fd_set_false(`$FDVar$`,` Elm`)` (cp only)
- `fd_size(`$FDVar$`) =` $Size$
- `fd_true(`$FDVar$`,` Elm`)`
- `fd_vector_min_max(`Min`,` Max`)`
- `global_cardinality(`$List$`,` $Pairs$`)`
- `hcp(`Vs`,` Es`)`
- `hcp(`Vs`,` Es`,` K`)`
- `hcp_grid(`A`)`
- `hcp_grid(`A`,` Es`)`
- `hcp_grid(`A`,` Es`,` K`)`
- `increasing(`L`)`
- `increasing_strict(`L`)`
- `indomain(`Var`)` (nondet) (cp only)
- `indomain_down(`Var`)` (nondet) (cp only)
- `int_to_bv(`C`) =` Res (sat only)
- `lex_le(`L_1`,` L_2`)`
- `lex_lt(`L_1`,` L_2`)`
- `matrix_element(`$Matrix$`,` I`,` J`,` V`)`
- `matrix_element0(`$Matrix$`,` I`,` J`,` V`)`
- `neqs(`$NeqList$`)` (cp only)
- `new_bv(`N`) =` Res (sat only)
- `new_dvar() =` $FDVar$
- `nvalue(`N`,` $List$`)`
- `path(`Vs`,` Es`,` Src`,` $Dest$`)`
- `path_d(`Vs`,` Es`,` Src`,` $Dest$`)`
- `regular(`L`,` Q`,` S`,` M`,` $Q0$`,` F`)`
- `reset_store`
- `scalar_product(`A`,` X`,` $Product$`)`
- `scalar_product(`A`,` X`,` Rel`,` $Product$`)`
- `scc(`Vs`,` Es`)`
- `scc(`Vs`,` Es`,` K`)`
- `scc_d(`Vs`,` Es`)`
- `scc_d(`Vs`,` Es`,` K`)`
- `scc_grid(`A`)`
- `scc_grid(`A`,` K`)`
- `serialized(`$Starts$`,` $Durations$`)`
- `solve(`$Opts$`,` $Vars$`)` (nondet)
- `solve(`$Vars$`)` (nondet)
- `solve_all(`$Opts$`,` $Vars$`) =` $List$
- `solve_all(`$Vars$`) =` $List$
- `solve_suspended(`$Opts$`)` (cp only)
- `solve_suspended` (cp only)
- `subcircuit(`$FDVars$`)`
- `subcircuit_grid(`A`)`
- `subcircuit_grid(`A`,` K`)`
- `table_in(`$DVars$`,` R`)`
- `table_notin(`$DVars$`,` R`)`
- `tree(`Vs`,` Es`)`
- `tree(`Vs`,` Es`,` K`)`

Module `planner`

- best_plan(*S*, *Limit*, *Plan*)
- best_plan(*S*, *Limit*, *Plan*, *Cost*)
- best_plan(*S*, *Plan*)
- best_plan(*S*, *Plan*, *Cost*)
- best_plan_bb(*S*, *Limit*, *Plan*)
- best_plan_bb(*S*, *Limit*, *Plan*, *Cost*)
- best_plan_bb(*S*, *Plan*)
- best_plan_bb(*S*, *Plan*, *Cost*)
- best_plan_bin(*S*, *Limit*, *Plan*)
- best_plan_bin(*S*, *Limit*, *Plan*, *Cost*)
- best_plan_bin(*S*, *Plan*)
- best_plan_bin(*S*, *Plan*, *Cost*)
- best_plan_nondet(*S*, *Limit*, *Plan*) (nondet)
- best_plan_nondet(*S*, *Limit*, *Plan*, *Cost*) (nondet)
- best_plan_nondet(*S*, *Plan*) (nondet)
- best_plan_nondet(*S*, *Plan*, *Cost*) (nondet)
- best_plan_unbounded(*S*, *Limit*, *Plan*)
- best_plan_unbounded(*S*, *Limit*, *Plan*, *Cost*)
- best_plan_unbounded(*S*, *Plan*)
- best_plan_unbounded(*S*, *Plan*, *Cost*)
- current_plan() = *Plan*
- current_resource() = *Amount*
- current_resource_plan_cost(*Amount*, *Plan*, *Cost*)
- is_tabled_state(*S*)
- plan(*S*, *Limit*, *Plan*)
- plan(*S*, *Limit*, *Plan*, *Cost*)
- plan(*S*, *Plan*)
- plan(*S*, *Plan*, *Cost*)
- plan_unbounded(*S*, *Limit*, *Plan*)
- plan_unbounded(*S*, *Limit*, *Plan*, *Cost*)
- plan_unbounded(*S*, *Plan*)
- plan_unbounded(*S*, *Plan*, *Cost*)

Module `nn` (Neural Networks)

- new_nn(*Layers*) = *NN*
- new_sparse_nn(*Layers*) = *NN*
- new_sparse_nn(*Layers*, *Rate*) = *NN*
- new_standard_nn(*Layers*) = *NN*
- nn_destroy(*NN*)
- nn_destroy_all
- nn_load(*File*) = *NN*
- nn_print(*NN*)
- nn_run(*NN*, *Input*) = *Output*
- nn_run(*NN*, *Input*, *Opts*) = *Output*
- nn_save(*NN*, *File*)
- nn_set_activation_function_hidden(*NN*, *Func*)
- nn_set_activation_function_layer(*NN*, *Func*, *Layer*)
- nn_set_activation_function_output(*NN*, *Func*)
- nn_set_activation_steepness_hidden(*NN*, *Steepness*)
- nn_set_activation_steepness_layer(*NN*, *Steepness*, *Layer*)
- nn_set_activation_steepness_output(*NN*, *Steepness*)
- nn_train(*NN*, *Data*)
- nn_train(*NN*, *Data*, *Opts*)
- nn_train_data_get(*Data*, *I*) = *Pair*
- nn_train_data_load(*File*) = *Data*
- nn_train_data_save(*Data*, *File*)
- nn_train_data_size(*Data*) = *Size*

Module `datetime`

- current_date() = *Date*
- current_datetime() = *DateTime*
- current_day() = *WDay*
- current_time() = *Time*

Module `sys` (imported by default)

- abort
- cl
- cl(*File*)
- cl_facts(*Facts*)
- cl_facts(*Facts*, *IndexInfo*)
- cl_facts_table(*Facts*)
- cl_facts_table(*Facts*, *IndexInfo*)
- command(*String*) = *Int*
- compile(*File*)
- compile_bp(*File*)
- debug
- exit
- garbage_collect
- garbage_collect(Size)
- halt
- help
- initialize_table
- load(*File*)
- loaded_modules() = *List*
- nodebug
- nolog
- nospy
- notrace
- picat_path() = *String*
- spy *Functor*
- statistics
- statistics(*Key*, *Value*) (nondet)
- statistics_all() = *List*
- time(*Goal*)
- time2(*Goal*)
- time_out(*Goal*, *Limit*, *Result*)
- trace

Module `util`

- `array_matrix_to_list(`*AMatrix*`)` = *List*
- `array_matrix_to_list_matrix(`*AMatrix*`)` = *LMatrix*
- `chunks_of(`*List*`, `*K*`)` = *ListOfLists*
- `columns(`*Matrix*`)` = *List*
- `diagonal1(`*Matrix*`)` = *List*
- `diagonal2(`*Matrix*`)` = *List*
- `drop(`*List*`, `*K*`)` = *ResList*
- `find(`*String*`, `*SubString*`, `*From*`, `*To*`)` (nondet)
- `find_first_of(`*Term*`, `*Pattern*`)` = *Index*
- `find_ignore_case(`*String*`, `*SubString*`, `*From*`, `*To*`)` (nondet)
- `find_last_of(`*Term*`, `*Pattern*`)` = *Index*
- `join(`*Tokens*`)` = *String*
- `join(`*Tokens*`, `*Separator*`)` = *String*
- `list_matrix_to_array_matrix(`*LMatrix*`)` = *AMatrix*
- `lstrip(`*List*`)` = *ResList*
- `lstrip(`*List*`, `*Elms*`)` = *ResList*
- `matrix_multi(`*MatrixA*`, `*MatrixB*`)` = *MatrixC*
- `nextto(`*E1*`, `*E2*`, `*List*`)` (nondet)
- `permutation(`*List*`, `*Perm*`)` (nondet)
- `permutations(`*List*`)` = *ResLists*
- `power_set(`*List*`)` = *ResLists*
- `replace(`*Term*`, `*Old*`, `*New*`)` = *NewTerm*
- `replace_at(`*Term*`, `*Index*`, `*New*`)` = *NewTerm*
- `rows(`*Matrix*`)` = *List*
- `rstrip(`*List*`)` = *ResList*
- `rstrip(`*List*`, `*Elms*`)` = *ResList*
- `split(`*String*`)` = *Tokens*
- `split(`*String*`, `*Separators*`)` = *Tokens*
- `strip(`*List*`)` = *ResList*
- `strip(`*List*`, `*Elms*`)` = *ResList*
- `take(`*List*`, `*K*`)` = *ResList*
- `transpose(`*Matrix*`)` = *Transposed*

Index

abort/0, 23, 131
abs/1, 124
acos/1, 127
acosh/1, 128
acot/1, 127
acoth/1, 128
acsc/1, 127
acsch/1, 128
action/4, 69
acyclic/2, 103
acyclic_d/2, 103
acyclic_term/1, 44
all_different/1, 103
all_different_except_0/1, 103
all_distinct/1, 103, 107
and_to_list/1, 44
any-port, 17, 92, 93
append/3, 33, 48
append/4, 33
apply, 13, 15, 43
arg/3, 37
arity/1, 26, 37
array/1, 37
array_matrix_to_list/1, 137
array_matrix_to_list_matrix/1, 137
ascii_alpha/1, 29
ascii_alpha_digit/1, 29
ascii_digit/1, 29
ascii_lowercase/1, 29
ascii_uppercase/1, 29
asec/1, 127
asech/1, 128
asin/1, 126
asinh/1, 128
assignment/2, 103
at_end_of_stream/1, 88
at_least/3, 104
at_most/3, 104
atan/1, 127
atan2/2, 127
atanh/1, 128
atom/1, 29
atom_chars/1, 29
atom_codes/1, 29
atomic/1, 29
attr_var/1, 28
avg/1, 33, 38
best_plan/2, 70
best_plan/3, 70
best_plan/4, 70
best_plan_bb/2, 71
best_plan_bb/3, 71
best_plan_bb/4, 70
best_plan_bin/2, 71
best_plan_bin/3, 71
best_plan_bin/4, 71
best_plan_nondet/2, 72
best_plan_nondet/3, 72
best_plan_nondet/4, 72
best_plan_unbounded/2, 72
best_plan_unbounded/3, 72
best_plan_unbounded/4, 72
between/3, 30
between/4, 30
bigint/1, 30
bind_vars/2, 41
binom/2, 129
bool_dvar/1, 28
bound-port, 17, 92, 93
break, 8, 56
bv_add/3, 109
bv_and/3, 109
bv_div/3, 109
bv_drop/2, 109
bv_eq/2, 109
bv_ge/2, 109
bv_gt/2, 109
bv_le/2, 109
bv_lt/2, 109
bv_mod/3, 109
bv_mul/3, 110
bv_neq/2, 110

bv_or/3, 110
bv_pow/3, 110
bv_sum/2, 110
bv_take/2, 110
bv_to_int/1, 110
bv_xor/3, 110
call_cleanup/2, 13, 17, 43, 63
call, 13, 43
catch/3, 13, 17, 43, 63
cbc, 112
cd/1, 121
ceiling/1, 125
char/1, 29
chdir/1, 121
chr/1, 29, 100
chunks_of/2, 136
circuit/1, 104, 108
cl/0, 22, 131
cl/1, 22, 76, 130
cl_facts/1, 131
cl_facts/2, 131
cl_facts_table/1, 131
cl_facts_table/2, 131
clear/1, 39
close/1, 91
columns/1, 137
command/1, 134
compare_terms/2, 45
compile/1, 22, 130
compile_bp/1, 130
compound/1, 33
cond, 8, 42
copy_term/1, 26
copy_term_shallow/1, 26
cos/1, 126
cosh/1, 127
cot/1, 126
coth/1, 128
count/3, 104
count/4, 104
count_all/1, 13, 43
cp/2, 121
csc/1, 126
csch/1, 128
cumulative/4, 104, 108
current_date/0, 140
current_datetime/0, 140
current_day/0, 140
current_plan/0, 71
current_resource/0, 71
current_resource_plan_cost/3, 71
current_time/0, 140
cvc4, 113
cwd/0, 121
debug/0, 24, 131
decreasing/1, 104
decreasing_strict/1, 104
del/2, 39
delete/2, 33, 139
delete_all/2, 33
diagonal1/1, 137
diagonal2/1, 137
different_terms/2, 45, 95
diffn/1, 104
digit/1, 29
dir/0, 121
directory/1, 121, 122
disjoint/2, 139
disjunctive_tasks/1, 105
dom-port, 17, 92, 93
drop/2, 136
dump, 112
dvar/1, 28
dvar_or_int/1, 28
element/3, 105
element0/3, 105
end_of_file, 88
env_exists/1, 123
even/1, 129
exactly/3, 105
executable/1, 122
exists/1, 122
exit/0, 21, 135
exp/1, 125
ext, 112
e, 124
fail, 5, 19, 50
false, 5, 50
fd_degree/1, 98
fd_disjoint/2, 99
fd_dom/1, 99
fd_false/2, 99
fd_max/1, 99
fd_min/1, 99
fd_min_max/3, 99
fd_next/2, 99
fd_prev/2, 99
fd_set_false/2, 99

fd_size/1, 99
fd_true/2, 99
fd_vector_min_max/2, 98
file/1, 122
file_base_name/1, 122
file_directory_name/1, 122
file_exists/1, 122
final/1, 69
final/3, 69
find/4, 136
find_all/2, 13, 43
find_first_of/2, 136
find_ignore_case/4, 136
find_last_of/2, 136
findall/2, 13, 15, 43
findall, 78
first/1, 33, 38
flatten/1, 34
float/1, 30, 32
floor/1, 125
flush/1, 91
fold/3, 43
frand/0, 129
frand/2, 129
freeze/2, 13, 43, 95
functor/3, 37
garbage_collect/0, 135
garbage_collect/1, 135
gcd/2, 129
get/2, 3, 18, 39
get/3, 39
get_attr/2, 3, 28
get_attr/3, 28
get_global_map/0, 18, 45
get_global_map/1, 18, 45
get_heap_map/0, 18, 45
get_heap_map/1, 18, 45
get_table_map/0, 18, 45
get_table_map/1, 18, 45
getenv/1, 123
global_cardinality/2, 105
glpk, 112
ground/1, 45
gurobi, 112
halt/0, 1, 21, 135
has_key/2, 3, 39
hash_code/1, 26
hcp/2, 105
hcp/3, 106
hcp_grid/1, 106
hcp_grid/2, 106
hcp_grid/3, 106
head/1, 34
heap_is_empty/1, 40
heap_pop/1, 40
heap_push/2, 40
heap_size/1, 40
heap_to_list/1, 40
heap_top/1, 40
help/0, 1, 134
heuristic/1, 69
import, 12, 76
include directive, 21, 76
increasing/1, 106
increasing_strict/1, 106
index, 7
indomain/1, 110
indomain_down/1, 110
initialize_table/0, 68
insert/2, 139
insert/3, 34
insert_all/3, 34
insert_ordered/2, 34
insert_ordered_down/2, 34
ins-port, 17, 92, 93, 95
int/1, 30
int_to_bv/1, 110
integer/1, 30, 93
intersection/2, 139
is/2, 42
is_tabled_state/1, 71
join/1, 136
join/2, 136
keys/1, 39
last/1, 34, 38
len/1, 29, 34, 37, 38
length/1, 29, 33, 34, 38, 88
lex_le/2, 106
lex_lt/2, 107
list/1, 34
list_matrix_to_array_matrix/1, 137
list_to_and/1, 45
listdir/1, 121
load/1, 12, 22, 76, 130
loaded_modules/0, 78, 134
log/1, 125
log/2, 126
log10/1, 125

log2/1, 126
lowercase/1, 29
ls/0, 121
lstrip/1, 137
lstrip/2, 137
map/1, 39
map/2, 43
map/3, 43
map_to_list/1, 39
matrix_element/4, 107
matrix_element0/4, 107
matrix_multi/2, 137
max/1, 34, 38
max/2, 30, 52
maxint_small/0, 32
maxof/2, 13, 44
maxof/3, 13, 44
maxof_inc/2, 13, 44
maxof_inc/3, 13, 44
max, 65
membchk/2, 34
member/2, 6, 34, 78
min/1, 34, 38
min/2, 32, 52
minint_small/0, 32
minof/2, 13, 44
minof/3, 13, 44
minof_inc/2, 13, 44
minof_inc/3, 13, 44
min, 65
mkdir/1, 121
mmax, 65
mmin, 65
modf/1, 125
module, 11, 76
name/1, 26, 37
neqs/1, 107
new_array, 2, 38
new_bv/1, 110
new_dvar/0, 99
new_list/1, 34
new_list/2, 34
new_map/0, 39
new_map/1, 2, 39
new_map/2, 39
new_max_heap/1, 40
new_min_heap/1, 40
new_nn/1, 115
new_ordset/1, 139
new_set/0, 39
new_set/1, 2, 39
new_set/2, 39
new_sparse_nn/1, 116
new_sparse_nn/2, 116
new_standard_nn/1, 115
new_struct/2, 2, 37
nextto/3, 138
nl/0, 90
nl/1, 90
nn_destroy/1, 116
nn_destroy_all/0, 116
nn_load/1, 118
nn_print/1, 116
nn_run/2, 119
nn_run/3, 119
nn_save/2, 118
nn_set_activation_.../2, 117
nn_set_activation_.../3, 116, 117
nn_train/2, 118
nn_train/3, 118
nn_train_data_get/2, 117
nn_train_data_load/1, 117
nn_train_data_save/2, 117
nn_train_data_size/1, 117
nodebug/0, 24, 131
nolog/0, 134
nonvar/1, 28
nospy/0, 131
not/1, 13, 26, 50
notrace/0, 24, 131
nth/3, 34, 38
number/1, 32
number_chars/1, 32
number_codes/1, 32
number_vars/1, 45
number_vars/2, 45
nvalue/2, 107
odd/1, 129
once/1, 6, 13, 26, 47, 50, 93, 95, 134
open/1, 63, 85, 86, 89, 91
open/2, 85, 86, 89, 91
ord/1, 30, 100
ordset/1, 139
parse_radix_string/2, 45
parse_term/1, 45
parse_term/3, 45
path/4, 107
path_d/4, 107

peek_byte/1, 88
peek_char/1, 88
permutation/2, 138
permutations/1, 138
picat_path/0, 134
picat, 1, 20
pi, 124
plan/2, 70
plan/3, 70
plan/4, 70
plan_unbounded/2, 72
plan_unbounded/3, 72
plan_unbounded/4, 72
post_event/2, 17, 92
post_event_any/2, 92
post_event_bound/1, 92
post_event_dom/2, 92
post_event_ins/1, 92
pow/2, 125
pow_mod/3, 125
power_set/1, 137
prime/1, 129
primes/1, 129
print/1, 90
print/2, 90
printf, 90, 141
println/1, 90
println/2, 90
prod/1, 34, 38
put/2, 39
put/3, 3, 18, 39
put_attr/2, 28
put_attr/3, 3, 28
pwd/0, 121
rand_max/0, 128
random/0, 128
random/1, 128
random/2, 128
random2/0, 128
read_atom/0, 86
read_atom/1, 86
read_byte/0, 86
read_byte/1, 86
read_byte/2, 86, 88
read_char/0, 86
read_char/1, 86
read_char/2, 86, 88
read_char_code/0, 86
read_char_code/1, 86
read_char_code/2, 86, 88
read_file_bytes/0, 86
read_file_bytes/1, 86
read_file_chars/0, 86
read_file_chars/1, 86
read_file_codes/0, 86
read_file_codes/1, 86
read_file_lines/0, 87
read_file_lines/1, 87
read_file_terms/0, 87
read_file_terms/1, 87
read_int/0, 62, 87
read_int/1, 87
read_line/0, 87
read_line/1, 87, 88
read_number/0, 87
read_number/1, 87
read_picat_token/0, 87
read_picat_token/1, 87
read_picat_token/2, 87
read_picat_token/3, 87
read_real/0, 87
read_real/1, 87
read_term/0, 88
read_term/1, 88
readable/1, 122
readln/0, 87
readln/1, 87
real/1, 32
reduce/2, 44
reduce/3, 44
regular/6, 107
remove_dups/1, 35
rename/2, 121
repeat/0, 50
replace/3, 136
replace_at/3, 136
reset_store/0, 113
reverse/1, 35, 38
rm/1, 121
rmdir/1, 121
round/1, 125
rows/1, 138
rstrip/1, 137
rstrip/2, 137
scalar_product/3, 107
scalar_product/4, 107
scc/2, 107
scc/3, 108

scc_d/2, 108
scc_d/3, 108
scc_grid/1, 108
scc_grid/2, 108
scip, 112
sec/1, 126
sech/1, 127
second/1, 45
select/3, 35
separator/0, 120
sequence/2, 69
serialized/2, 108
sign/1, 124
sin/1, 126
sinh/1, 127
size/1, 39, 122
slice/2, 35, 38
slice/3, 35, 38
solve/1, 15, 97, 98, 110
solve/2, 15, 97, 98, 110
solve_all/1, 110
solve_all/2, 110
solve_suspended/0, 110
solve_suspended/1, 110
sort/1, 35, 38
sort/2, 35, 38
sort_down/1, 35, 38
sort_down/2, 35
sort_down_remove_dups/1, 35, 38
sort_down_remove_dups/2, 36
sort_remove_dups/1, 35, 38
sort_remove_dups/2, 35, 38
sorted/1, 36, 38
sorted_down/1, 36, 38
split/1, 137
split/2, 137
spy/1, 25, 131
sqrt/1, 125
statistics/0, 132
statistics/2, 132, 133
statistics_all/0, 133
stderr, 91
stdin, 91
stdout, 91
string/1, 36
strip/1, 137
strip/2, 137
struct/1, 37
subcircuit/1, 108
subcircuit_grid/1, 109
subcircuit_grid/2, 109
subset/2, 139
subsumes/2, 45
subtract/2, 139
sum/1, 36, 38
table_in/2, 99
table_notin/2, 99
table, 11, 64
tail/1, 36
take/2, 137
tan/1, 126
tanh/1, 127
throw/1, 6, 16, 51, 63
time/1, 13, 134
time2/1, 13, 134
time_out/3, 13, 134
to_array/1, 36
to_atom/1, 30
to_binary_string/1, 32
to_codes/1, 26, 32
to_degrees/1, 126
to_float/1, 32
to_fstring, 28, 32, 141
to_hex_string/1, 32
to_int/1, 32
to_integer/1, 32
to_list/1, 37
to_lowercase/1, 36
to_number/1, 32
to_oct_string/1, 32
to_radians/1, 126
to_radix_string/2, 32
to_real/1, 32
to_string/1, 28
to_uppercase/1, 36
trace/0, 24, 131
transpose/1, 138
tree/2, 109
tree/3, 109
true, 5, 50, 54, 59
truncate/1, 32, 125
union/2, 139
uppercase/1, 30
values/1, 39
var/1, 17, 28, 93, 95
variant/2, 46
vars/1, 46
writable/1, 122

write/1, 8, 53, 89, 90
write/2, 89, 90
write_byte/1, 89
write_byte/2, 89
write_char/1, 89
write_char/2, 89
write_char_code/1, 89
write_char_code/2, 89
writef, 89, 90, 141
writeln/1, 89
writeln/2, 89
z3, 113
zip, 36, 55
=../2, 37
=/2, 41
=:=/2, 41
==/2, 40
=\=/2, 42
\+/1, 13, 50

accumulator, 51, 52, 59, 60
action rule, 17, 92–95, 103
activation function, 114
anonymous definition
    implementation, 83
anonymous function, 14, 80
anonymous predicate, 14, 81
anonymous variable, 28
append mode, 85
arity, 1, 5, 26, 36, 37, 39, 47
array, 1, 2, 26, 36, 37
array comprehension, 3, 58
as-pattern, 49
assignment, 8, 53, 54, 59
atom, 1, 12, 13, 26, 29, 36
attributed variable, 1, 26, 28, 41

backtrackable rule, 5, 6, 17, 47, 48

call trace, 24, 25
car, 33, 40, 49
cdr, 33, 40
closure, 81
complete list, 33
compound value, 1, 3, 8, 26, 33, 53–55, 58
cons, 33, 50, 52
constraint, 15, 16, 97
context variable, 81
cut, 51

DCG rule, 47
debugging, 1
Definite Clause Grammar, 47
directed graph, 103, 105, 107, 108
do-while loop, 9, 54, 57, 58, 60
don't-care symbol, 99

environment variable, 22
exception, 5, 7, 16, 62, 63
execution trace, 24
extensional constraint, 99

failure-driven loop, 50, 55
feedforward neural network, 114
file descriptor, 85, 86, 88, 89, 91
file descriptor table, 85, 91
file name, 22, 76
first-fail principle, 98, 111
foreach loop, 5, 8, 10, 16, 54–58, 60
free variable, 1, 7, 28, 37
function, 43, 47–49, 51
function fact, 7, 8, 48, 78
functor, 36, 39, 40

garbage collector, 132, 135
global map, 18
global module, 12, 76
goal, 5, 47, 50, 51
grid graph, 106, 108, 109
ground, 45

Hamiltonian cycle, 104, 105, 109
handler, 63
hard link, 121
heap, 1, 2, 26, 40
heap map, 18
higher-order call, 3, 12, 13, 43, 78, 79
    module resolution, 83
Horn clause, 6, 47

if statement, 5, 6, 51, 54
imperative, 53
index declaration, 7, 8, 11
instantiated variable, 17, 28, 34, 41, 45
integer, 1, 3, 26, 30, 32
interrupt, 16, 62
iterator, 3, 8, 54–56, 58, 60

lambda expression, 14, 80
last-call optimization, 51
linear tabling, 68
list, 1, 26, 29, 32–34, 36, 37, 39, 40, 46

list comprehension, 3, 10, 53, 58, 60, 61
local variable, 9, 16, 58, 60

map, 1–3, 18, 26, 28, 39, 45
map-set, 2
max-heap, 2, 40
min-heap, 2, 40
mode-directed tabling, 11, 65
module file, 12, 22

name scope, 81
nested loop, 56, 60
neural network, 114
non-backtrackable rule, 5, 7, 17, 47, 48
non-trace mode, 24, 131
number, 1, 26, 29, 30, 32

occurs-check problem, 41

planner, 69
planning, 69
prebuilt map, 18
predicate, 5, 43, 47–51
primitive value, 1, 26, 40
Prolog, 6, 47

read mode, 85

scope, 9, 16, 53
set, 1, 2, 26, 39
single-assignment, 28, 53
spy mode, 24, 131
spy point, 25
standalone, 24
string, 1, 26, 29, 32, 36
structure, 1, 26, 33, 36, 37, 39, 40
symbol table, 77
symbolic link, 121, 122

table constraint, 99
table map, 18
tabling, 11, 18, 64, 65, 68
tail recursion, 10, 48, 49, 51, 59
term, 1
trace mode, 24, 25, 131

undirected graph, 103, 107, 109

while loop, 5, 8, 54, 56, 57, 59, 60
write mode, 85