Picat: A Scalable Logic-based Language and System

Neng-Fa Zhou and Jonathan Fruhman
The City University of New York

Last updated April 9, 2013

Preface

Despite the elegant concepts, new extensions (e.g., tabling and constraints), and successful applications (e.g., knowledge engineering, NLP, and search problems), Prolog has a bad reputation for being old and difficult. Many ordinary programmers find the implicit non-directionality and non-determinism of Prolog to be hard to follow, and the non-logical features, such as cuts and dynamic predicates, are prone to misuses, leading to absurd codes. The lack of language constructs and libraries for programming everyday things is also considered a big weakness of Prolog. The backward compatibility requirement has made it hopeless to remedy the language issues in current Prolog systems, and there are urgent calls for a new language.

Several successors of Prolog have been designed, including Mercury, Erlang, Oz, and Curry. The requirement of many kinds of declarations in Mercury has made the language difficult to use; Erlang's abandonment of non-determinism in favor of concurrency has made the language unsuited for many applications despite its success in the telecom industry; Oz has never attained the popularity that the designers sought, probably due to its unfamiliar syntax and implicit laziness; Curry is considered too close to Haskell. All of these successors were designed in the 1990s, and now the time is ripe for a new logic-based language.

This project aims to design and implement a simple, and yet powerful, logic-based programming language, named Picat, for a variety of applications. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, constraints, and tabling. Picat lacks Prolog's non-logical features, such as the cut operator and dynamic predicates, making Picat more reliable than Prolog. Picat also provides imperative language constructs for programming everyday things. The resulting system will be used for not only symbolic computations, which is a traditional application domain of declarative languages, but also for scripting tasks for the Web, games, and mobile applications.

Picat is a general-purpose language that incorporates features from logic programming, functional programming, and scripting languages. The following Picat list of pairs tells the meanings of the letters in the name.

```
['P'=predicates, 'I'=imperative, 'C'=constraints, 'A'=actors, 'T'=tabling]
```

- **P**redicates: A *predicate* defines a relation, and can have zero, one, or multiple answers. A *function* is a special kind of a predicate that always succeeds with *one* answer. Picat is a rule-based language. Predicates and functions are defined with pattern-matching rules.

- **I**mperative: Picat incorporates features of imperative programming languages, which tell the computer *how* to perform operations. Picat provides assignment and loop statements for programming everyday things.

- **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.

- **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. Picat treats threads as channels, and allows the use of action rules to program concurrent threads.

- **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.

In addition to these language features, the Picat system also provides external interfaces with other language and database systems, and a rich set of library modules including threads, sockets, and Web services.

Contents

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`. Once the interpreter is started, users can type a command line after the prompt `picat>`. The `help` command shows the usages of commands, and the `halt` command terminates the Picat interpreter. The OS command `picatc` compiles a file (with extension name `pi`) and all of its dependent files into bytecode files (with extension name `qi`), and the OS command `picate` executes a Picat program as a standalone application.

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 the 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 an identifier that begins with a lower-case letter or a single-quoted sequence of characters.

A compound value can be a *list* in the form `[`t_1`,`$\ldots$`,`t_n`]` or a *structure* in the form `$`s`(`t_1`,`$\ldots$`,`t_n`)$` 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 two dollar symbols are used to distinguish a structure from a function call. Strings, arrays, and maps are special compound values. A *string* is a list of single-character atoms. An *array* takes the form `{`t_1`,`$\ldots$`,`t_n`}`, which 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.

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. 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 = 1e10    % numbers
```

```
picat> A1 = x1, A2 = '_AB', A3 = ''     % atoms

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

picat> write("hello"++"picat")          % strings
"hellopicat"

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

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

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(["one"=1,"two"=2])

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 enclosed in a pair of dollar symbols in order for them to be treated as data. Without the dollar symbols, 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, list comprehensions, and loops.

For each type, Picat provides a set of built-in functions and predicates. The index notation X`[`I`]`, where X references a compound value and I is an integer expression, is a special function that returns a single component 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.

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

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

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 combination 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.

The predicate `put(`X, Key, Val`)` attaches the key-value pair Key=Val to X, where X is either a variable or a map, Key is a non-variable term, and Val is any term. An attributed variable has a map attached to it. The function `get(`X, Key`)` returns Val of the key-value pair Key=Val attached to X. The predicate `has_key(`X, Key`)` returns true iff X contains a pair with the given key.

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(X,one,1), One = get(X,one)  % attributed variable
One = 1

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

picat> S = new_array(2,3), S[1,1] = 11, S[2,3] = 23, D2 = length(S[2])
S = {{11,_3d4,_3d8},{_3e0,_3e4,23}}
D2 = 3

picat> M = new_map(), put(M,"one",1), One = get(M,"one")
One = 1
```

Picat also allows OOP notations for accessing attributes and for calling predicates and func-

tions. 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 a module name, in which case A_1 is a module qualifier for f. The notation $A.Attr$ is the same as the function call `get`$(A, Attr)$. A structure is assumed to have two attributes called `name` and `length`.

Example

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

picat> Len = [a,b,c,d].length()
Len = 4

picat> X.put(one,1), One = X.one
One = 1

picat> X = math.pi
X=3.14159

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

picat> S = new_array(2,3), S[1,1] = 11, S[2,3] = 23, D2 = S[2].length
S = {{11,_3d4,_3d8},{_3e0,_3e4,23}}
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` always fails. A *goal* is made from predicate calls and statements, including conjunction (A, B), disjunction $(A; B)$, negation (not A), if-then-else, `foreach` loops, and `while` loops.

A predicate is defined with pattern-matching rules. Picat has two types of rules: the non-backtrackable rule $Head, Cond$ `=>` $Body$, and the 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 if-then-else as follows:

```
fib(N,F) =>
    if (N=0; N=1) then
        F=1
    elseif N>1 then
        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` $Cond$ `then` $Goal_1$ `else` $Goal_2$ `end`. The `then` part 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 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.

Picat allows inclusion of *predicate facts* in the form p`(`t_1`,...,`t_n`)` in predicate definitions. Facts are translated into pattern-matching rules before they are compiled. A predicate definition that consists of facts can be preceded by an *index declaration* in the form `index` `(`$M_{11}, M_{12}, \ldots, M_{1n}$`)` ... `(`$M_{m1}, M_{m2}, \ldots, M_{mn}$`)` where each M_{ij} is either $+$ (meaning indexed) or $-$ (meaning not indexed). For each index pattern `(`$M_{i1}, M_{i2}, \ldots, M_{in}$`)`, the compiler generates a version of the predicate that indexes all of the $+$ arguments.

Example

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

For a predicate of indexed facts, a matching version of the predicate is selected for a call. If no matching version is available, Picat throws an exception. For example, for the call `edge(X,Y)`, if both `X` and `Y` are free, then no version of the predicate matches this call and Picat throws an exception. If predicate facts are not preceded by any index declaration, then no argument is indexed.

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`,`$\ldots$`,`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 with parentheses, unless the function is module-quantified, 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 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).
```

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 (`E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`)`
  $Goal$
`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 $E_1 \in D_1$, ..., $E_n \in D_n$ that satisfies the conditions $Cond_1$, ..., $Cond_n$. A while statement takes the form

`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 $Cond$.

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]
...
L_n=[e_n|L_{n-1}]
List=L_n

Note that the list of integers is in reversed 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]
...
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)=F,N>1 => F=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, and `min` indicates that the corresponding variable should be minimized.

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 editing distance between the two lists. The table mode `table(+,+,min)` tells the system to keep a fact with the minimal editing 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` $Name$.

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 either a module name or $M.S$, where M is a module name, and S is a predicate or function symbol. 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. 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.

1.7 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 three solver modules, including `cp`, `sat`, and `mip`.

Example

```
import cp.

go =>
    Vars=[S,E,N,D,M,O,R,Y],  % generate variables
    Vars in 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 in 1..N,
    foreach(Row in 1..N)
        all_different([A[Row,Col] : Col in 1..N])
    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..J*M],
        all_different(Square)
    end,
    solve(A),
    foreach(I in 1..N) write(A[I]) end.

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

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.

1.8 Exceptions

An exception is an event that occurs during the execution of a program that requires a special treatment. In Picat, an exception is just a term. Example exceptions include `divide_by_zero(`$Source$`)`, `file_not_found(`$Name$`,`$Source$`)`, `number_expected(`$ExArg$`,`$Source$`)`, `interrupt(`$Source$`)`, and `out_of_range(`$ExArg$`,`$Source$`)`. The exception `interrupt(keyboard)` is raised when `ctrl-c` is typed during a program's execution. The built-in predicate `throw` $Exception$ throws $Exception$.

The `try` statement, which takes the following form, is provided for catching and handling exceptions.

```
try
   Goal
catch (Pattern_1)
   Handler_1
```

$\vdots$
`catch (`$Pattern_n$`)`
  $Handler_n$
`finally`
  $Handler_{fin}$
`end`

where each $Pattern_i$ is a term pattern like the head of a rule, and each $Handler_i$ is a goal. For an exception, a catch clause is said to be applicable if the exception matches the pattern of the clause. When an exception is thrown during the execution of $Goal$, the system searches for the first applicable catch clause and executes the handler. If no catch clause is applicable to the exception, then $Handler_{fin}$ is executed. The `finally` block, which is optional, is also executed when $Goal$ ends normally, i.e., when $Goal$ fails or $Goal$ succeeds deterministically with no choice points left behind.

1.9 Higher-Order Calls

A predicate or function is said to be *higher-order* if it takes calls as arguments. The built-ins `call`, `apply`, and `findall` are higher-order. The predicate `call(`S`,`Arg_1`,...,`Arg_n`)`, where S is an atom or a structure, 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`,`Arg_1`,...,`Arg_n`)` returns a list of all possible solutions of `call(`S`,`Arg_1`,...,`Arg_n`)` in the form of $Template$.

Example

```
picat> S=$member(X)$, call(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> C=lambda([X,Y],X+Y), Z=apply(C,1,2)
Z=3
```

A lambda term in the form `lambda(`$VList$`,`Exp`)`, where $VList$ is a list of input variables, and Exp is an expression, denotes an anonymous function. The last command in the example is the same as `Z=apply(add,1,2)`, where `add` is a function defined as follows:

```
add(X,Y)=Z => Z=X+Y.
```

The meta-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])=[apply(F,X)|map(F,Xs)].
```

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

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

A call that is passed to a higher-order predicate or function is assumed to invoke a definition in the same module or an imported module. If the compiler cannot bind a call to a definition because the name is unknown, then it generates code to search the enclosing module, and the imported modules for a definition at runtime.

1.10 Action Rules and Threads

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:

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

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(Flag) => writeln(done).
```

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 `Flag=1` is inserted before `post_event(X,picat)`, 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`.

A thread is represented as an attributed variable that contains, among other attributes, a thread descriptor. A thread can serve as a communication channel. A thread can send a message to another thread by posting an event. Action rules can be used to program concurrent threads.

Example

```
import thread.

go =>
    EchoThread = new_thread(install_echo_actor),
    SenderThread = new_thread(send,3,EchoThread),
    EchoThread.start(),
    SenderThread.start().

install_echo_actor =>
    echo(this_thread(),Flag),
    loop(Flag).

echo(X,Flag),var(Flag),{event(X,T)} =>
    writeln(T),
    if (T==done) then Flag=1 end.
echo(_,_) => true.

loop(Flag),var(Flag) => loop(Flag).
loop(_) => true.

send(N,EchoThread) =>
    foreach(I in 1..N)
        post_event(EchoThread,hello)
    end,
    post_event(EchoThread,done).
```

The built-in function `new_thread(`S`,`Arg_1`,...,`Arg_n`)` creates a new thread to execute the call

`call(`S`,`Arg_1`,...,`Arg_n`).`

The built-in function `this_thread()` returns the executing thread of the function call. In this example, the `EchoThread` installs an actor, and then loops until `Flag` becomes a non-variable; the `SenderThread` sends `hello` to `EchoThread` three times, and then sends `done` to `EchoThread`, causing it to kill itself. After sending the messages, the `SenderThread` terminates.

1.11 Global Maps

Each thread has a map, called a *global heap map*, which is created on the heap immediately after the thread is created. The built-in function `get_heap_map()` returns the map that belongs to the current thread. A thread map is like a normal map. Users use `put` to add key-value pairs into a map. Users use `get` to retrieve a value that is associated with a key in the map. Changes to a map up to a choice point are undone when execution backtracks to that choice point.

The Picat system has a *global map* that is shared by all threads. The global map is created in the global area when the Picat system is started. The built-in function `get_global_map()` returns this map. A big difference between this 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 advantage of using global 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 global maps is discouraged because it may cause unanticipated effects.

Example

```
go ?=>
    get_heap_map().put(one,1),
    get_global_map().put(one,1),
    fail.
go =>
    if (get_heap_map().contains_key(one)) then
       writef("heap map has key%n")
    else
       writef("heap map has no key%n")
    end,
    if (get_global_map().contains_key(one)) then
       writef("global map has key%n")
    else
       writef("global map has no key%n")
    end.
```

For the call `go`, the output is `"heap map has no key"` followed by `"global map 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 remains.

1.12 External Language Interfaces and Libraries

Picat has well-defined interfaces with C and Java. The dynamic libraries (`dll` or `so` files) of the Picat system will be made available so that users can easily link Picat programs with external software components though C or Java. Picat also supports access to databases through the ODBC interface.

Picat provides a rich library of modules for applications such as language processing (Regix, DCG, etc.), Web services (server side programming, RIF data processing, semantic web applications, etc.), mobile applications for handheld devices, and game programming.

1.13 Resources

- An overview of Picat: `http://www.picat-lang.org/download/picat_proposal.pdf`
- Examples: `http://www.picat-lang.org/download/exs.pi.txt`
- Libraries: `http://www.picat-lang.org/download/builtin.pdf`
- Lexical grammar: `http://www.picat-lang.org/download/lex_grammar.txt`
- Syntax grammar: `http://www.picat-lang.org/download/syntax_grammar.txt`

1.14 Programming Exercises

Select five problems from the Euler Project problem set at `http://projecteuler.net/problems` and write a program in Picat for each of them.

Chapter 2

How to Use the Picat System

The environment variable `PICATDIR` should be set to store the full path of the directory in which Picat is installed. The Picat system is written in both C and Picat. The part that is written in C is compiled into the executable named `$PICATDIR/Emulator/picat` on Unix systems, or `%PICATDIR%\Emulator\picat.exe` on Windows. The part that is written in Picat is divided into files stored in `$PICATDIR/Compiler` and `$PICATDIR/Library`.

Picat source files have the extension name `pi`. A *module file* is a source file that begins with a module declaration. Not every source file is a module file. A module can be spread across multiple files. In that case, only the file `name.pi` will have the module declaration `module` $Name$. Furthermore, `name.pi` will have an `include` statement, listing the other files that contain parts of the module. For example, the file `basic.pi` has the following content:

```
module basic.
include "basic_list.pi", "basic_array.pi",
        "basic_map.pi", "basic_term.pi", "basic_io.pi".
```

The file `basic.pi` does not define any predicates or functions, but it includes a bunch of other source files. None of the included files has a `module` declaration.

The files with the extension names `qi` and `dbqi` are *bytecode files* that are generated from the module files. The Picat system can be run in either *debug* mode or *non-debug* mode. The `dbqi` files are used for debug mode, and the `qi` files are used for non-debug mode. When the Picat system is in debug mode, it allows debugging, and dumps the stack trace when it encounters an uncaught exception.

There are three executable script files in the directory `$PICATDIR`: `picat`, `picatc`, and `picate`. The script `picat` starts the Picat interpreter; the script `picatc` compiles Picat files; the script `picate` runs a Picat program as a standalone application.

2.1 How to Use the Picat Interpreter

The Picat system provides an interactive programming environment for users to 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 path must be properly set. In Unix, this can be done by adding the following line to the script file `.cshrc`, `.bshrc`, or `.kshrc` depending on the shell that is used:

```
alias   picat       $PICATDIR/picat
```

In Windows, Picat's root directory, `%PICATDIR`, must be added to the environment variable named `path`.

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.

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
```

Users can change the prompt by using the query `prompt(`$NewPrompt$`)`. For example, the command

```
prompt("?-")
```

changes the prompt to `?-`. The `help` predicate shows the usages of the main commands.

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 in 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. A file name can start with an environment variable, such as `$V` or `%V%`, which will be replaced by its value before the file name is actually processed. Picat treats both `'/'` and

`'\'` as file name 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 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 Picat library directory `$PICATDIR/Library`, in the current working directory, and in the directories that are stored in the environment variable `PICATPATH`. For $FileName$`.pi` and each dependent module file, the compiler creates a byte-code file with the same main name and the extension name `.qi` or `.dbqi`, depending on the current execution mode. If the mode is non-debug mode, then the extension name is `qi`; otherwise, if the mode is debug mode, then the extension name is `dbqi`. If the directory in which the source file resides is not writable, then no byte-code file will be created, and the byte code will be loaded directly into the system.

The built-in predicate `compile(`$FileName$`)` compiles the file $FileName$`.pi` and all of its dependent module files without loading the generated byte-code files. The destination directory for the byte-code 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.

The built-in predicate `load(`$FileName$`)` loads the byte-code file $FileName$`.qi` or $FileName$`.dbqi`, depending on the current execution mode, and all of its dependent byte-code files. For $FileName$ and its dependent file names, the system searches for a byte-code file in the Picat library directory `$PICATDIR/Library`, in the current working directory, and in the directories that are stored in the environment variable `PICATPATH`. If no byte-code file exists for a file name, then this built-in throws an exception.

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. Users can ask the system to find the next solution by typing `';'` after a solution. 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`.

2.2 How to Use the Debugger

There are two execution modes: *debug mode* and *non-debug mode*. When the Picat interpreter is started, it runs in non-debug mode. The predicate `debug` changes the mode to debug. In order

to enable the debugger to display debugging information for a program, users have to switch the mode to debug before loading the program. The predicate `nodebug` changes the mode to non-debug.

In debug 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 no answer. The information about a call includes the module name, the name of the call, and the arguments. If the call belongs to the global module or the basic module of the library, then the module name is not shown. 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. For a loop, the debugger displays its name, which can be `foreach`, `while`, `do-while`, or `list-comp`, when the loop is entered or exited.

Consider, for example, the following program:

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

The following shows a trace for a query:

```
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 debug 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 that users are spying. Users can use the predicate

`spy` $M.Name/N$

to set the functor $Name/N$ of module M as a spy point, where the module name M and arity N are optional. If no module name is given in a spy point, then the functor is assumed to belong to the global module, the basic module, or one of the currently imported modules. 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. The following debugging commands are accepted:

`RET`	creep, show the next call trace.
`c`	creep, same as `RET`.
`l`	leap, be silent until a spy point is encountered.
`s`	skip, be silent until the call is completed (Exit or Fail).
`r`	repeat, continue to creep or leap without intervention.
`a`	abort, quit debugging, moving control to the top level.
`h`	help, display the debugging commands.
`?`	help, same as `h`.
`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.
`<`	reset the print depth to 10.
`< d`	reset the print depth to d.

2.3 How to Use the `picate` and `picatc` Commands

The script file `picate` executes a Picat program as a standalone application. The command is used in the following way:

`picate -path` $Path_1$`;...;`$Path_n$ `-debug` $FileName$ Arg_1 ... Arg_m

where the option `path` specifies the paths where byte code files are searched, and the option `debug` tells the system to load `dbpi` byte code files rather than optimized `pi` files. Although users cannot use the debugger when running a standalone application in debug mode, users can view the stack trace in case an uncaught exception occurs during the execution. The command takes exactly one file name $FileName$. The $FileName$'s byte code file and all of its dependent byte code files will be loaded. The system will search for the byte code files in the paths that are specified in the command. If no path is given, then the system searches the paths that are included in the environment variable `$PICATPATH`. The command can also take several arguments after the file name. The file of $FileName$ must contain a predicate named `main(`$Args$`)`, where $Args$ is a list. All of the command arguments $Arg_1 \ldots Arg_m$ will be passed to the `main` predicate as a list of strings.

The script file `picatc` compiles Picat source files. The command is used in the following way:

`picatc -path` $Path_1$`;...;`$Path_n$ `-debug` $FileName_1$ ... $FileName_n$

where the options are the same as in the command `picate`. The command compiles all of the source files $FileName_1 \ldots FileName_n$ and all of their dependent files into byte code files. If the option `debug` is given, then the debuggable byte code files with the extension name `dbqi` are generated; otherwise, the optimized byte code files with the extension name `qi` are generated.

2.4 How to Use the Profiler

The built-in predicate `profile_src(`$FileName$`)`, where $FileNames$ is the main name of a source file, reports the following information about the source file and all of its dependent files:

- What predicates and functions are defined?
- What predicates and functions are used but not defined?
- What predicates and functions are defined but not used?
- What built-ins are used?

The predicate `profile_src` can be used in both debug and non-debug modes.

The built-in predicate `profile(`$Query$`)` prints the number of times that each predicate or function is called during the execution of $Query$. The reported statistics are helpful for fine-tunning programs for better performance. The predicate `profile` can be used only in debug mode with `dbpi` files.

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, and maps 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 disjunction (`;`), will be described in Chapter 4 on Predicates and Functions. The constraint operators (the ones that begin with `#`) will be described in Chapter 14 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 accessing attributes and 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 a module name, in which case A_1 is a module qualifier for f. If an atom happens to be the same as one of the imported module names, and the atom needs to be passed as the first argument to a function or a predicate, then this notation cannot be used. The notation $A.Attr$, where $Attr$ does not have the form `f(...)`, is the same as the function call `get`$(A, Attr)$. For example, the expression S`.name` returns the name, 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 `a.b().c()` is the same as `c(b(a))`, unless `a` is the name of an imported module.

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 the underscore. For example, the following are valid variable names:

```
X1    _    _ab
```

Table 3.1: Operators in Picat

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

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.
- `has_key(`X`, `Key`)`: This predicate is true if X has an attribute named Key.
- `keys(`X`) = `$List$: This function returns the list of names of the attributes of X.
- `get(`X`, `Key`) = `Val: This function returns the Val of the key-value pair Key=Val that is attached to X. It throws an error if X has no attribute named Key.
- `put(`X`, `Key`, `Val`)`: This predicate attaches the key-value pair Key=Val to X, where Key is a non-variable term, and Val is any term.
- `values(`X`) = `$List$: This function returns the list of values of the attributes of X.

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 lower-case 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'   '$%'
```

No atom name can last 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:

- `atom(`$Term$`)`: This predicate is true if $Term$ is an atom.
- `atomic(`$Term$`)`: This predicate is true if $Term$ is an atom or a number.
- `char_code(`$Char$`) = `Int: This function returns the code of the character $Char$. It throws an error if $Char$ is not a single-character atom.
- `atom_chars(`Atm`) = `$String$: This function returns 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.

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, rounded down
X `mod` Y	modulo, same as X - floor(X / Y) * Y
X `rem` Y	remainder (X - (X // Y) * Y)
X `+` Y	addition
X `-` Y	subtraction
X `>>` Y	right shift
X `>>>` Y	unsigned 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

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 optional integer part, an optional decimal fraction preceded by a decimal point, and an optional exponent. If an integer part exists, then it must be followed by either a fraction or an exponent in order to distinguish the real number from an integer literal. For example, the following are valid real numbers.

```
12.345   .123   12-e10   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 an integer, then this predicate determines whether X is between $From$ and To. Otherwise, if X is unbound, then this predicate nondeterministically selects X from the integers that are between $From$ and To. It is the same as `member(`X`, `$From$`..`To`)`.
- `number(`$Term$`)`: This predicate is true if $Term$ is a number.
- `integer(`$Term$`)`: This predicate is true if $Term$ is an integer.
- `float(`$Term$`)`: This predicate is true if $Term$ is a real number.
- `real(`$Term$`)`: This predicate is true if $Term$ is a real number. It is the same as `float(`$Term$`)`.
- `max(`X`, `Y`)` = Val: This function returns the maximum of X and Y.
- `min(`X`, `Y`)` = Val: This function returns the minimum of X and Y.
- `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 the same as `to_fstring("%f",`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()`.
- `to_binary_string(`Int`)` = $String$: This function returns the binary representation of the integer Int as a string.
- `to_oct_string(`Int`)` = $String$: This function returns the octal representation of the integer Int as a string.
- `to_hex_string(`Int`)` = $String$: This function returns the hexadecimal representation of the integer Int as a string.
- `to_integer(`Num`)` = Int: This function is the same as `truncate(`Num`)` in the `math` module.
- `to_real(`Num`)` = $Real$: This function is the same as Num`*1.0`.

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

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 represents a subscript. The index notation X`[`I`]` is a special function that returns the Ith component of X, counting from the beginning. 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.

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 the functions `get(`L`,length)` and `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 attribute `length` is accessed.

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 following built-ins on lists are provided by the `basic` module:

- $Term_1$ `++` $Term_2$ `=` $List$: This function returns the concatenated list of $Term_1$ and $Term_2$. If an operand is a list, then the list is concatenated. If an operand is a primitive value, then the value is converted to a string, using `to_string`, before it is concatenated. If an operand is a structure, then it is converted to a string, using `to_list`, before it is concatenated. If an operand is a variable or an incomplete list, then an error is thrown.

- `append(`X`,` Y`,` Z`)` (nondet): When all three parameters are bound, this predicate determines whether Y can be appended to X in order to create Z. When Z is a variable, this predicate appends list Y to list X in order to create list Z. In all other cases, this predicate may backtrack, instantiating X and Y to lists, such that appending Y to X will create Z.

- `avg(`$List$`)` `=` Val: This function returns the average of all of the numbers in $List$.

- `delete(`$List$`,` X`)` `=` $ResList$: This function deletes the first occurrence of X from $List$, returning the result in $ResList$.

- `delete_all(`$List$`,` X`)` `=` $ResList$: This function deletes all occurrences of X from $List$, returning the result in $ResList$.

- `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 `sublist(`$List$`,` `1`, $Index$`-1)++[`Elm`|sublist(`$List$`,` $Index$`,` $List$`.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 `sublist(`$List$`,` `1`, $Index$`-1)++`$AList$`++sublist(`$List$`,` $Index$`,` $List$`.length)`.

- `length(`$Compound$`)` `=` Len: This function returns the number of elements that are contained in a compound term.

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

- `max(`$List$`)` `=` Val: This function returns the maximum value that is in $List$.

- `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 $List$.

- `new_list(`N`)` = $List$: This function creates a new list that has N free variable arguments.
- `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 $ResList$.
- `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.
- `sort(`$List$`)` = $SList$: This function sorts the elements of $List$ in ascending order, returning the result in $SList$.
- `sort_down(`$List$`)` = $SList$: This function sorts the elements of $List$ in descending order, returning the result in $SList$.
- `sublist(`$List$`,` $Start$`,` End`)` = $SubList$: Given $List$, and indices $Start$ and End, this function returns the sublist [$List_{Start}$, ..., $List_{End}$].
- `sum(`$List$`)` = Val: This function returns the sum of all of the values in $List$.
- `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 tuples. The jth tuple in the list takes the form `(`$E_{1j}, \ldots, E_{nj}$`)`, where E_{ij} is the jth element in $List_i$.

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 $UString$.

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 two dollar symbols are used to distinguish a structure from a function call. The *functor* of a structure comprises the name and the arity of the structure. A structure has two attributes: `name` and `arity`. The attribute `arity` is also named `length`.

The following types of structures can never denote functions, meaning that they do not need to be placed between two $ symbols.

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

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]`
If-then:	`if X>Y then Z=X else Z=Y end`
Loops:	`foreach (X in L) writeln(X) end`

The compiler will report a syntax error when it encounters any of these expressions within a pair of $ symbols.

The following built-ins are provided for structures:

- `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 $Struct$.

An *array* takes the form $\{t_1, \ldots, t_n\}$, which is a special structure with the name `'{}'` and arity n. 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.

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`. In addition to the built-ins for structures, the following built-ins are provided for maps:

- `new_map(`$PairsList$`)` = Map: This function creates a map. $PairsList$ is a list of pairs, where each pair has the form Key=Val.
- `map(`$Term$`)`: This predicate is true if $Term$ is a map.
- `get(`X`,` Key`)` = Val: This function returns Val of the key-value pair Key=Val that is attached to X. It throws an error if the variable has no attribute named Key.
- `put(`X`,` Key`,` Val`)`: This predicate attaches the key-value pair Key=Val to X, where Key is a non-variable term, and Val is any term.
- `keys(`X`)` = $List$: This function returns the list of names of the attributes of X.
- `values(`X`)` = $List$: This function returns the list of values of the attributes of X.
- `has_key(`X`,` Key`)`: This predicate is true if X has an attribute named Key.
- `map_to_list(`Map`)` = $PairsList$: This function returns a list of Key=Val pairs that constitute Map.

Most of the built-ins are overloaded for attributed variables.

3.5 Equality Testing and Unification

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, unlike in C-family languages, in which the equality test operator == only compares addresses.

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.

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> X = $f(X)$
X = f(f(......
```

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 entirely 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.

3.6 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]`.

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 Basic I/O

The `basic` module provides the following functions to perform input and output to the console. Users can use these functions without importing the io module.

- `read_int() =` Int: This is the same as `io.fread_int(stdin)`.
- `read_real() =` $Real$: This is the same as `io.fread_real(stdin)`.
- `read_char() =` Val: This is the same as `io.fread_char(stdin)`.
- `read_char(`N`) =` $String$: This is the same as `io.fread_char(stdin,` N`)`.
- `read_unicode_char() =` Val: This is the same as `io.fread_unicode_char(stdin)`.
- `read_unicode_char(`N`) =` $String$: This is the same as `io.fread_unicode_char(stdin,` N`)`.
- `read_token() =` $String$: This is the same as `io.fread_token(stdin)`.
- `read_term() =` $Term$: This is the same as `io.fread_term(stdin)`.
- `read_line() =` $String$: This is the same as `io.fread_line(stdin)`.
- `readln() =` $String$: This is the same as `io.freadln(stdin)`.
- `write(`$Term$`)`: This is the same as `io.fwrite(stdout,` $Term$`)`.
- `write_byte(`$Bytes$`)`: This is the same as `io.fwrite_byte(stdout,` $Bytes$`)`.
- `writeln(`$Term$`)`: This is the same as `io.fwriteln(stdout,` $Term$`)`.
- `writef(`$Format$`,` $Args\ldots$`)`: This is the same as `io.fwritef(stdout,` $Format$`,` $Args\ldots$`)`.
- `print(`$Term$`)`: This is the same as `io.fprint(stdout,` $Term$`)`.
- `printf(`$Format$`,` $Args\ldots$`)`: This is the same as `io.fprintf(stdout,` $Format$`,` $Args\ldots$`)`.
- `println(`$Term$`)`: This is the same as `io.fprintln(stdout,` $Term$`)`.
- `flush`: This is the same as `io.flush(stdout)`.

For predicates that read up to N values, the user can terminate input before entering N values by typing `ctrl-z` on Windows systems, and `ctrl-d` on Unix systems.

The `io` module provides additional built-ins for file I/O. See Chapter 9.

3.8 Other Built-ins on Terms

- `get_global_map()` = Map: This function returns the global map, which is shared by all threads.
- `get_heap_map()` = Map: This function returns the current thread's heap map. Each thread has its own heap map.
- `apply(`S`,` Arg_1`, ...,` Arg_n`)` = Val: This is a higher-order call. 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.
- `call(`S`,` Arg_1`, ...,` Arg_n`)`: This is a higher-order call. 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 $Arg_1, \ldots, Arg_n$.
- `findall(`$Template$`,` S`,` Arg_1`, ...,` Arg_n`)` = $List$: This is a higher-order call. It returns a list of all possible solutions of `call(`S`,` Arg_1`, ...,` Arg_n`)` in the form of $Template$.
- `acyclic_term(`$Term$`)`: This predicate is true if $Term$ is acyclic, meaning that $Term$ does not contain itself.
- `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.
- `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.
- `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.
- `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. Different variables receive different numberings, and the occurrences of the same variable all receive the same numbering.
- `unnumber_vars(`$Term_1$`)` = $Term_2$: $Term_2$ is a copy of $Term_1$, with all numbered variables being replaced by Picat variables. Different numbered variables are replaced by different Picat variables.
- `vars(`$Term$`)` = $Vars$: This function returns a list of variables that occur in $Term$.
- `variant(`$Term_1$`,` $Term_2$`)`: This predicate is true if $Term_2$ is a variant of $Term_1$.
- `subsumes(`$Term_1$`,` $Term_2$`)`: This predicate is true if $Term_1$ subsumes $Term_2$.
- `freeze(`X`,` $Goal$`)`: This predicate delays the evaluation of $Goal$ until X becomes a non-variable term.
- `ground(`$Term$`)`: This predicate is true if $Term$ is ground. A *ground* term does not contain any variables.

- `hash_code(`$Term$`) = `Int: This function returns the hash code for $Term$.
- `parse_term(`$String$`, `$Term$`, `$Vars$`, `$RString$`)`: This predicate uses the Picat parser to extract a term $Term$ from $String$. $Vars$ is a list of pairs, where each pair has the form $Name$=Var. $RString$ is the remaining string of unconsumed characters.
- `parse_term(`$String$`, `$Term$`, `$Vars$`)`: This is the same as `parse_term(`$String$`, `$Term$`, `$Vars$`, [])`.
- `parse_term(`$String$`) = `$Term$: This is the same as `parse_term(`$String$`, `$Term$`, _, [])`.

Chapter 4

Predicates and Functions

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable* rule

$Head, Cond$ => $Body$.

and the *backtrackable* rule

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

Each rule is terminated by a dot (`.`) followed by a white space.

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 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, *non-deterministic* 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], append(XsR,Ys,Zs).

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 answers 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 a 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 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 `=>` 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)).

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

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

The function `reverse(L)` returns the reversed list of `L`. Note that the function `reverse(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, 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 `reverse(L)`, where `L` is a variable, Picat will throw the following exception:

```
unresolved_function_call(reverse(L)).
```

A pattern can contain *as-patterns* in the form V`@`$Pattern$, where V is a new variable in the rule, and $Pattern$ is a non-variable term. The as-pattern V`@`$Pattern$ 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.
- $p(t_1, \ldots, t_n)$: 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 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).

- `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.

- `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-then`: An if-then statement takes the form

```
if Cond_1 then
   Goal_1
elseif Cond_2 then
   Goal_2
   ⋮
elseif Cond_n then
   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-then statement, Picat finds the first condition $Cond_i$ that is true. If such a condition is found, then the truth value of the if-then statement is the same as $Goal_i$. If none of the conditions is true, then the truth value of the if-then statement is the same as $Goal_{else}$. Note that no condition can succeed more than once.

- `try-catch`: A `try-catch` statement specifies a goal to try, the exceptions that need to be handled when they occur during the execution of the goal, and a clean-up goal that is executed no matter whether the goal succeeds, fails, or is terminated by an exception. The detailed syntax and semantics of the `try-catch` statement will be given in Chapter 6 on Exceptions.

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

For an extensional relation that contains a large number of tuples, it is tedious to define such a relation as a predicate with pattern-matching rules. It is worse if the relation has multiple keys. In order to facilitate the definition of extensional relations, Picat allows the inclusion of *predicate facts* in the form $p(t_1,\ldots,t_n)$ in predicate definitions. Facts are translated into pattern-matching rules before they are compiled. A predicate definition that consists of facts can be preceded by an *index declaration* in the form

`index` $(M_{11},M_{12},\ldots,M_{1n})$ `...` $(M_{m1},M_{m2},\ldots,M_{mn})$

where each M_{ij} is either $+$ (meaning indexed) or $-$ (meaning not indexed). For each index pattern $(M_{i1},M_{i2},\ldots,M_{in})$, the compiler generates a version of the predicate that indexes all of the $+$ arguments.

Example

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

The predicate `edge` is translated into the following rules:

```
edge(X,Y),nonvar(Y) =>
    'edge_-+'(X,Y).
edge(X,Y),var(X) =>
    throw $index_violation(edge(X,Y))$.
edge(a,Y) ?=> Y=b.
edge(a,Y) =>  Y=c.
edge(b,Y) =>  Y=c.
edge(c,Y) =>  Y=b.

'edge_-+'(X,b) ?=> X=a.
'edge_-+'(X,c) ?=> X=a.
'edge_-+'(X,c) =>  X=b.
'edge_-+'(X,b) =>  X=c.
```

Two predicates are generated. The predicate `'edge_-+'` is for the second index pattern `(-,+)`, and the predicate `edge` consists of a dispatching rule (the first rule), an index-checking rule (the second rule), and rules for the encoded facts for the index pattern `(+,-)`. Note that this translation favors the first index pattern, since a call never needs to be dispatched if it matches the index pattern.

4.6 Tail Recursion

A rule is said to be *tail-recursive* if the last call of the body is the same predicate as the head. 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).

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).

reverse([]) = [].
reverse([X|Xs]) = reverse_helper(Xs,[X]).

reverse_helper([],R) = R.
reverse_helper([X|Xs],R) = reverse_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 `reverse_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 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 refers to the parts of a program where a variable occurs.

Consider the `test` example. This example binds `X` to 0. Then, the example tries to bind `X` to `X + 1`. However, `X` is still in scope, meaning that `X` is already bound to 0. 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 then
       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 = X * Z,
   Yout = Yin.
p(Xin, Xout, Yin, Yout) =>
   Xout = Xin,
   Yout = Y - 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 (I in L)
        println(I)
    end.

loop_ex2 =>
    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(I, L),
         println(I),
         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(I in 1 .. 2 .. 9)
        println(I)
    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 then
               printf("%d is not a multiple of %d%n", J, I)
            end
        end
    end.
```

5.2.3 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_ex6 =>
    I = 1,
    while (I <= 9)
        println(I),
        I := I + 2
    end.

loop_ex7 =>
    J = 6,
    while (J <= 5)
        println(J),
        J := J + 1
    end.

loop_ex8 =>
    E = read_int(),
    while (E mod 2 == 0; E mod 5 == 0)
```

```
        println(E),
        E := read_int()
    end.

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

The while loop in `loop_ex6` 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_ex7` 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_ex8` 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_ex9` also demonstrates a compound condition. Unlike in `loop_ex8`, in which either condition must be true, in `loop_ex9`, 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.

5.2.4 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_ex10 =>
    J = 6,
    do
        println(J),
        J := J + 1
    while (J <= 5).
```

Unlike `loop_ex7`, `loop_ex10` 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 Comprehensions

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

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

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.

Example

```
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. Instead, a predicate fact must be used, as in the next example.

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 also visible in the inner loop. However, variables that are local to the inner loop do not occur earlier, in 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,
        println()
    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 =>
    List = L,
    foreach (A in [a, b], X in 1 .. 2)
        L = [(A, X) | T],
        L := T
    end,
    L = [].
```

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 a special treatment. In Picat, an exception is just a term. A built-in exception is a structure, where 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

A built-in exception is one of the following:

- `divide_by_zero(`$Source$`)`: $Source$ divides a number by zero.
- `file_not_found(`$EArg$`,`$Source$`)`: $Source$ tries to open a file named $EArg$ that does not exist.
- `function_not_found(`$FName$`,`$Source$`)`: $Source$ tries to call a function that is not defined in the imported modules, where $Source$ is a higher-order call in which names cannot be completely bound to definitions at compile time.
- `interrupt(`$Source$`)`: The execution is interrupted by a signal. For an interrupt caused by `ctrl-c`, $Source$ is `keyboard`.
- `io_error(`ENo`,`$EMsg$`,`$Source$`)`: An I/O error with the number ENo and message $EMsg$ occurs in $Source$.
- `key_not_found(`Key`,`$Source$`)`: $Source$ tries to access a map or an attributed variable with a Key that does not exist.
- `load_error(`$FName$`,`$Source$`)`: An error occurs while loading the byte-code file named $FName$. This error is caused by the malformatted byte-code file.
- `out_of_memory(`$Area$`)`: The system runs out of memory while expanding $Area$, which can be: `stack_heap`, `trail`, `program`, `table`, or `findall`.
- `out_of_range(`$EIndex$`,`$Source$`)`: $Source$ tries to access an element of a compound value using the index $EIndex$, which is out of range. An index is out of range if it is less than or equal to zero, or if it is greater than the length of the compound value.
- `predicate_not_found(`$PredName$`,`$Source$`)`: $Source$ tries to call a predicate that is not defined in the imported modules, where $Source$ is a higher-order call in which names cannot be completely bound to definitions at compile time.

- `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, and `parse_term("a()")` also throws this exception, because the string `"a()"` is not a valid term.
- `unresolved_function_call(`$FCall$`)`: No rule is applicable to the function call $FCall$.
- $Type$`_expected(`$EArg$`,`$Source$`)`: The argument $EArg$ in $Source$ is not an 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 file_not_found("abc.txt",open("abc.txt"))
```

6.3 Defining Exception Handlers

The `try` statement, which takes the following form, is provided for defining exception handlers:

```
try
   Goal
catch (Pattern_1)
   Handler_1
⋮
catch (Pattern_n)
   Handler_n
finally
   Handler_fin
end
```

The `catch` clauses and the `finally` clause are optional. However, the `finally` clause is mandated if there is no `catch` clause, and there must be at least one `catch` clause if there is no `finally` clause. Each $Pattern_i$ is a term pattern, like the head of a rule, and each handler is a goal. For an exception, a `catch` clause is said to be *applicable* if the exception matches the pattern of the clause. When an exception is thrown during the execution of $Goal$, Picat walks along the chain of ancestor calls of the thrower of the exception until it finds an ancestor that is wrapped inside a `try` statement. For the `try` statement, Picat searches for the first applicable `catch` clause and executes the handler. If no `catch` clause exists, or no `catch` clause is applicable to the exception, then $Handler_{fin}$ is executed before the exception is re-thrown. The `finally` clause is always executed, if it exists. It is executed when $Goal$ terminates with an exception, and it is executed when $Goal$ finishes its normal execution. This means that the `finally` clause is executed when $Goal$ succeeds deterministically with no choice points left behind, and when $Goal$ fails. For example,

```
try
    S = open(File),
    process(S)
catch (E)
    writeln(E)
finally
    close(S).
```

The `finally` clause specifies a clean-up action for the `try` goal.

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 a $N \times N$ 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 memorize 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: a plus-sign (+) indicates input, a minus-sign (-) indicates output, `max` indicates that the corresponding variable should be maximized, and `min` indicates that the corresponding variable should be minimized. Input arguments are assumed to be ground. Output arguments, including `min` and `max` arguments, are assumed to be variables. An argument with the mode `min` or `max` 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 a table mode declaration is provided, Picat tables only one optimal answer for the same input arguments.

Example

```
table(+,+,-,min)
shortest_path(X,Y,Path,W) ?=>
    Path = [(X,Y)],
    edge(X,Y,W).
shortest_path(X,Y,Path,W) =>
    Path = [(X,Z)|Path1],
    edge(X,Z,W),
    shortest_path(Z,Y,Path1,W1),
    W = W+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 `shortest_path(X,Y,Path,W)` states that `Path` is a path from `X` to `Y` with the minimum weight `W`. Note that whenever the predicate `shortest_path/4` is called, the first two arguments must always be instantiated. For each pair, the system stores only one path 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,[(X,Y)],Path,O) ?=>
    Path = [(X,Y)],
    O = (Wxy,1),
    edge(X,Y,Wxy).
sp(X,Y,Path,O) =>
    Path = [(X,Z)|Path1],
```

```
    edge(X,Z,Wxz),
    shortest_path(Z,Y,Path1,O1),
    O1 = (Wzy,Len1),
    O = (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 would be `(Len,W)`, then the program would find a shortest path, breaking a tie by selecting one with the minimum weight.

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.*

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. It 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 fixed point 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.

7.3 Primitives on Tables

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

```
go =>
    state(s,s,s,s,Path,_),
    writeln([(s,s,s,s)|Path]).

table (+,+,+,+,-,min)
state(n,n,n,n,Path,Len) => Path=[], Len=0.
state(F,W,G,C,_Path,_Len), not safe(F,W,G,C)  => fail.
state(F,F,G,C,Path,Len) ?=>
    Path=[(F1,F1,G,C)|Path1],
    opposite(F,F1),
    state(F1,F1,G,C,Path1,Len1),
    Len = Len1+1.
state(F,W,F,C,Path,Len) ?=>
    Path=[(F1,W,F1,C)|Path1],
    opposite(F,F1),
    state(F1,W,F1,C,Path1,Len1),
    Len = Len1+1.
state(F,W,G,F,Path,Len) ?=>
    Path=[(F1,W,G,F1)|Path1],
    opposite(F,F1),
    state(F1,W,G,F1,Path1,Len1),
    Len = Len1+1.
state(F,W,G,C,Path,Len) =>
    Path=[(F1,W,G,C)|Path1],
    opposite(F,F1),
    state(F1,W,G,C,Path1,Len1),
    Len = Len1+1.

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

safe(F,W,F,C) => true.
safe(F,F,G,F) =>
    opposite(F,G).
```

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

- `table_get_all(`$Goal$`) = `$List$: This function returns a list of answers of the subgoals that are subsumed by $Goal$. For example, the `table_get_all(_)` fetches all of the answers in the table, since any subgoal is subsumed by the anonymous variable.
- `table_get_one(`$Goal$`)` : If there is a subgoal in the subgoal table that is a variant of $Goal$, and that has answers, then $Goal$ is unified with the first answer. This predicate fails if there is no variant subgoal in the table, or if there is no answer available.

Chapter 8

Modules

A module is a bundle of predicate and function definitions that are stored in one file. A module forms a name space. 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.

8.1 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 module can be spread across multiple files. In that case, only the file $Name$`.pi` will have an the module declaration `module` $Name$. Furthermore, $Name$`.pi` will have an `include` statement, listing the other files that contain parts of the module. For example, the file `basic.pi` has the following content:

```
module basic.
include "basic_list.pi", "basic_array.pi",
        "basic_map.pi", "basic_term.pi", "basic_io.pi".
```

The file `basic.pi` does not define any predicates or functions, but it includes a bunch of other source files. None of the included files has a `module` declaration. Note that the file names in the `include` statement are strings, and are not atoms. Also note that the paths of the included files must be relative to the file that contains the `include` statement.

A file that does not begin with a module declaration is assumed to belong to the default *global* module.

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 one of the following items:

- A module name M. All of the public predicate and function symbols of module M are visible to the importing module.
- $M.S/N$ where M is a module name, S is a symbol, and N is an arity. The symbol S/N of module M is visible.

- $M.S$ where M is a module name, and S is a symbol. If M contains several versions of S with different arities, then all of the versions are visible.

For each imported item, the compiler first searches for a module for the item in the search path that is specified by the environment variable PICATPATH. If the imported item is in the form of $M.S$ or $M.S/N$, the compiler also searches for a definition of the symbol S in the module. If no module or definition is found, the compiler gives an error message. The global module is imported by default. Symbols that are defined in the global module cannot be redefined.

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 `load("xxx")` compiles the file `xxx.pi` and loads the generated code into the interpreter. The generated bytecode is stored in a file named `xxx.qi` in the same path as the source file. 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.

8.2 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 none-higher-order) calls to their definitions takes place at compile time. For each call, the compiler first searches the enclosing module and the global module 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 list of imported items. The compiler searches the items in the order that they were imported." If no definition is found in any of these modules, then the compiler will issue an warning.[1]

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)).
```

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

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

The module `qsort.pi` 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.

```
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 `qsort` is loaded before `isort`, the `sort` function defined in `qsort` 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, or if M is an atom that is not a module name.

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).
```

8.3 Accessing Attributes of Modules

A function with no argument that is defined with one function fact in a module is called an *attribute* of the module. For example, the `math` module contains a function named `pi`, where

```
pi=3.14159.
```

The advantage of treating no-arg functions as attribute-value pairs is that users can use the dot notation to access values. For example, users can use `math.pi` to retrieve the value that is associated with `pi` in the module `math`.

Users can use no-argument functions in order to simulate C-style enum types.

Example

```
module color.

black = 0.
```

```
red = 1.
blue = 2.
green = 3.
white = 4.
cyan = 5.
yellow = 6.
magenta = 7.
```

For a color, say `red`, users can retrieve the integer that is associated with it by using `color.red`. Note that a dot notation is an expression, but is not a valid term. Therefore, a dot notation cannot occur in a head pattern.

8.4 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 `apply(F,X,Y)`, then the compiler generates code to search the enclosing module and the imported modules for a definition at runtime. 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.

8.5 Library Modules

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

Chapter 9

I/O

Picat has an `io` module for reading input from files and writing output to files.

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

```
import io.

rw =>
     Reader = open("input_file.txt"),
     Writer = open("output_file.txt", write),
     L = fread_line(Reader),
     while (L != eof)
            fprintln(Writer, L),
            flush(Writer),
            L := fread_line(Reader)
     end,
     close(Reader),
     close(Writer).
```

9.1 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 filename 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 four atoms: `read`, `write`, `append`, or `create`. 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. The `create` atom is also used to create a new file for both reading and writing; if the file already exists, then the program will throw an error.

9.2 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 unicode characters, tokens, strings, and bytes. Recall that strings are stored as lists of single-character atoms.

The `fread` 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.

- `fread_int(`FD`) =` Int: This function reads a single integer from the file that is represented by FD.
- `fread_real(`FD`) =` $Real$: This function reads a single real number from the file that is represented by FD.
- `fread_char(`FD`) =` Val: This function reads a single character, whose byte size depends on the current system, from the file that is represented by FD.
- `fread_char(`FD`, `N`) =` $String$: This function reads up to N characters, whose byte size depends on the current system, from the file that is represented by FD. It returns a string that contains the characters that were read.
- `fread_unicode_char(`FD`) =` Val: This function reads a single Unicode character from the file that is represented by FD.
- `fread_unicode_char(`FD`, `N`) =` $String$: This function reads up to N Unicode characters from the file that is represented by FD. It returns a string that contains Unicode characters that were read.
- `fread_token(`FD`) =` $String$: This function reads a single Picat token from the file that is represented by FD.
- `fread_term(`FD`) =` $Term$: This function reads a single Picat term from the file that is represented by 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 string.
- `fread_line(`FD`) =` $String$: This function reads a string from the file that is represented by FD, stopping when either a newline ('`\r\n`' on Windows, and '`\n`' on Unix) is read, or the `eof` atom is returned. The newline is not stored in the returned string.
- `freadln(`FD`) =` $String$: This function does the same thing as `fread_line`.
- `fread_byte(`FD`) =` Val: This function reads a single byte from the file that is represented by FD.
- `fread_byte(`FD`, `N`) =` $List$: This function reads up to N bytes from the file that is represented by FD. It returns the list of bytes that were read.
- `fread_file_chars(`FD`) =` $String$: This function reads an entire character file into a string. The next function that tries to read from the same file will return `eof`, unless the read/write pointer is repositioned or the file is modified.

- `fread_file_bytes(`FD`) = `$List$: This function reads an entire byte file into a list. The next function that tries to read from the same file will return `eof`, unless the read/write pointer is repositioned or the file is modified.

There are cases when the `fread_char(`FD`, `N`)`, `fread_unicode_char(`FD`, `N`)`, and `fread_byte(`FD`, `N`)` functions will 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 `fread` functions wait until data is written to the pipe. As soon as the pipe has data, the `fread` functions read the data. If a pipe has fewer than N values when a read occurs, then these three functions will return a string 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 `fread` or `peek` function will return the same value, unless the read/write pointer is repositioned or the file is modified.

- `peek_int(`FD`) = `Int
- `peek_real(`FD`) = `$Real$
- `peek_char(`FD`) = `Val
- `peek_unicode_char(`FD`) = `Val
- `peek_byte(`FD`) = `Val

9.2.1 End of File

The end of a file is detected through the `eof` atom. If the input function returns a single value, and the read/write pointer is at the end of the file, then the `eof` atom is returned. If the input function returns a list, then the end-of-file behavior is more complex. If no other values have been read into the list, then the `eof` atom is returned. However, if other values have already been read into the list, then reaching the end of the file causes the function to return the list, and the `eof` atom will not be returned until the next input function is called.

Instead of checking for `eof`, 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

```
import io.

rw =>
     Reader = open("file1.txt"),
     Writer = open("file2.txt", write),
     while (not at_end_of_stream(Reader))
            L := fread_line(Reader),
            fprintln(Writer, L),
            flush(Writer)
```

```
    end,
    close(Reader),
    close(Writer).
```

The advantage of using the `at_end_of_stream` predicate instead of using the `eof` 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 `eof` atom, an extra `fread_line` function was needed before the end of the file was detected. In the above example, which used `at_end_of_stream`, `fread_line` was only called if there was data remaining to be read.

9.3 Writing to a File

The `fwrite` and `fprint` predicates take a file descriptor as the first parameter. The file descriptor is the same descriptor that the `open` function returns.

- `fwrite(`FD`,` $Term$`)`: This predicate writes $Term$ to a file. Single-character lists are treated as strings. Strings are double-quoted, and atoms are single-quoted when necessary. This predicate does not print a newline, meaning that the next write will begin on the same line.
- `fwrite_byte(`FD`,` $Bytes$`)`: This predicate writes a single byte or a list of bytes to a file.
- `fwriteln(`FD`,` $Term$`)`: This predicate prints a newline, meaning that the next write will begin on the next line.
- `fwritef(`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.

Note that these predicates write both primitive values and compound values.

The `fwritef` 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`, `%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 E.

Example

```
import io.

formatted_print =>
    FD = open("birthday.txt"),
    Format1 = "Hello, %s.  Happy birthday!  ",
    Format2 =  "You are %d years old today.  ",
    Format3 =  "That is %.2f%% older than you were last year",
```

```
    fwritef(FD, Format1, "Bob"),
    fwritef(FD, Format2, 7),
    fwritef(FD, Format3, 7.0 / 6.0),
    close(FD).
```

This writes "`Hello, Bob. Happy birthday! You are 7 years old today. That is 1.17% older than you were last year`".

The `io` module also has the three `fprint` predicates.

- `fprint(`FD`,` $Term$`)`: This predicate writes $Term$ to a file. Unlike the `fwrite` predicates, the `fprint` predicates do not place quotes around strings and atoms.
- `fprintln(`FD`,` $Term$`)`
- `fprintf(`FD`,` $Format$`,` $Args\ldots$`)`: This predicate is the same as `fwritef`, except that `fprintf` uses `fprint` to display the arguments in the $Args$ parameter, while `fwritef` uses `fwrite` to display the arguments in the $Args$ parameter.

The following example demonstrates the differences between the `write` and `print` predicates. It uses the `write` and `print` predicates from the `basic` module.

Example

```
picat> write("abc")
"abc"
picat> write([a,b,c])
"abc"
picat> write('a@b')
'a@b'
picat> writef("%w %s%n",[a,b,c],"abc")
"abc" "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
```

9.4 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.

9.5 Repositioning I/O Pointers Within Files

Sometimes, sequential file access is not enough. Picat provides functions and predicates that allow a program to access and to modify its current location in a file. These built-ins are demonstrated in the following example.

Example

```
import io.

rw =>
     Writer = open("bond.txt", write),
     fprintln(Writer, " Bond, James The name is."),
     flush(Writer),
     close(Writer),
     Reader = open("bond.txt", read),
     CS = sizeof_char(),
     setpos(Reader, 13 * CS)
     L = fread_char(Reader, 11),
     print(L), % "The name is"
     rewind(Reader),
     L := fread_char(Reader, 13),
     print(L), % " Bond, James "
     seek(Reader, -12 * CS, current),
     L := fread_char(Reader, 4),
     print(L), % "Bond"
     seek(Reader, 1 * CS, end)
     C = fread_char(Reader),
     print(C), % "."
     close(Reader).
```

The above example prints "`The name is Bond, James Bond.`" if there are no errors during I/O. The following five functions and predicates are used for repositioning file locations.

- `getpos(`FD`) = `Pos: This function returns the current file position, indicating the offset in the number of bytes from the beginning of the file.
- `setpos(`FD`, `Pos`)`: This predicate changes the current file position of the read/write pointer to Pos, which is the number of bytes from the beginning of the file.
- `rewind(`FD`)`: This predicate repositions the read/write pointer at the beginning of the file.
- `seek(`FD`, `$Offset$`, `$From$`)`: This predicate is similar to `setpos`, because both predicates change the position of the read/write pointer. However, the `seek` predicate uses two arguments to indicate the file location. The $Offset$ argument indicates the number of bytes to move. The $From$ argument is an atom. If $From$ is `beginning`, then `seek` will move the read/write pointer to $Offset$ bytes from the beginning of the file. If $From$ is `current`, then `seek` will move the read/write pointer to $Offset$ bytes from the current position. This is the only case where $Offset$ can be negative, because a negative offset indicates that the read/write pointer should be moved backwards. If $From$ is `end`, then `seek` will move the

read/write pointer to $Offset$ bytes before the end of the file; note that, although the pointer is moved backwards from the end of the file, $Offset$ must be a positive number.

- `sizeof_char() = Size`: The bytesize of a character can differ on different systems. Therefore, the `sizeof_char` function indicates the size of a character, in bytes, on the current system. The `getpos`, `setpos`, and `seek` functions all indicate offsets in bytes. If you want to show how many *characters* to move the read/write pointer, use the `sizeof_char` function, and multiply the $Offset$ or Pos parameter by the result, as shown in the previous example. Note that if you seek to the middle of a character while performing character I/O, the program will have unpredictable behavior.

The repositioning functions have no effect if they are passed the file descriptor of standard input, standard output, or standard error. The `getpos` function will return 0, while the `setpos`, `seek`, and `rewind` functions will not change the file pointer location.

9.6 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. An advantage of using these atoms is that they can be used to allow the user to redirect standard input, standard output, and standard error to files, as shown in the following section.

9.7 Redirection

The following example shows how to redirect standard input and standard output to files. This allows the program to read from and to write to files by using the functions and predicates that are defined in the `basic` module.

Example

```
import io.

rw1 =>
     Reader = open("input_file.txt"),
     Writer = open("output_file.txt", write),
     close(stdin),
     InFD = dup(Reader),  % Redirects standard input
     close(stdout),
     OutFD = dup(Writer), % Redirects standard output
     close(Reader),
     close(Writer),
     L = read_line(),
     while (not at_end_of_stream(InFD))
            L := read_line(), % Reads from input_file.txt
            writeln(L)        % Writes to output_file.txt
     end.

rw2 =>
     Reader = open("input_file.txt"),
```

```
Writer = open("output_file.txt", write),
dup2(Reader, stdin),  % Redirects standard input
dup2(Writer, stdout), % Redirects standard output
close(Reader),
close(Writer),
L = read_line(),
while (not at_end_of_stream(stdin))
       L := read_line(), % Reads from input_file.txt
       writeln(L)        % Writes to output_file.txt
end.
```

The above example uses the `dup` and `dup2` built-ins.

- `dup(`FD`) = `$NewFD$: This function modifies the program's file descriptor table. It takes the lowest available file descriptor, and adds it to the program's file descriptor table. Then, `dup` copies the name of FD's file to the new file descriptor. The `dup` function returns the new file descriptor.

- `dup2(`$FromFD$`, `$ToFD$`)`: This predicate modifies the program's file descriptor table, performing two operations. First, `dup2` closes the entry to which $ToFD$ points in the file descriptor table. Then, `dup2` copies the name of the file of $FromFD$ to $ToFD$ in the file descriptor table.

If the above examples do not throw any errors, then they close standard input, copying `input_file.txt` to the file descriptor that is usually reserved for standard input. Then, they close standard output, copying `output_file.txt` to the file descriptor number that is usually reserved for standard output. At this point, `input_file.txt` and `output_file.txt` each have two entries in the file descriptor table. Then, since `Reader` and `Writer` will not be used, their file descriptors are closed. At this point, `input_file.txt` and `output_file.txt` each have one entry in the file descriptor table. Finally, the program uses `read_line` and `writeln`, both of which are defined in the `basic` Picat module, in order to read from and to write to the files.

Notice the differences between `rw1` and `rw2`. The `rw1` example explicitly closes `stdin`. Then, the lowest available file descriptor is the file descriptor that was used for `stdin`, so `dup` copies `input_file.txt` to the file descriptor that is usually reserved for standard input. Then, `rw1` explicitly closes `stdout`. The lowest available file descriptor is the file descriptor that was used for `stdout`, so `dup` copies `output_file.txt` to the file descriptor that is usually reserved for standard output. Unlike `rw1`, the `rw2` example explictly indicates standard input and standard output as the destination file descriptors for `dup2`. The `dup2` predicate closes `stdin` and `stdout` automatically.

Note that a program's file descriptor table is initialized as soon as it begins execution. Each process has its own file descriptor table. The `dup` and `dup2` built-ins only modify the current program's file descriptor table, which is destroyed when the program finishes execution. Therefore, if a program is executed multiple times, then `stdin`, `stdout`, and `stderr` are initialized to their default values before execution.

Note that standard input, standard output, and standard error do not need to be explicitly closed, even if they are redirected. This is why the program does not call `close` after the while loop terminates.

The following example shows how to use `dup2` together with the `process` module in order to imitate the command "`ls -l > dir.txt`".

Example

```
import io, process.

ls =>
    Writer = open("dir.txt", write),
    dup2(Writer, stdout),
    close(Writer),
    process.exec("ls", "-l").
```

The `dup2` predicate redirects standard output to "`dir.txt`". Therefore, when `exec` is called, the output of "`ls`", will also redirect to "`dir.txt`". For more information about the `exec` predicate, see Chapter 13.

9.8 Temporary Files and Pipes

The `io` module has four functions whose purpose is to create new files and file descriptors for the purposes of communication.

9.8.1 Temporary Files

The `mktmp` function is used to create temporary files.

- `mktmp() =` FD: This function creates a temporary file in the file system, and returns a file descriptor for the temporary file. This file descriptor can be used to read from and to write to the file. The temporary file exists until the file descriptor is closed, or until termination of the program that called the `mktmp` function.

9.8.2 Pipes

Pipes are special files that are used for interprocess communication. Picat provides three functions and predicates for the creation of pipes.

- `mkpipe() =` FD_Map: This function creates an unnamed pipe, which is stored in in kernel memory. This function returns a map with two keys. The `readFD` key represents the file descriptor that is used to read from the pipe. The `writeFD` key represents the file descriptor used to write to the pipe. The unnamed pipe can only be used by related processes. It is removed from kernel memory when all processes that use the pipe have terminated. Note that if the pipe is empty, and no process has the pipe open for writing, then the `fread` functions will return `eof`.

- `mkfifo(`$Path$`)`: This predicate creates a named pipe, or a *FIFO*, which is stored as a file in the location specified by $Path$. The FIFO can be used by unrelated processes. Like a regular file, the FIFO remains in memory even if it is not currently being used by any processes. In order to remove the FIFO from memory, use the `rm` predicate that is defined in the `os` module.

- `mkfifo(`$Path$`,` $Mode$`)`: This version of `mkfifo` has a $Mode$ parameter. This parameter is either a single atom or a list of two or three atoms. The format of the atoms is specified by the regular expression `r?w?(u|g|o)`. If the atom contains `r`, then it provides permission to read from the FIFO. If the atom contains `w`, then it provides permission to write to the FIFO. The second part of the atom indicates the receiver(s) of the permission, where `u`

specifies the user, `g` specifies anybody who is in the user's group, and `o` specifies anybody who is not in the user's group. At most one atom can exist for each of `u`, `g`, and `o`. If a permission is not explicitly specified in the $Mode$ parameter, then the permission is not provided. However, the `mkfifo(`$Path$`)` predicate provides a default permission list of `[rwu, rwg]`, which allows the user and the user's group to read from and to write to the FIFO.

The following example demonstrates the difference between unnamed pipes and named pipes.

Example:

```
% File 1
import io, process.

unnamed =>
    FDs = mkpipe(), % create the pipe
    ID = process.fork(),
    if ID == 0 then % child process
       close(FDs.writeFD),
       reader(FDs.readFD)
    else            % parent process
       close(FDs.readFD),
       writer(FDs.writeFD)
    end.

reader(FD) =>
    Str = fread_line(FD),
    close(FD).

writer(FD) =>
    fwriteln(FD, "Communicating"),
    close(FD).

% File 2
import io.

fifo =>
   mkfifo("fifo.txt", [rwu, rwg, rwo]),
   Writer = open("fifo.txt", append),
   foreach (I in 1..100)
            fwriteln(Writer, I)
   end,
   close(Writer).

% File 3 -- executed by any user some time after File 2 was executed
import io, os.

nofifo =>
   Reader = open("fifo.txt", read),
   while (not at_end_of_stream(Reader))
```

```
        I := fread_int(Reader),
        write_int(I), % prints I to standard output
        writeln()
end,
close(Reader),
os.rm("fifo.txt"). % deletes the fifo
```

Notice the differences between the unnamed pipe and the named pipe. The unnamed pipe is used by two related processes. Each process closes one of the pipe's file descriptors immediately, and closes the other file descriptor as soon as the process finishes communicating. As soon as the `unnamed` predicate finishes executing, the unnamed pipe is removed from memory. The named pipe can be used by two unrelated processes. After the `fifo` predicate finishes executing, the named pipe remains in memory until it is removed in the `nofifo` predicate.

For more information on processes, see Chapter 13.

9.8.3 A Note on Errors

The functions and predicates in the `io` module can throw a large number of errors. For example, attempts to read from and to write to a non-existent file descriptor will generate an error. Another error occurs when trying to create a file that already exists. Errors can also be generated when the user tries to perform an operation on a file, such as writing to the file, when the user does not have permission to perform the operation.

In most cases, if an I/O error occurs, then Picat will throw `io_error(`ENo`,` $EMsg$`,` $Source$`)`, where ENo is the error number, $EMsg$ is a string that indicates the error that occurred, and $Source$ is the goal that caused the error to occur.

If the `open` function is unable to find the file that is passed to it, then the `open` function will throw `file_not_found(`$Earg$`,` $Source$`)`, where $Earg$ is the name of the file, and $Source$ is the function or predicate that called `open`.

Chapter 10

The File System

Picat has an `os` module for manipulating files and directories.

10.1 The $Path$ Parameter

Many of the functions and predicates in this module have a $Path$ parameter. This parameter is a string, representing the path of a file or directory. This path can be an absolute path, from the system's root directory, or a relative path, from the current file location. Different systems use different separator characters to separate directories in 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() =` Val

10.2 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.

directory_traversal =>
   Root = root(), % get the root directory
   traverse(Root).

traverse(Dir) =>
   DirsList = listdir(Dir),
   if (DirsList != [])
       print("Inside "),
       println(Dir),
       println(DirsList)
   end,
   foreach (File in DirsList)
       if (directory(File)) % if File is a directory, traverse it
           FullPath = Dir ++ separator() ++ File,
           traverse(FullPath) % recursive traversal
```

```
    end
end.
```

The following functions 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.
- `listdir(`$Path$`, `$Pattern$`) = `$List$: The $Pattern$ parameter is a string that represents a regular expression. The list that is returned will only contain files and directories whose names match the regular expression.
- `root() = `$Path$: This function returns a string representing the path of the root of the file system tree (such as "C:\", or "/").

The above example also uses the `directory` predicate, which will be discussed in Section 10.4.

10.2.1 The Current Working Directory

The `os` module includes two functions that obtain the program's current working directory:

- `cwd() = `$Path$
- `pwd() = `$Path$

The `os` module also includes two predicates to change the program's current working directory:

- `cd(`$Path$`)`
- `chdir(`$Path$`)`

If the `cd` and `chdir` predicates cannot move to the directory specified by $Path$, the functions throw an error. This can occur if $Path$ does not exist, if $Path$ is not a directory, or if the program does not have permission to access $Path$.

10.3 Modifying Files and Directories

In a file system, each directory entry refers to a structure called an *inode*, which contains file information. The `ino` function returns the number of the inode to which $Path$ refers.

10.3.1 Creation

The `os` module contains a number of predicates for creating new files and directories:

- `create(`$Path$`)`: This predicate creates a new file at location $Path$. The file 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.
- `create(`$Path$`, `$Mode$`)`: The $Mode$ parameter indicates access permissions. For details, see the `chmod` function, in Section 10.3.2. If a permission is not explicitly specified in the $Mode$ parameter, then the permission is not provided.

- `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.

- `mkdir(`$Path$`, `$Mode$`)`: The $Mode$ parameter indicates access permissions. For details, see the `chmod` function, in Section 10.3.2. If a permission is not explicitly specified in the $Mode$ parameter, then the permission is not provided. Note that permissions that include `x` are not allowed for directories.

- `mkdirs(`$Path$`)`: This predicate performs recursive directory creation. It creates a new directory at location $Path$. If the parent directories of $Path$ do not exist, then this predicate also creates the parent directories. This predicate will throw an error if it does not have permission to create the highest directory that it must create.

- `mkdirs(`$Path$`, `$Mode$`)`: All of the directories that `mkdirs` creates will have the same permissions.

- `mv(`$Path1$`, `$Path2$`)`: This moves a file or a directory from $Path1$ to $Path2$. This predicate will throw an error if $Path1$ does not exist. An error will also occur if the program does not have permission to write to $Path1$ or $Path2$.

- `cp(`$Path1$`, `$Path$`)`: This copies a file or directory from $Path1$ to $Path2$. This predicate will throw an error if $Path1$ does not exist. An error will also occur if the program does not have permission to read from $Path1$, or if it does not have permission to write to $Path2$.

- `link(`$Path1$`, `$Path2$`)`: This creates a hard link to $Path1$ in location $Path2$. A *hard link* is a directory entry. There can be multiple hard links to the same inode. This predicate will throw an error if it does not have permission to write to $Path2$.

- `shortcut(`$Path1$`, `$Path2$`)`: This creates a symbolic link to $Path1$ in location $Path2$. A *symbolic link* is a file that contains the name of another file or directory. This predicate might not be supported by Windows. This predicate will throw an error if it does not have permission to write to $Path2$.

10.3.2 Modification

The `os` module has one predicate for modifying access permissions.

- `chmod(`$Path$`, `$Mode$`)`: An error will be thrown if the program cannot modify the permissions.

The $Mode$ parameter is either a single atom or a list of two or three atoms. This parameter specifies the access permissions. The format of the atoms is specified by the regular expression `r?w?x?(u|g|o)`. If the atom contains `r`, then it provides permission to read from the file. If the atom contains `w`, then it provides permission to write to the file. If the atom contains `x`, then it provides permission to execute the file. The second part of the atom indicates the receiver(s) of the permission, where `u` specifies the user, `g` specifies anybody who is in the user's group, and `o` specifies anybody who is not in the user's group. At most one atom can exist for each of `u`, `g`, and `o`.

Note that the `chmod` predicate will only modify the permissions for the specified receivers. If a receiver is not specified in the $Mode$ parameter, then the receiver will have the same permissions as the receiver had before `chmod` was called.

10.3.3 Deletion

The `os` module contains a number of predicates for deleting files and directories.

- `rm(`$Path$`)`: This deletes a file. Errors 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. Errors will be thrown if the directory does not exist, the program does not have permission to delete the directory, the directory is not empty, or if $Path$ does not refer to a directory.
- `unlink(`$Path$`)`: This removes a hard link or a symbolic link.

10.4 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. These predicates will all throw an error if the program does not have permission to read from $Path$.

- `dev_id(`$Path$`) =` Int: This function returns the device ID of the device that contains $Path$.
- `ino(`$Path$`) =` Int: This function returns the inode number of $Path$.
- `mode(`$Path$`) =` $String$: This function returns the file permissions for $Path$ in a string. The string will contain atoms, in the same format as the atoms that are passed to the `chmod` predicate. This function will throw an error if $Path$ does not exist.
- `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?
- `mode(`$Path$`, `$Value$`)`: Does the program's current permission for the file include $Value$? The $Value$ parameter is one of the atoms: `read`, `write`, or `execute`.
- `nlink(`$Path$`) =` Int: This function returns the number of hard links to the inode to which $Path$ refers.
- `uid(`$Path$`) =` Int: This function returns the user ID of the user who created $Path$. This function will throw an error if $Path$ does not exist.
- `gid(`$Path$`) =` Int: This function returns the group ID of the user who created $Path$. This function will throw an error if $Path$ does not exist.
- `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.
- `file_base_name(`$Path$`) =` $List$: 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(`$Path$`)` = $List$: 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/`".
- `atime(`$Path$`)` = $DateTime$: This function returns the date and time that $Path$ was last accessed.
- `ctime(`$Path$`)` = $DateTime$: This function returns the date and time that $Path$ was created.
- `mtime(`$Path$`)` = $DateTime$: This function returns the date and time that $Path$ was last modified.
- `file_type(`$Path$`)` = $Term$: This returns the type of $Path$. The value returned is one of the atoms: `regular`, `directory`, `hard_link`, `symbolic_link`, `fifo`, `socket`, `block_special`, `char_special`, `message_queue`, `semaphore`, `shared_memory`, or `unknown`.
- `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?
- `directory_exists(`$Path$`)`: This tests whether $Path$ exists, and, if it exists, whether $Path$ refers to a directory.
- `link(`$Path$`)`: Does $Path$ refer to a hard link?
- `shortcut(`$Path$`)`: Does $Path$ refer to a symbolic link? This predicate might not be supported by Windows.
- `fifo(`$Path$`)`: Does $Path$ refer to a named pipe?
- `socket(`$Path$`)`: Does $Path$ refer to a socket?
- `block_special(`$Path$`)`: Does $Path$ refer to a block special file? A *special file* is used to communicate with hardware. A block special file is capable of reading or writing blocks of data at a time. This predicate might not be supported by Windows.
- `char_special(`$Path$`)`: Does $Path$ refer to a character special file? A character special can only read or write one character at a time.
- `message_queue(`$Path$`)`: Does $Path$ refer to a message queue? Message queues, as defined by POSIX, allow processes to exchange data through messages. This predicate is true if the system implements POSIX message queues and $Path$ refers to a message queue.
- `semaphore(`$Path$`)`: Does $Path$ refer to a semaphore? Semaphores are used to protect critical sections, in which race conditions can occur, in multithreaded code. This predicate is true $Path$ refers to a semaphore. For more on semaphores, see Chapter 12.

- `shared_memory(`$Path$`)`: Does $Path$ refer to a shared memory object? Shared memory objects, as defined by POSIX, allow unrelated processes to share an area of memory. This predicate is true if the system implements POSIX shared memory objects, and $Path$ refers to a shared memory object.

The following example shows how to use a few of the predicates.

Example

```
import os.

test_file(Path) =>
          if (not exists(Path)) then
              create(Path)
          elseif (directory(Path)) then
              println("Directory")
          elseif (file(Path)) then
              println("File")
          else
              println("Unknown")
          end.
```

Chapter 11

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.

11.1 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`, respectively. 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 or 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`-ports of the channels specified by X, where X can be a channel variable, a conjunction of channel variables `(`X_1`,`X_2`,...,`X_n`)`, or a disjunction of channel variables `(`X_1`;`X_2`;...;`X_n`)`. When X is a conjunction, the event is posted to the actors attached to the `dom`-ports of *all* of the channel variables; when X is a disjunction, the event is posted to actors attached to the `dom`-port of at least one of the channels. For example, suppose that an actor is attached to variable X_1, but the actor is not attached to variable X_2; then, the actor will *not* be activated by the call `post_event((`X_1`,`X_2`),`T`)`, but the actor *will* be activated by the call `post_event((`X_1`;`X_2`),`T`)`. Operationally, `post_event((`X_1`;`X_2`),`T`)` is different from posting T to X_1 and X_2 separately; if an actor is attached to both X_1 and X_2, then `post_event((`X_1`;`X_2`),`T`)` causes the actor to be activated one time, while posting T to X_1 and X_2 separately causes the actor to be activated twice.

The following built-ins are used to post events to one of a channel's four ports:

- `post_event_ins(`X`)`: posts the event atom `ins` to the `ins`-ports of the channels of X.

- `post_event_bound(`X`)`: posts the event atom `bound` to the `bound`-ports of the channels of X.
- `post_event_dom(`X`, `T`)`: posts the event T to the `dom`-ports of the channels of X.
- `post_event_any(`X`, `T`)`: posts the event T to the `any`-ports of the channels 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 channel variable has two uses: posting exclusions of inner elements from the domain, and posting general term events.

11.2 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 only contain 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 '$Head, \{Event\}, Cond => Body$', 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.

Each thread 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.

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), {event(X,T)},
    var(Flag),
=>
    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.

11.3 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 `X` is registered as an actor on the `ins`-port of `X`, and `X` 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:

```
different_terms(X,Y) =>
    different_terms(X,Y,1).

different_terms(X,Y,B),(var(X);var(Y)),{ins(X),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) then
       B=1
    else
       Bs=new_array(X.length),
       foreach(I in 1 .. X.length)
           different_terms(X[I],Y[I],B[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.

11.4 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`:

```
x_in_c_y_ac(X,Y,C),var(X),var(Y),
   {dom(Y,Ey)}
    =>
   domain_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:

```
x_add_y_eq_c_ic(X,Y,C),var(X),var(Y),
    {generated,ins(X),ins(Y),bound(X),bound(Y)}
=>
    X in C-Y.max .. C-Y.min,
    Y in C-X.max .. C-X.min.
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 `X` and `Y` are 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.

11.5 Timers and Time Events

In some applications, actors need to be activated regularly at a predefined time rate. For example, a clock animator is activated every second, and the scheduler in a time-sharing system switches control to the next process after a certain time quota elapses. To facilitate the description of time-related behavior of actors, Picat provides a module named `timer`.

The function `new_timer(`$Interval$`)` returns a timer that posts a time event at the specified time rate. A timer runs as a separate thread, and starts ticking immediately after it is created. A timer itself is a channel. It posts the event `time` to itself in every $Interval$ milliseconds. A timer T stops posting events after the predicate call `stop(`T`)`. A stopped timer can be started again. A timer is destroyed after the call `kill(`T`)` is executed.

- `new_timer(`$Interval$`) = `T: `T` is a timer that posts a time event every $Interval$ milliseconds.
- `new_timer() = `T: This is equivalent to `new_timer(200)`.
- `start(`T`)`: Start the timer T. After a timer is created, it starts ticking immediately. Therefore, it is unnecessary to start a timer with this call. This predicate is used to restart a stopped timer.
- `stop(`T`)`: Stop the timer.

- `kill(`T`)`: Kill the timer.
- `set_interval(`T`,` $Interval$`)`: Reset the interval of the timer T to $Interval$. The update is destructive, and the old value is not restored upon backtracking.
- `get_interval(`T`)` `=` $Interval$: Get the interval of the timer T.

Example

The following example shows two actors that behave in accordance with two timers.

```
go =>
    T1 = new_timer(100),
    T2 = new_timer(1000),
    T1.add_actor(ping),
    T2.add_actor(pong),
    while (true) true end.

ping,{_} => writeln(ping).

pong,{_} => writeln(pong).
```

Note that the empty `while` loop in predicate `go` is needed to let the main thread run. Without it, the query `go` would succeed before any time event is posted, meaning that neither of the actors would get a chance to be activated.

Chapter 12

Threads

With event-driven actors, a program's execution in Picat is no longer single-threaded. When an event occurs, the predicate or function that is currently being executed will be interrupted, and control will be moved to the actors that are activated. Actors run concurrently, but do not run in parallel. This means that, at any time, only one actor can run. An actor's execution can be interrupted by another event, causing a different actor to run. After all of the activated actors finish their execution, the predicate or function that was interrupted will resume its execution.

With the availability of multi-core processors, parallel processing has gained new prominence in delivering high-performance computing. The emergence of multi-core processors has the potential for breaking the performance barrier, especially for combinatorial optimization problems, which demand ever-increasing computational power. Picat's `thread` module can be used to program concurrent and parallel tasks as threads. A *thread* is represented as an attributed variable that contains, among other attributes, a thread descriptor. A thread can serve as a communication channel to which actors can be attached, and through which messages can be sent to actors running in the same or in different threads.

Each thread runs on a separate Picat virtual machine that has its own stack and heap. All threads share the following areas: symbol tables, the code area, the file and socket table, the table area, and the global map. At the implementation level, Picat synchronizes the operations of these areas in order to ensure consistency of data.

12.1 Starting and Terminating Threads

Let's begin with an example.

Example

```
import thread.

main =>
    PingThread = new_thread(ping_pong,ping,10),
    PongThread = new_thread(ping_pong,pong,10),
    PingThread.start(),
    PongThread.start().

ping_pong(Msg,N) =>
    foreach(I in 1..N)
        writeln(Msg)
```

```
end.
```

The `main` predicate creates two threads: `PingThread` and `PongThread`. After starting, `PingThread` executes the call `ping_pong(ping,10)` to display `ping` 10 times, and `PongThread` executes the call `ping_pong(pong,10)` to display `pong` 10 times. The main thread will wait until both `PingThread` and `PongThread` terminate. In the standard output, users will see the lines in a random order.

This example uses the following built-ins.

- `new_thread(`S`,` Arg_1`, ...,` Arg_n`)` `=` $Thread$: This function creates a new thread, which will execute the call

  `call(`S`,`Arg_1`,...,`Arg_n`).`

  The return value of the function `new_thread` is an attributed variable that contains, among other attributes, an attribute named `thread_id`.

- `start` ($Thread$) : This predicate puts $Thread$ into the *ready* state, allowing the scheduler to schedule it for execution.

A thread terminates in the following situations: (1) when it executes the predicate `halt`; (2) when the call that it executes succeeds, and all of the sub-threads that are created by the call have terminated; (3) when the call that it executes fails. When a thread terminates, all of its sub-threads will also terminate.

In this example, the `EchoThread` installs an actor, and then loops until `Flag` becomes a non-variable. The `SenderThread` sends `hello` to `EchoThread` three times, and then sends `done` to `EchoThread`, causing it to kill itself. After sending the messages, the `SenderThread` terminates.

12.2 Making Threads Wait

Consider the following code.

Example

```
import thread.

main =>
    Thread1 = new_thread(make_key),
    Thread1.start(),
    println(get_global_map().get(thread_key)).

make_key =>
    get_global_map().put(thread_key, 1).
```

This code creates a new thread, which puts a key-value pair in the global map, which is shared by all threads. The main thread is supposed to print the value that the other thread, `Thread1`, stores in the global map. However, the main thread and `Thread1` are executing in parallel, so there is no guarantee that `Thread1` will have stored `thread_key` on the global map by the time that the main thread tries to retrieve the key's value. If the main thread calls `println` before

`Thread1` stores `thread_key` on the global map, then the program will raise the exception `key_not_found(thread_key, main)`.

The solution to this problem is to have the `main` thread wait for `Thread1` to finish execution. This is accomplished by using the `join` predicate, as in the following code.

Example

```
import thread.

main =>
    Thread1 = new_thread(make_key),
    Thread1.start(),
    join(Thread1),
    println(get_global_map().get(thread_key)).

make_key =>
    get_global_map().put(thread_key, 1).
```

The line `join(Thread1)` causes the `main` thread to wait for `Thread1` to finish execution. Then, assuming that no error has occurred, `thread_key` will be stored on the global map before the `main` thread tries to access it.

The `thread` module has two predicates that cause a thread to wait.

- `join(`$Thread$`)`: This predicate causes the current thread to wait until $Thread$ finishes running. This means that the current thread does not execute any more code until $Thread$ terminates.
- `sleep(`$Milliseconds$`)`: This predicate causes the current thread to pause execution until $Milliseconds$ time has passed.

12.2.1 Deadlock

The `thread` module has a `this_thread()` function, which returns the thread that calls the function.

- `this_thread() =` $Thread$: The built-in function `this_thread()` returns the executing thread of the function call.

Suppose that a thread calls `join(this_thread())`. This causes a thread to wait for itself before the thread continues to run. This code is the simplest example of a *deadlock*. Deadlocks occur when at least one thread is waiting forever, consuming resources.

12.3 Mutual Exclusion

Mutual exclusion occurs when multiple threads need to access a shared resource, but only one thread can access the resource at any given time.

Consider the following code.

Example

```
import thread, io.

main =>
    FD = io.open("threads.txt", write),
    Thread1 = new_thread(print_val, 1, FD),
    Thread2 = new_thread(print_val, 2, FD),
    Thread1.start(),
    Thread2.start(),
    join(Thread1),
    join(Thread2),
    io.close(FD).

print_val(I, FD) =>
    io.fprintf(FD, "Thread %d is writing %d.", I, I).
```

In this code, both `Thread1` and `Thread2` are accessing `threads.txt`. It is possible for both threads to write to the file at the same time, causing their output to interleave. For example, after the above code executes, the contents of `threads.txt` can be:

```
Thread 1 is wriThread2 isting writing 2
1
```

This occurs because both threads are simultaneously trying to access a shared resource, `threads.txt`. In order to solve this problem, and to allow mutual exclusion, access to shared resources, such as variables and files, should be placed in a *critical section* of code, which can only be accessed by one thread at a time. The `thread` module has four ways to provide mutual exclusion: using a *mutex*, using a *semaphore*, using a *read-write lock*, and using a *condition variable*.

12.3.1 Mutex Locks

A *mutex* is a structure that has two possible states: *locked* and *unlocked*. When a thread wants to access a critical section of code, it tries to acquire the mutex. If the mutex is unlocked, then the thread acquires the mutex immediately, and locks the mutex; after the thread leaves the critical section, it unlocks the mutex. Otherwise, if the mutex is locked, then that means that another thread has locked the mutex, and is executing the critical section; in this case, the thread that is currently trying to acquire the mutex will block until the mutex is unlocked.

Picat has three built-ins that manage mutex locks.

- `new_mutex() =` $Mutex$: This function creates a new mutex lock.
- `acquire_mutex(`$Mutex$`)`: This predicate is used to acquire a mutex lock. If the mutex is currently in the `unlocked` state, then the current thread acquires the mutex and locks it. Otherwise, if the mutex is in the `locked` state, then the current thread waits until the mutex is unlocked, after which the thread tries again to acquire the mutex.
- `release_mutex(`$Mutex$`)`: This predicate is used to release a mutex lock. It unlocks the mutex, allowing waiting threads, which are trying to acquire the mutex, to continue running. Note that a mutex can only be released by the thread that has acquired the mutex.

For example, the following code modifies the `print_val` example to use a mutex lock.

Example

```
import thread, io.

main =>
    FD = io.open("threads.txt", write),
    Mutex = new_mutex(),
    Thread1 = new_thread(print_val, Mutex, 1, FD),
    Thread2 = new_thread(print_val, Mutex, 2, FD),
    Thread1.start(),
    Thread2.start(),
    join(Thread1),
    join(Thread2),
    io.close(FD).

print_val(Lock, I, FD) =>
   acquire_mutex(Lock), % Enter critical section
   output_val(I, FD),
   release_mutex(Lock). % Leave critical section

output_val(I, FD) =>  % Critical Section
    io.fprintf(FD, "Thread %d is writing %d.", I, I).
```

This code locks the mutex before writing to `threads.txt`, meaning that the output of the threads will not interleave.

Deadlock

It is possible for mutex locks to cause deadlocks, in which two or more threads are waiting for each other to release mutex locks.

Example

```
import thread.

main =>
    Mutex1 = new_mutex(),
    Mutex2 = new_mutex(),
    Thread1 = new_thread(go, Mutex1, Mutex2),
    Thread2 = new_thread(go, Mutex2, Mutex1),
    Thread1.start(),
    Thread2.start(),

go(First_Lock, Second_Lock) =>
    acquire_mutex(First_Lock),
    acquire_mutex(Second_Lock),
    critical_code(), % user-defined critical section
    release_mutex(Second_Lock),
    release_mutex(First_Lock).
```

In this example, it is possible for `Thread1` to acquire `Mutex1` while `Thread2` acquires `Mutex2`. Then, `Thread1` waits forever for `Thread2` to release `Mutex2`, while `Thread2` waits forever for `Thread1` to release `Mutex1`. This causes a deadlock. One way to avoid this problem is to ensure that all threads acquire locks in the same order. For example, if both `Thread1` and `Thread2` would try to acquire `Mutex1` before `Mutex2`, then the thread that acquires `Mutex1` can also acquire `Mutex2`, and can continue running while the other thread waits.

Starvation

Another problem that can occur with mutex locks is *starvation*, where a thread can wait forever to access a critical section. For example, consider the following code fragment.

```
while (true)
    acquire_mutex(Mutex),
    critical_code(), % user-defined critical section
    release_mutex(Mutex)
end.
```

If multiple threads are executing this while loop, then it is possible for the system to give these threads different priorities. It is possible for a low-priority thread to wait forever to acquire `Mutex`, while threads with higher priorities repeatedly acquire and release the mutex.

12.3.2 Semaphores

A *semaphore* is a structure that contains an integer. Unlike mutex locks, which only have two states, the semaphore's integer can any value between 0 and the maximum integer to which the semaphore is initialized. When a thread wants to access a critical section of code, it tries to access the semaphore by decreasing the semaphore's integer value. If the integer value is greater than 0, then the value is decreased by one, and the thread can access the critical section. Otherwise, the integer value is currently 0, and the thread must wait for the integer value to increase; then, the thread again tries to decrease the semaphore's integer value and continue.

Picat has four built-ins that manage semaphores.

- `new_semaphore() =` $Semaphore$: This function creates a new semaphore, with an initial integer value of 1.
- `new_semaphore(`N`) =` $Semaphore$: This function creates a new semaphore, with an initial integer value of N.
- `p_semaphore(`$Semaphore$`)`: This predicate tries to decrease the semaphore's integer value. If the value is currently non-zero, then this predicate decreases the value by one, and the current thread continues. Otherwise, the value is zero, and the current thread waits until the value is increased, and then tries again to decrease the semaphore's integer value.
- `v_semaphore(`$Semaphore$`)`: This predicate increases the semaphore's integer value by one. If the value was zero, then threads that have blocked while calling `p_semaphore` will try again to decrease the semaphore's integer value.

For example, the following code modifies the `print_val` example to use a semaphore.

Example

```
import thread, io.

main =>
    FD = io.open("threads.txt", write),
    Semaphore = new_semaphore(),
    Thread1 = new_thread(print_val, Semaphore, 1, FD),
    Thread2 = new_thread(print_val, Semaphore, 2, FD),
    Thread1.start(),
    Thread2.start(),
    join(Thread1),
    join(Thread2),
    io.close(FD).

print_val(Semaphore, I, FD) =>
   p_semaphore(Semaphore), % Enter critical section
   output_val(I, FD),
   v_semaphore(Semaphore). % Leave critical section

output_val(I, FD) =>   % Critical Section
    io.fprintf(FD, "Thread %d is writing %d.", I, I).
```

Differences Between Mutex Locks and Semaphores

A mutex lock has two states: `locked` and `unlocked`. This means that only one thread can access the shared code at any time. A semaphore can take multiple values. If a semaphore is initialized with `new_semaphore(`N`)`, then up to N threads can access the shared code at any time.

Mutex locks can cause starvation, because a mutex can only be unlocked by the thread that locked it. Semaphores decrease the problem of starvation, because multiple threads can call `v_semaphore`, allowing a low-priority thread to have a greater possibility of entering a shared section of code.

12.4 Read-Write Locks

Sometimes, when multiple threads need to access a file, using a mutex lock can have a large amount of overhead. For example, when multiple threads need to read from a file, if each thread uses a mutex before reading, then the other threads will need to wait, even though the file is not being modified.

This problem is solved by using a *read-write lock*. A read-write lock is a structure that has three possible states: *read_locked*, *write_locked*, and *unlocked*. Read-write locks allow multiple readers to acquire the lock at the same time, unless a writer has acquired the lock. In other words, at any given time, if the lock is locked, then either a single writer has acquired the lock, or one or more readers have acquired the lock. This means that readers only need to wait when another thread is writing.

Picat has four built-ins that manage read-write locks.

- `new_rwlock()` = *RW Lock*: This function creates a new read-write lock.

- `rdlock(`*RWLock*`)`: This predicate is used to acquire a read-write lock for reading. If the read-write lock is currently in the `unlocked` or the `read_locked` state, then the current thread acquires the read-write lock, and locks it for reading. Otherwise, if the read-write lock is in the `write_locked` state, then the current thread waits until the read-write lock is either unlocked, or locked for reading, after which the thread tries again to acquire the read-write lock.

- `wrlock(`*RWLock*`)`: This predicate is used to acquire a read-write lock for writing. If the read-write lock is currently in the `unlocked` state, then the current thread acquires the read-write lock, and locks it for writing. Otherwise, if the read-write lock is in the `read_locked` or the `write_locked` state, then the current thread waits until the read-write lock is unlocked, after which the thread tries again to acquire the read-write lock.

- `rwunlock(`*RWLock*`)`: This predicate is used to release a read-write lock. It unlocks the read-write lock. If no other threads have acquired the read-write lock for reading, then, waiting threads, which are trying to acquire the read-write lock, can continue running.

The following code shows how to use a read-write lock to manage two threads that are reading from a file, and one thread that is writing to a file.

Example

```
import thread, io.

main =>
    FD = io.open("threads.txt", append),
    RWLock = new_rwlock(),
    Thread1 = new_thread(write_vals, RWLock, FD),
    Thread2 = new_thread(read_all, RWLock, FD),
    Thread3 = new_thread(read_all, RWLock, FD),
    Thread1.start(),
    Thread2.start(),
    Thread3.start(),
    join(Thread1),
    join(Thread2),
    join(Thread3),
    io.close(FD).

write_vals(RWLock, FD) =>
   wrlock(RWLock),   % Enter critical section
   foreach (I in 1 .. 1000000)
       io.fprintln(FD, I)
   end,
   wrunlock(RWLock). % Leave critical section

read_all(RWLock, FD) =>
    rdlock(RWLock),   % Enter critical section
    Str = io.fread_file_chars(FD),
    wrunlock(RWLock). % Leave critical section
```

In this example, it is possible for `Thread2` and `Thread3` to acquire `RWLock` at the same time. However, if `Thread1` has already aquired `RWLock`, then `Thread2` and `Thread3` must wait until the lock is unlocked.

12.5 Condition Variables

Consider the following code.

Example

```
import thread.

main =>
    get_global_map().put(X, 0),
    Mutex = new_mutex(),
    Thread1 = new_thread(change_map, Mutex),
    Thread2 = new_thread(blastoff, Mutex),
    Thread1.start(),
    Thread2.start().

change_map(Mutex) =>
    X1 = 0,
    foreach (I in 1 .. 1000000)
       process_code(), % user-defined code
       acquire_mutex(Mutex),
       X1 := get_global_map().get(X) + 1,
       get_global_map().put(X, X1),
       release_mutex(Mutex)
    end.

blastoff(Mutex) =>
    acquire_mutex(Mutex),
    X = get_global_map.get(X),
    release_mutex(Mutex),
    while (X != 1000000)
           acquire_mutex(Mutex),
           X := get_global_map.get(X),
           release_mutex(Mutex).
    end,
    println("Blastoff!").
```

In this code, `Thread2` repeatedly tests the value of `X` in the global map, until `X` is set to 1000000. This is called *busy waiting*, and wastes CPU processing time. Furthermore, if `Thread2` has a higher priority than `Thread1`, then `Thread1` can be starved, meaning that `Thread1` might never acquire `Mutex`, and the value of `X` might never reach 1000000. It would be more efficient for `Thread2` to block until `X` is set to 10000000 instead of repeatedly locking a mutex and testing the global map.

This problem is solved by using a *condition variable*. A condition variable is used together with a mutex. After a thread has acquired the mutex, the thread tests a condition, and waits to be signaled by the condition variable. The condition variable suspends the thread, and unlocks

the mutex. When the condition variable signals the thread, the thread automatically reacquires the mutex. This decreases the amount of busy waiting, and allows other threads to access shared resources until the condition is true.

Picat has four built-ins that manage condition variables.

- `new_cv() =` CV: This function creates a new condition variable.
- `wait_cv(`CV`,` $Mutex$`)`: When a thread calls this predicate, the thread is suspended, and the mutex is temporarily released, until the condition variable signals the thread.
- `signal_cv(`CV`)`: After a thread modifies a value which is associated with a condition, the thread can call this predicate. This predicate wakes at least one thread that is waiting for the condition variable. The thread reacquires its mutex, and tests the condition again.
- `broadcast_cv(`CV`)`: After a thread modifies a value which is associated with a condition, the thread can call this predicate. This predicate wakes all of the threads that are waiting for the condition variable. Each thread reacquires its mutex, and tests the condition again. If more than one thread has released the same mutex, then only one of the threads can continue execution, and the other threads continue blocking until they can reacquire the mutex.

The following code shows how to modify the `blastoff` example in order to eliminate busy waiting.

Example

```
import thread.

main =>
    get_global_map().put(X, 0),
    Mutex = new_mutex(),
    CV = new_cv(),
    Thread1 = new_thread(change_map, Mutex, CV),
    Thread2 = new_thread(blastoff, Mutex, CV),
    Thread1.start(),
    Thread2.start().

change_map(Mutex, CV) =>
    X1 = 0,
    foreach (I in 1 .. 1000000)
       process_code(), % user-defined code
       acquire_mutex(Mutex),
       X1 := get_global_map().get(X) + 1,
       get_global_map().put(X, X1),
       release_mutex(Mutex)
    end,
    signal_cv(CV).

blastoff(Mutex, CV) =>
    acquire_mutex(Mutex),
    X = get_global_map.get(X),
```

```
    release_mutex(Mutex),
    while (X != 1000000)
           wait_cv(CV, Mutex),
           acquire_mutex(Mutex),
           X := get_global_map.get(X),
           release_mutex(Mutex).
    end,
    println("Blastoff!").
```

In this code, the instruction `wait_cv(CV, Mutex)` blocks `Thread2` until `Thread1` calls `signal_cv(CV)`. Then, even if `Thread2` has a higher priority, and has already acquired `Mutex`, the condition variable causes `Thread2` to release the mutex, allowing `Thread1` to run.

Note that `wait_cv` should be called within the loop that tests the condition. It is possible for a thread to be signaled when the condition has not yet been fulfilled. Therefore, the thread must test the condition after the thread has been signaled. If the condition has not been fulfilled, then the thread is blocked again, until the next time that the thread is signaled.

Chapter 13

Processes

As an alternative to threads, Picat has a `process` module, which allows the user to create new processes.

13.1 Creating New Processes

Let's begin with an example.

Example

```
import process.

ex1 =>
    Id = fork(),
    if Id == 0 then
       printf("I am the child process, with process ID %d, ", pid()),
       printf("and parent process %d.%n", ppid())
    else
       printf("I am the parent process, with process ID %d, ", pid()),
       printf("and child process %d.%n", Id)
    end.

ex2 =>
    new_process(execute, "ls, -l"),
    printf("I have created a new process").
```

The `ex1` predicate uses `fork` in order to create a new *child* process. The new process has the original process, which called `fork`, as its *parent*. The `fork` function returns 0 to the child process, and returns the child's process ID to the parent process. The child process prints its ID followed by the parent's ID, while the parent process prints its ID followed by the child's ID. In the standard output, users will see the lines in a random order.

The `ex2` predicate uses `new_process` in order to create a new process, which immediately runs the "`ls -l`" command. Meanwhile, the original process prints a string to standard output. In the standard output, users will see the parent's output and the child's output in a random order.

These examples use the following built-ins.

- `fork() =` ID: This function creates a new process, which continues running the same code from the point where `fork` was called. The new process receives a separate copy

of the parent process's memory, including the instantiated variables, the free variables, and the file descriptor table. The `fork` function returns 0 to the new process, and returns the process ID of the new process to the parent process. If `fork` fails, then it throws an error.

- `new_process(`S`, `Arg_1`, ..., `Arg_n`) = `ID: This function creates a new process, which will execute the call

  `call(`S`,`Arg_1`,...,`Arg_n`).`

  The new process has a separate copy of the parent process's memory. The `new_process` function returns the same values that `fork` returns. The `exec` predicate will be discussed in Section 13.2.

- `pid() = `ID: This function returns the ID of the current process. The ID number is an integer.

- `ppid() = `ID: This function returns the ID of the current process's parent process. The ID number is an integer.

13.2 Executing Other Code

The `ex2` example showed how to execute different code in a child process. The following examples illustrate two other functions that allow a process to run different code.

Example

```
import process.

ex3 =>
    Id = fork(),
    if Id == 0 then
       exec("ls", "-l", "*.pdf")
    else
       printf("I am the parent process, with process ID %d, ", pid()),
       printf("and child process %d.%n", Id)
    end.

ex4 =>
    Id = fork(),
    if Id == 0 then
       execl("ls", ["-l", "*.pdf"])
    else
       printf("I am the parent process, with process ID %d, ", pid()),
       printf("and child process %d.%n", Id)
    end.
```

In both of these examples, the parent process forks a new process, which will run the "`ls` command, with the parameters "`-l`" and "`*.pdf`".

These examples use the following built-ins.

- `exec(`S`, `Arg_1`, ..., `Arg_n`)`: This executes S with the parameters Arg_1, ..., Arg_n. Unlike the `call` predicate, the S argument to `exec` either specifies a command that should be run, or specifies the path to a file that should be run. Note that each argument, Arg_i, must be a string. If `exec` fails, then an error is thrown.

- `execl(`S`, `$ArgList$`)`: This is similar to `exec`, except that the second parameter is a list, which contains the arguments that should be passed to S. Each member of $ArgList$ must be a string.

As shown in Chapter 9, a process's file descriptor table stays the same when `exec` or `execl` is called. This means that if standard input, standard output, or standard error was redirected to a file before `exec` or `execl` is called, then the code that is executed will also redirect to the same file.

13.3 Making Processes Wait

Sometimes, it is necessary for the parent process to wait for one or more of its child processes to finish, allowing the parent process to determine the exit status of the child processes for which it waits.

The `process` module includes two built-ins for allowing a parent process to wait for its children.

- `wait() =` $StatMap$: This causes the parent process to wait until one of its children finishes running. The `wait` function returns a map with the keys `pid` and `status`. The `pid` key indicates the child process's process number. The `status` key refers to an integer that indicates the child process's exit status. If `wait` fails, then an error is thrown.

- `waitpid(`ID`) =` $StatMap$: This causes the parent process to wait for the process that has the specified ID number. If ID is 0 or -1, then this function is the same as `wait`. This function returns a map with the keys `pid` and `status`. If `waitpid` fails, then an error is thrown.

The following code demonstrates these functions.

Example

```
import process.

ex5 =>
    foreach (I in 1..5)
        if fork() == 0 then
           printf("Process %d%n.", pid())
        end
    end,
    Stat = wait(),
    printf("Process %d has status %d", Stat.pid, Stat.status).

ex6 =>
    L = [],
    foreach (I in 1..5)
        F = fork(),
```

```
        if F == 0 then
           printf("Process %d%n.", pid())
        else
           L := [F | L]
        end
    end,
    Index = math.randrange(1, length(L) + 1),
    Stat = waitpid(L[Index]),
    printf("Process %d has status %d", Stat.pid, Stat.status).
```

The `ex5` example causes the parent process to wait and get the exit status of one of its child processes. The `ex6` example chooses the ID of a random one of the the child processes, and passes the ID to `waitpid`.

13.4 Differences Between Processes and Threads

There are a number of differences between processes and threads.

Each process runs at least one thread. In addition, each process has its own memory. This means that each process has its own copy of instantiated variables and free variables, and that each process has its own file descriptor table.

A thread is a light-weight process. Threads from the same process share the same memory space. This means that each thread accesses the same copy of instantiated variables and free variables. Furthermore, threads from the same process share the same file descriptor table.

An advantage of processes is that, since processes have their own memory space, the synchronization of processes is not as difficult as the synchronization of threads. An advantage of threads is that threads share memory, meaning that threads do not have as much memory overhead as processes have. Another advantage of threads is that threads multitask. If a process is slow, and the process is timeshared with other processes, then the process loses its timeslot; however, if a thread within a process is slow, then other threads in the same process can still run.

Chapter 14

Constraints

Picat provides three solver modules, including `cp`, `sat`, and `mip`, for modeling and solving constraint satisfaction and optimization problems. All three of these modules implement the same set of built-in constraints. The `cp` and `sat` modules support integer-domain variables, and the `mip` module also supports real-domain variables. In order to use a solver, users must first import the module. As the three modules have the same interface, this chapter describes the three modules together. Figure 14.1 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 three of the 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 14.1: Constraint operators in Picat

Precedence	Operators
Highest	`in`, `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

```
import cp.

queens(N) =>
    Qs=new_array(N),
    Qs in 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).
```

This program imports the `cp` module in order to solve the N-queens problem. The same program runs with the SAT solver if `sat` is imported, or runs with the LP/MIP solver if `mip` is imported. The predicate `Qs in 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)`.

14.1 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`. Integer-domain variables are declared with the built-in predicate $Vars$ `in` Exp. The `mip` module also supports real domains. Real-domain variables are declared with the built-in predicate `lp_in(`$Vars$`,`$LExp$`,`$UExp$`)`. A variable has a default domain. The `cp` and `sat` modules assume the default domain to be `-268435455..268435455`, and the `mip` module assumes the default domain to be from `0.0` to `inf`. Integer domains must be explicitly declared when `mip` is used.

- $Vars$ `in` Exp: This predicate restricts the domain or domains of $Vars$ to Exp. $Vars$ and Exp can be one of the following: (1) $Vars$ is a single variable, and Exp is an expression that returns a list of integers; (2) $Vars$ is a list or an array of variables, and Exp is an expression that returns a list of integers; (3) $Vars$ is a tuple of variables in the form `(`$V1$`,...,`Vn`)`, and Exp is a list of tuples of integers; or (4) $Vars$ is a list of tuples of variables, and Exp is a list of tuples of integers. In cases (3) and (4), the predicate specifies a positive table constraint.

- `lp_in(`$Vars$`,` $LExp$`,` $UExp$`)`: This predicate restricts the domain or domains of $Vars$ to the interval with the lower bound $LExp$ and the upper bound $UExp$. $Vars$ is either one variable, a list of variables, or an array of variables, and $LExp$ and $UExp$ are real expressions. The lower bound $LExp$ must be greater than or equal to 0.

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 $Vars$ `in` 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 $FDVar$. The return value is always 0 if `sat` or `mip` is used.

- `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 next element of 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 previous element of 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.
- `fd_size(`$FDVar$`) = `$Size$: This function returns the size of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.
- `fd_superset(`$FDVar1$`, `$FDVar2$`)`: This predicate is true if $FDVar1$'s domain is a superset of $FDVar2$'s domain.
- `fd_true(`$FDVar$`, `Elm`)`: This predicate is true if the integer Elm is an element in the domain of $FDVar$.
- `fd_var(`$Term$`)`: This predicate is true if $Term$ is an integer-domain variable.
- `new_fd_var() = `$FDVar$: This function creates a new domain variable with the domain `-268435455..268435455`.

14.2 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 $DVars$ `in` 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 of integers in which each tuple takes the form $(a_1, \ldots, a_n)$. A negative constraint takes the form $DVars$ `notin` R.

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 `char_code(`$Char$`)` returns the code of $Char$, and that the function `code_char(`$Code$`)` returns the character of $Code$.

```
    crossword(Vars):-
        Vars=[X1,X2,X3,X4,X5,X6,X7],
        Words2=[(char_code('I'),char_code('N')),
                (char_code('I'),char_code('F')),
(char_code('A'),char_code('S')),
(char_code('G'),char_code('O')),
(char_code('T'),char_code('O')]),
        Words3=[(char_code('F'),char_code('U'),char_code('N')),
(char_code('T'),char_code('A'),char_code('D')),
(char_code('N'),char_code('A'),char_code('G')),
(char_code('S'),char_code('A'),char_code('G'))],
        [(X1,X2),(X1,X3),(X5,X7),(X6,X7)] in Words2,
        [(X3,X4,X5),(X2,X4,X6)] in Words3,
        solve(Vars),
        writeln([code_char(Code) : Code in Vars]).
```

14.3 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), `/` (division), `//` (integer division), `div` (integer division), `mod`, `**` (power), `abs`, `avg`, `min`, `max`, and `sum`. Except for index notations 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:

- `cond(`$BoolConstr$`,` $ThenExp$`,` $ElseExp$`)`: This expression is the same as $BoolConstr$`*`$ThenExp$`+(1-`$BoolConstr$`)*`$ElseExp$.
- `min(`$DVars$`)`: The minimum of $DVars$, where $DVars$ is a list or an array of domain variables.
- `max(`$DVars$`)`: The maximum of $DVars$, where $DVars$ is a list or an array of domain variables.
- `min(`$Exp1$`,` $Exp2$`)`: The minimum of $Exp1$ and $Exp2$.
- `max(`$Exp1$`,` $Exp2$`)`: The maximum of $Exp1$ and $Exp2$.
- `sum(`$DVars$`)`: The sum of $DVars$, where $DVars$ is a list or an array of domain variables.

When a constraint occurs in an arithmetic expression, it is 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] in 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.

14.4 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 `in` $Domain$ or Var `notin` $Domain$), or a Boolean constraint. As shown in Table 14.1, the operator #~ has the highest precedence, 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.

14.5 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 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.

Picat provides the following global constraints.

- `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_distinct(`$FDVars$`)`: This constraint is the same as `all_different`, but 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,
    Xs in 1..N,
    Ys in 1..N,
    foreach(I in 1..N, J in 1..N)
        X[I] #= J #<=> Y[J] #= I
    end.
```

- `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).
```

- `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.

- `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(`$Tasks$`)`: $Tasks$ 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 `disj_tasks(`S_1`,` D_1`,` S_2`,` D_2`)` in $Tasks$, but it may be more efficient, because it converts the disjunctive tasks into global constraints.

- `element(`I`,` $List$`,` V`)`: This constraint is true if the Ith element of $List$ is V, where I and V are integer-domain variables, and $List$ is 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) =>
    foreach(Key=V in Pairs)
        sum([E#=Key : E in List]) #= V
    end.
```

- `neqs(`$NeqList$`)`: $NeqList$ 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.

- `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.

14.6 Solver Invocation

- `solve(`$Options$`,` $Vars$`)`: This predicate calls the imported solver to label the variables $Vars$ with values, where $Options$ is a list of options for the solver. The options will be detailed below. For `cp`, this predicate can be called multiple times for a problem, and each call can backtrack in order to find multiple solutions. However, for `mip` and `sat`, this predicate can be called only once for a problem, and no call to it can backtrack.

- `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`])`.

14.6.1 Common Solving Options

The following options are accepted by all three of the solvers.

- `min(`Var`)`: Minimize the variable Var.
- `max(`Exp`)`: Maximize the variable Var.
- `dump(`Exp`)`: Dump the model in some format to the standard output. If `cp` is used, the output is a Picat predicate; if `mip` is used, then the output is in the CPLEX lp format; if `sat` is used, then the output is in CNF. Note that the predicate `solve(`$Options$`,`$Vars$`)` preprocesses the accumulated constraints before dumping the model. If any constraint fails during preprocessing, then the predicate `solve(`$Options$`,`$Vars$`)` fails, and no model will be dumped. Also note that the solver is not called to solve the problem, and that the variables $Vars$ can only be instantiated during preprocessing.
- `dump(`$File$`)`: Dump the model to $File$.

14.6.2 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]`.
- `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.
- `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.

Chapter 15

Sockets

Sockets allow communication across computer networks. Picat's `socket` module enables both connection-oriented and connectionless communication. This module supports communication in the Internet domain and in the Unix domain. All Picat programs that use sockets must import the `socket` module.

15.1 Connection-Oriented Communication

Connection-oriented communication uses TCP, the Transmission Control Protocol. Before data is transmitted, a client and a server establish a full-duplex connection. TCP is reliable, meaning that it checks for errors during transmission, and that it sequences packets that arrive in the wrong order. The following code skeleton shows how a server and a client communicate over TCP.

Example

```
import socket, process.

server =>
     FD = socket(inet, stream),
     Port = bind(FD, inet, inaddr_any, 0),
     listen(FD), % wait for connection
     do          % infinite loop
          Client = accept(FD),
          P = process.fork(),
          if P == 0 then     % child process
             echo(Client.client_fd)
          else               % parent process
             close(Client.client_fd)
          end
     while (true).

client(Address, Port) =>
     FD = socket(inet, stream),
     connect(FD, inet, Address, Port),
     hello_to_server(FD),
     close(FD).
```

```
echo(Client) => % Server code to communicate with client
     % Begin communication
     send(Client, "Enter Input:"),
     Str = recv(Client), % wait for client input
     send(Client, Str),
     % End communication
     close(Client).

hello_to_server(FD) => % Client code to communicate with server
     % Begin communication
     Str = recv(FD), % wait for server's message
     println(Str)
     send(FD, "Hello"),
     Str := recv(FD),
     println(Str).
     % End communication
```

The server performs the following steps:

1. The server creates a communication endpoint, using the `socket` function.
2. The server associates itself with a port number, using `bind`.
3. The server waits for connections, using the `listen` predicate.
4. When a connection request arrives, the server handles the connection, using the `accept` function.
5. For connection-oriented communication, multiple messages can be exchanged. Therefore, if the server is capable of receiving multiple requests from different clients, the server can fork a new process for each request. The new process communicates with the client by using `send` and `recv`, while the parent process closes its copy of the client's file descriptor, and continues listening for other clients. Otherwise, the server receives one request at a time, meaning that there is only one process, which finishes communicating with one client before listening for another client.

The client performs the following steps:

1. The client creates a communication endpoint, using the `socket` function.
2. The client tries to connect to the server, using the `connect` predicate.
3. After the server accepts the connection request, the client communicates with the server by using `send` and `recv`.
4. When communication is complete, the client closes the socket's file descriptor.

The following functions and predicates allow a server and a client to communicate over TCP:

- `socket(`$Domain$`, `$Type$`) = `FD: This returns a file descriptor for the communication endpoint. The $Domain$ parameter can be `inet`, `inet6`, or `unix`. For communication over the Internet domain, use either `inet` or `inet6`. The domain `unix` allows communication over the Unix domain. The domain `inet` uses IP version 4, and the domain `inet6` uses IP version 6. The $Type$ parameter can be one of the following five atoms:

`stream`, `dgram`, `raw`, `seqpacket`, or `rdm`. The `stream` atom allows connection-oriented communication over TCP. The `dgram` atom allows connectionless communication over UDP. The other atoms are used as follows: `raw` is used for checking communication paths, `seqpacket` allows reliable and bidirectional transmission of packets, and `rdm` is used for reliably-delivered messages.

- `tcp_socket() =` FD: This performs the same operation as `socket(inet, stream)`. It creates a TCP socket that uses IPv4.

- `bind(`FD`, `$INet$`, `$Address$`, `$Port$`)`: This associates a communication endpoint with a port number. The FD parameter is the file descriptor for the communication endpoint that the `socket` function returns. In the Internet domain, the $INet$ parameter must be either `inet`, which specifies that IPv4 is used, or `inet6`, which specifies that IPv6 is used. The $Address$ parameter is a string that specifies the IP address that will be used for the connections. If IPv4 is used, then $Address$ is an address string in dotted-decimal notation, such as "127.0.0.1". If IPv6 is used, then $Address$ is an address string in hexadecimal notation, such as "2001:DB8:0:0:8:800:200C:417A". The $Address$ parameter can be the atom `inaddr_any`, which indicates that the server will listen for connections on any of the network interfaces. It can also be the atom `localhost`, which represents the address of the local machine. The $Port$ parameter is an integer that specifies the port that should be used; set $Port$ to 0 in order to allow the operating system to pick the port that should be used.

- `tcp_bind(`FD`, `$Address$`, `$Port$`)`: This is the same as `bind(`FD`, inet, `$Address$`, `$Port$`)`. It uses IPv4.

- `listen(`FD`, `$Backlog$`)`: This is only used by the server. Listening causes the server to wait for incoming connection requests. The FD parameter is the server's file descriptor, which the `socket` function returns. The $Backlog$ parameter specifies the maximum number of pending client connection requests.

- `listen(`FD`)`: This is only used by the server. This function uses a default backlog of 5.

- `accept(`FD`) =` $Client$: This is only used by the server. This function accepts incoming connections. The FD parameter is the server's file descriptor, which the `socket` function returns. The `accept` function returns a map with the keys `client_fd`, `client_domain`, `client_address`, and `client_port`. The key `client_fd` stores the file descriptor that is used to communicate with the client. The key `client_domain` indicates the domain that the client is using to communicate, as specified in the `socket` function. The key `client_address` is a string that stores the client's IP address. The key `client_port` stores the port number that the client is using to receive data from the server.

- `connect(`FD`, `$INet$`, `$Address$`, `$Port$`)`: This is only used by the client. It causes the client to send a connection request to the server. The FD parameter is the client's file descriptor, which the `socket` function returns. In the Internet domain, the $INet$ parameter is either the atom `inet`, specifying that IPv4 is used, or the atom `inet6`, specifying that IPv6 is used. The $Address$ parameter is a string that specifies the server's IP address, as described in the explanation of `bind`. The $Address$ parameter can be the atom `localhost` in order to represent the address of the local machine. The `connect` predicate requires a $Port$ parameter in the Internet domain, specifying the number of the port on which the server will receive the connection.

- `tcp_connect(`FD`, `$Address$`, `$Port$`)`: This performs the same operation as `connect(`FD`, inet, `$Address$`, `$Port$`)`. It connects TCP sockets that use IPv4.
- `send(`FD`, `$Message$`) = `$NBytes$: This is used for reliable communication. It blocks until it is able to send the message. The FD parameter is the file descriptor that the `socket` function returns. The $Message$ parameter is a string that one endpoint is transmitting to the other endpoint. The `send` function returns the number of bytes that were sent.
- `send(`FD`, `$Message$`, `$Flags$`) = `$NBytes$: The $Flags$ parameter is a list of options. The options can be `oob`, `dontroute`, `dontwait`, and `nosignal`. The option `oob` is used to indicate out-of-band urgent data. The `dontroute` option indicates that the data should not be sent over a router. The `dontwait` option indicates that, instead of blocking, if `send` cannot immediately transmit the message, then it should throw an error. The `nosignal` option prevents a signal from being raised if a message is sent to a host that is not receiving. Windows only supports the `oob` and `dontroute` options.
- `recv(`FD`) = `$Message$: This is used for reliable communication. It blocks until data arrives. The FD parameter is the file descriptor that the `socket` function returns. The `recv` function blocks until data is received, and returns the message that was received. The `recv` function returns a message string.
- `recv(`FD`, `$Flags$`) = `$Message$: The $Flags$ parameter is a list of options. The options can be `oob`, `peek`, `waitall(Length)`, and `dontwait`. The option `oob` is used to receive out-of-band urgent data. The `peek` atom indicates that the receiver should look at the message without removing it from the incoming message buffer. The `waitall(Length)` option indicates that the function should block until $Length$ bytes are received. The option `dontwait` indicates that the function should return immediately, even if data is unavailable.
- `close(`FD`)`: This performs the same operation as the `close` predicate in the `io` module.

Binding and connecting in the Unix domain will be discussed in Section 15.4.

The endpoints can also communicate by using the built-ins from the `io` module. However, further management is required. For example, the `fread` functions are non-blocking. If the other communication endpoint did not yet send data, the `fread` functions would return `eof` immediately.

The following example shows how a client and server would send data to each other by using `io` built-ins.

Example

```
import io.

echo(Client) =>
     % Begin communication
     io.fprintln(Client, "Enter Input:"),
     Str = io.fread_file(Client),
     while (Str == eof)
            Str := io.fread_file(Client)
     end,
     io.fprintln(Client, Str),
     % End communication
```

```
     close(Client).

hello_to_server(FD) =>
     % Begin communication
     Str = io.fread_file(FD),
     while (Str == eof)
            Str := io.fread_file(FD)
     end,
     println(Str),
     io.fprintln(FD, "Hello"),
     Str = io.fread_file(FD),
     while (Str == eof)
            Str := io.fread_file(FD)
     end,
     println(Str).
     % End communication
```

15.2 Connectionless Communication

There are times when connection-oriented communication has too much overhead. The transmission reliability is not necessary, faster connectionless communication can be used. Connectionless communication occurs over UDP, the User Datagram Protocol. Unlike TCP, UDP does not require connection establishment and termination. UDP is unreliable, meaning that there can be errors during transmission, and data can arrive out of order. UDP can be used with applications that only require a simple request and response, such as DNS or NTP, multicasting communication, and real-time applications, such as video. The following code skeleton shows how a server and a client communicate over UDP.

Example

```
import socket.

server =>
     FD = socket(inet, dgram),
     Port = bind(FD, inet, inaddr_any, 0),
     while (true)                  % infinite loop
            Message = recvfrom(FD, inet),
            sendto(FD, "Message Received", inet,
                   Message.address, Message.port)
     end.

client(Address, Port) =>
     FD = socket(inet, dgram),
     MyPort = bind(FD, inet, inaddr_any, 0),
     sendto(FD, "Did you get this message?",
            inet, Address, Port),
     Message = recvfrom(FD, inet),
     close(FD).
```

The server performs the following steps:

1. The server creates a communication endpoint, using the `socket` function.
2. The endpoint associates itself with a port number, using `bind`.
3. In an infinite loop, the following occurs:
   (a) The server uses `recvfrom` to wait for an incoming message.
   (b) The server uses `sendto` to respond to the client's message.

The client performs the following steps:

1. The client creates a communication endpoint, using the `socket` function.
2. The client endpoint associates itself with a port number, using `bind`.
3. The client uses `sendto` to send a message to the server.
4. The client uses `recvfrom` to wait for the server's response.
5. When communication is complete, the client closes the socket's file descriptor.

Note that in the TCP example, both the client and the server used the client's socket's file descriptor for `send` and `recv`, while in the UDP example, each endpoint uses its own file descriptor for `sendto` and `recvfrom`.

Communication between a client and a server over UDP uses the following functions and predicates:

- `socket(`$Domain$`, `$Type$`) = `FD: For UDP, the socket's $Type$ parameter is `dgram`.
- `udp_socket() = `FD: This performs the same operation as `socket(inet, dgram)`.
- `bind(`FD`, `$INet$`, `$Address$`, `$Port$`)`: This associates a communication endpoint with a port number, as discussed in Section 15.1. Note that for UDP, both the server and the client must call `bind`.
- `udp_bind(`FD`, `$Address$`, `$Port$`)`: This performs the same operation as `bind(`FD`, inet, `$Address$`, `$Port$`)`. It uses IPv4.
- `sendto(`FD`, `$Message$`, `$Domain$`, `$Address$`, `$Port$`) = `$NBytes$: This is used for unreliable communication. This function blocks until it is able to send the message. It can use either IPv4 or IPv6. The FD parameter is the file descriptor that the `socket` function returns. Unlike the `send` function, where the domain is known from `connect` and `accept`, the `sendto` function must specify the domain. The $Domain$ parameter can be either `inet` or `inet6`. The $Message$ parameter is a string that one endpoint is transmitting to the other endpoint. The $Address$ parameter is a string that stores the IP address of the other endpoint, as described in the explanation of `bind`. The $Address$ parameter can be the atom `localhost` in order to represent the address of the current machine. The $Port$ parameter stores the number of the port that the other endpoint is using to communicate. The `sendto` function returns the number of bytes that were sent. See Section 15.4 for `sendto` in the Unix domain.
- `sendto(`FD`, `$Message$`, `$Flags$`, `$Domain$`, `$Address$`, `$Port$`) = `$NBytes$: The $Flags$ parameter is a list of options. The options can be `oob`, `dontroute`, `dontwait`, and `nosignal`. These options were discussed in the description of `send` in Section 15.1. Windows only supports `oob` and `dontroute`.

- `recvfrom(`FD`, `$Domain$`) = `$Message$: This is used for unreliable communication. It can use either IPv4 or IPv6. The FD parameter is the file descriptor that the `socket` function returns. Unlike the `recv` function, where the domain is known from `connect` and `accept`, the `recvfrom` function must specify the domain. The $Domain$ parameter can be `inet` or `inet6`. In the Unix domain, it can also be `unix`. The `recvfrom` function blocks until data is received, and returns the message that was received. Unlike the `recv` function, the `recvfrom` function returns a map. Since the communication is connectionless, information about the other communication endpoint is not known until a message is received. Therefore, the message is stored in a map, which contains information about the sender. In the Internet domain, the `recvfrom` function returns a map with the keys `address`, `port`, and `message`. The key `address` is a string that stores the IP address of the other endpoint. The key `port` stores the number of the port that the other endpoint is using to communicate. The key `message` contains the actual message string that was received. See Section 15.4 for `recvfrom` in the Unix domain.

- `recvfrom(`FD`, `$Flags$`, `$Domain$`) = `$Message$: The $Flags$ parameter is a list of options. The options can be `oob`, `peek`, `waitall(Length)`, and `dontwait`. These options were discussed in the description of `recv` in Section 15.1. Windows only supports `oob` and `peek`.

- `close(`FD`)`

15.3 Multicasting

Multicasting allows a group of receivers to receive a single message. Multicasting is implemented using UDP. The following example shows how multicasting communication can occur.

Example

```
import socket.

sender(GroupAddress, GroupPort) =>
     FD = socket(inet, dgram),
     while (true)                    % infinite loop
            sendto(FD, "Group Message", inet,
                   GroupAddress, GroupPort)
     end.

receiver(GroupAddress, GroupPort)  =>
     Messages = new_array(100),
     FD = socket(inet, dgram),
     MyPort = bind(FD, inet, inaddr_any, 0),
     joingroup(GroupAddress),
     foreach (I in 1 .. 100) % receive 100 messages
              Messages[I] = recvfrom(FD, inet)
     end,
     leavegroup(GroupAddress),
     close(FD).
```

A sender performs the following steps:

1. In order to send a message to a group, a sender creates a UDP communication endpoint, using the `socket` function.
2. The sender uses the `sendto` function, specifying the group's IP address. This will send a single message to every member of the group.

Each receiver performs the following steps:

1. A receiver creates a UDP communication endpoint, using the `socket` function.
2. The receiver chooses a port through which to receive group messages, using the `bind` function.
3. The receiver requests to join the group, using the `joingroup` predicate.
4. The receiver uses the `recvfrom` function to listen for incoming datagrams.
5. If a receiver would like to leave the group, it uses the `leavegroup` predicate. In the above example, the receiver leaves the group after receiving 100 group messages.
6. After the receiver leaves the group, it closes the socket's file descriptor.

Multicasting uses the following functions and predicates:

- `socket(`$Domain$`, `$Type$`) = `FD
- `udp_socket() = `FD: This performs the same operation as `socket(inet, dgram)`.
- `bind(`FD`, `$INet$`, `$Address$`, `$Port$`)`: The receiver associates a communication endpoint with a port number. Note that the sender is not expecting a response, so the sender does not need to bind itself to a port.
- `udp_bind(`FD`, `$Address$`, `$Port$`)`: This performs the same operation as `bind(`FD`, inet, `$Address$`, `$Port$`)`. It uses IPv4.
- `joingroup(`$GroupAddress$`)`: This allows a receiver to request to join a group. The $GroupAddress$ parameter is a string specifying the group's IP address in dotted-decimal notation (in IPv4) or hexadecimal notation (in IPv6). The `joingroup` predicate performs the same operation as `setsockopt(`FD`, ip, addmembership, `$GroupAddress$`)`. For a further discussion of `setsockopt`, see Section 15.5.1.
- `leavegroup(`$GroupAddress$`)`: This allows a receiver to leave a group. The $GroupAddress$ parameter is a string specifying the group's IP address. The `leavegroup` predicate performs the same operation as `setsockopt(`FD`, ip, dropmembership, `$GroupAddress$`)`. For a further discussion of `setsockopt`, see Section 15.5.1.
- `sendto(`FD`, `$Message$`, `$Domain$`, `$Address$`, `$Port$`) = `$NBytes$: This is used for unreliable communication. This function blocks until it is able to send the message. The `sendto` function returns the number of bytes that were sent.
- `sendto(`FD`, `$Message$`, `$Flags$`, `$Domain$`, `$Address$`, `$Port$`) = `$NBytes$: The $Flags$ parameter is a list of options. The options can be `oob`, `dontroute`, `dontwait`, and `nosignal`.
- `recvfrom(`FD`, `$Domain$`) = `$Message$: This is used for unreliable communication. The `recvfrom` function blocks until data is received, and returns the a map that contains the message that was received.

- `recvfrom(`FD`, `$Flags$`, `$Domain$`) = `$Message$: The $Flags$ parameter is a list of options. The options can be `oob`, `peek`, `waitall(Length)`, and `dontwait`.
- `close(`FD`)`

15.4 Communication on the Unix Domain

The Unix domain is used for inter-process communication on the Unix operating system. The processes are located on the same host. The following is an example of communication in the Unix domain.

Example

```
import socket.

server(Name) =>
     FD = socket(unix, stream),
     bind(FD, unix, Name),
     listen(FD), % wait for connection
     Client = accept(FD),
     hello_to_client(Client),
     close(FD).

client(Name) =>
     FD = socket(unix, stream),
     connect(FD, unix, Name),
     hello_to_server(FD),
     close(FD).

hello_to_client(Client) => % Server code to communicate with client
     % Begin communication
     send(Client, "Enter Input:"),
     Str = recv(Client), % wait for client input
     send(Client, Str),
     % End communication
     close(Client).

hello_to_server(FD) => % Client code to communicate with server
     % Begin communication
     Str = recv(FD), % wait for server's message
     println(Str)
     send(FD, "Hello"),
     Str := recv(FD),
     println(Str).
     % End communication
```

The server and the client perform the same steps as they do in the TCP example. They can also communicate over UDP. In this example, the server only listens for one request, so it does not fork a child process, and when the server completes, it closes its file descriptor. This example is similar to the first example of the chapter.

Communication between a server and a client in the Unix domain uses the following functions and predicates:

- `socket(`$Domain$`, `$Type$`) = `FD: The file descriptor FD represents a communication endpoint, which is a process in the Unix domain.
- `unix_socket() = `FD: This performs the same operation as `socket(unix, stream)`.
- `bind(`FD`, `$Unix$`, `$Name$`)`: Since the processes are located on the same host, the sockets do not need to be identified by addresses and ports in the `bind` predicate. Instead, Unix domain sockets are identified by a file name in the file system. Therefore, the process is bound to a file name using `bind(`FD`, unix, `$Name$`)`. The `unix` atom is the only value that is allowed for the $Unix$ parameter.
- `unix_bind(`FD`, `$Name$`)`: This performs the same operation as `bind(`FD`, unix, `$Name$`)`.
- `listen(`FD`, `$Backlog$`)`
- `listen(`FD`)`
- `accept(`FD`) = `$Client$
- `connect(`FD`, `$Unix$`, `$Name$`)`: Since the processes are located on the same host, the sockets do not need to be identified by addresses and ports in the `connect` predicate. Instead, Unix domain sockets are identified by a file name in the file system. Therefore, the client process connects to the server process by using the server process's file name in the `connect(`FD`, `$Unix$`, `$Name$`)` predicate. The `unix` atom is the only value that is allowed for the $Unix$ parameter.
- `unix_connect(`FD`, `$Name$`)`: This performs the same operation as `connect(FD, unix, Name)`.
- `send(`FD`, `$Message$`) = `$NBytes$: This is used for reliable communication. This function blocks until it is able to send the message. The `send` function returns the number of bytes that were sent.
- `send(`FD`, `$Message$`, `$Flags$`) = `$NBytes$: The $Flags$ parameter is a list of options. The options can be `oob`, `dontroute`, `dontwait`, and `nosignal`.
- `recv(`FD`) = `$Message$: This is used for reliable communication. It blocks until data arrives. The `recv` function returns a message string.
- `recv(`FD`, `$Flags$`) = `$Message$: The $Flags$ parameter is a list of options. The options can be `oob`, `peek`, `waitall(Length)`, and `dontwait`.
- `sendto(`FD`, `$Message$`, `$Name$`) = `$NBytes$: This is used for unreliable communication. This function blocks until it is able to send the message. Note that `sendto` in the Unix domain uses the name of a file. The `sendto` function returns the number of bytes that were sent.
- `sendto(`FD`, `$Message$`, `$Flags$`, `$Name$`) = `$NBytes$: The $Flags$ parameter is a list of options. The options can be `oob`, `dontroute`, `dontwait`, and `nosignal`.

- `recvfrom(`FD`, `$Domain$`) = `$Message$: This is used for unreliable communication. In the Unix domain, the `recvfrom` function returns a map with the keys `name`, and `message`. The key `name` is a string that stores the file name to which the process is bound. The key `message` contains the actual message string that was received.
- `recvfrom(`FD`, `$Flags$`, `$Domain$`) = `$Message$: The $Flags$ parameter is a list of options. The options can be `oob`, `peek`, `waitall(Length)`, and `dontwait`.
- `close(`FD`)`

15.5 Other Socket Functions and Predicates

15.5.1 Socket Options

Picat allows users to access and to modify socket options. For example, the Multicasting section showed how to modify a socket's options to allow it to join or to leave a group. The following example shows how to read a few of a socket's default attributes.

Example

```
import socket.

show_attributes =>
     println("TCP/IP socket"),
     attributes(inet, stream),
     println("UDP/IP socket"),
     attributes(inet, dgram),
     println("TCP/IPv6 socket"),
     attributes(inet6, stream).

attributes(Domain, Type) =>
     FD = socket(Domain, Type),
     Live = getsockopt(FD, socket, keepalive),
     println(Val),
     OOB = getsockopt(FD, socket, oobinline),
     println(OOB),
     Delay = getsockopt(tcp, nodelay),
     println(Delay),
     Multicast = getsockopt(ipv6, multicasthops),
     println(Multicast),
     close(FD).
```

The following functions are used to access and to modify the socket options:

- `setsockopt(`FD`, `$Level$`, `$Option$`, `$Value$`)`
- `getsockopt(`FD`, `$Level$`, `$Option$`) = `$Value$

The $Level$ parameter is an atom indicating the protocol level at which the $Option$ parameter is defined. The $Level$ parameter can be one of the following atoms: `socket`, `tcp`, `ipx`, `ip`, or `ipv6`. The $Option$ parameter is also an atom. For a list of available options for each level, see Appendix F. Note that some options can only be used in `getsockopt`, and that the `getsockopt` function returns the string "Non-existent" if it does not recognize the $Option$ parameter.

15.5.2 Host Information

Picat provides other socket functions that allow users to extract information about Internet hosts. The following example shows how to connect to `www.probp.com` by using `gethostbyname`.

Example

```
import socket.

probp(Port) =>
     FD = socket(inet, stream),
     Server = gethostbyname("http://www.probp.com")
     if length(Server.addresses) > 0 then
        Address = Server.addresses[1],
        connect(FD, inet, Address, Port),
        hello_to_server(FD) % Communicate with the server
     end,
     close(FD).
```

The following functions are used to extract Internet host information:

- `gethostbyname(`$Name$`) = `$Host$
- `gethostbyaddr(`$Addr$`) = `$Host$

These functions perform a DNS query. They return a map with the keys `names` and `hosts`. The key `names` is a list of names for the host, and the key `addresses` is a list of IP addresses by which the host is accessible. If the query does not have any results, then the lists will be empty.

In the `gethostbyname` and `gethostbyaddr` functions, the parameter can be replaced by the atom `localhost`, in order to extract information about the current machine.

Picat provides two other functions for querying the canonical name and a single address for a host. The following example shows how to connect to `www.probp.com` by using `getaddr`.

Example

```
import socket.

probp(Port) =>
     FD = socket(inet, stream),
     Address = getaddr("http://www.probp.com"),
     connect(FD, inet, Address, Port),
     hello_to_server(FD), % Communicate with the server
     close(FD).
```

The following functions are used to extract a single name or address for a host:

- `getaddr(`$Name$`) = `$Addr$
- `getcanonicalname(`$Addr$`) = `$Name$

In the `getaddr` and `getcanonicalname` functions, the parameter can be replaced by the atom `localhost`, in order to extract information about the current machine. Unlike the `gethostbyname` and `gethostbyaddr` functions, if the query does not return any results, then the `getaddr` and `getcanonicalname` functions will throw an error.

15.5.3 Services

Some services, like Telnet and FTP, have pre-defined port numbers. The following functions allow users to look up information about a service, such as the port number that it uses:

- `getservbyname(`$Name$`) = `$Service$: This function matches a service for any protocol. It returns a map, with the keys `name`, `aliases`, `port`, and `protocol`. The key `name` is the name of the service. The key `aliases` is a list of alternative names for the service. The key `port` is the port number to use, in order to access the service. The key `protocol` is a string, indicating the protocol to use in order to access the service.
- `getservbyname(`$Name$`, `$Type$`) = `$Service$: This function matches a service for a specific protocol. The $Type$ parameter is either the atom `tcp` or the atom `udp`.
- `getservport(`$Name$`) = `$Port$: This function returns the port number to use in order to access the service.

These functions throw an error if the service cannot be found.

Chapter 16

External Language Interface with C

Picat has a bi-directional interface with C, through which Picat programs can call functions written in C, and C programs can query Picat programs. C programs that use this interface must include the file `"picat.h"` in the directory `$PICATDIR/Emulator`.

16.1 Calling C from Picat

16.1.1 Term Representation

A term is represented by a word that contains a value and a tag. The tag distinguishes the type of the term. Floating-point numbers are represented as special structures in the form of `$float(I1,I2,I3)$`, where `I1`, `I2`, and `I3` are integers.

The value of a term is an address, except when the term is an integer (in this case, the value represents the integer itself). The location to which the address points is dependent on the type of the term. The address in a reference points to the referenced term. An unbound variable is represented by a self-referencing pointer. The address in an atom points to the record for the atom symbol in the symbol table. The address in a structure, $f(t_1, \ldots, t_n)$, 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. The address in a list, *[H|T]*, points to a block of two consecutive words, where the first word stores the car, *H*, and the second word stores the cdr, *T*.

16.1.2 Fetching Arguments of Picat Calls

C functions that define a Picat predicate should not take any argument. The function `picat_get_call_arg(i,arity)` is used to get the 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.

16.1.3 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_atom(TERM t)`: Term `t` is an atom.
- `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_nil(TERM t)`: Term `t` is nil.
- `int picat_is_list(TERM t)`: Term `t` is a list.
- `int picat_is_structure(TERM t)`: Term `t` is a structure (but not a list).
- `int picat_is_compound(TERM t)`: True if either `picat_is_list(t)` or `picat_is_structure(t)` is true.
- `int picat_is_unifiable(TERM t1, TERM t2)`: `t1` and `t2` are unifiable. This is equivalent to the Picat call `not(not(t1=t2))`.
- `int picat_is_identical(TERM t1, TERM t2)`: `t1` and `t2` are identical. This function is equivalent to the Picat call `t1==t2`.

16.1.4 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` is set. A C program that uses these functions must check whether `exception` is set in order to see whether data are converted correctly. The converted data are only correct when `exception` is `NULL`.

- `int picat_get_integer(TERM t)`: Convert the Picat integer `t` into C. `picat_is_integer(t)` must be true; otherwise 0 is returned before `exception` is set to `integer_expected`.
- `double picat_get_float(TERM t)`: Convert the Picat float `t` into C. `picat_is_float(t)` must be true; otherwise `exception` is set to `number_expected`, and 0.0 is returned. This function must be declared before any use.
- `(char *) picat_get_name(TERM t)`: Return a pointer to the string that is the name of term `t`. Either `picat_is_atom(t)` or `picat_is_structure(t)` must be true; otherwise, `exception` is set to `illegal_arguments`, and `NULL` is returned. This function must be declared before any use.
- `int picat_get_arity(TERM t)`: Return the arity of term `t`. Either `picat_is_atom(t)` or `picat_is_structure(t)` must be true; otherwise, 0 is returned, with `exception` being set to `illegal_arguments`.

16.1.5 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, or `PICAT_FALSE` if the unification fails.
- `TERM picat_get_arg(int i,TERM t)`: Return the `i`th argument of term `t`. The condition `picat_is_compound(t)` must be true, and `i` must be an integer that is between 1 and `t`'s arity; otherwise, `exception` is set to `illegal_arguments`, and the Picat integer `0` is returned.
- `TERM picat_get_car(TERM t)`: Return the car of the list `t`. `picat_is_list(t)` must be true; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.

- `TERM get_cdr(TERM t)`: Return the cdr of the list `t`. `picat_is_list(t)` must be true; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.
- `void picat_write(TERM t)`: Send term `t` to the current output stream.

16.1.6 Building Picat Terms

- `TERM picat_build_var()`: Return a free Picat variable.
- `TERM picat_build_integer(int 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 functor is `name`, arity is `arity`, and the arguments are all free variables.

16.1.7 Registering Predicates that were Defined in C

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 cannot take any argument. 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 built-ins of Picat.

Example

Consider the Picat predicate:

```
p(a,X) => X=$f(1)$.
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"

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(1) */
  picat_unify(picat_get_arg(1, f1), picat_build_integer(1));
  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 clauses */
  if (!picat_is_atom(a1))
      return PICAT_FALSE;
  name_ptr = picat_get_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 Recompile the system. Now, `p/2` is in the group of built-ins in Picat.

16.2 Calling Picat from C

In order to make Picat predicates callable from C, one must replace the `main.c` file in the emulator with a new file that starts his/her own application. The following function must be executed before any call to Picat predicates is executed:

```
initialize_bprolog(int argc, char *argv[])
```

In addition, the environment variable `BPDIR` must be correctly set to the home directory where Picat was installed. The function `initialize_bprolog()` allocates all of the stacks that

are used in Picat, initializes them, and loads the byte code file `bp.out` into the program area. `PICAT_ERROR` is returned if the system cannot be initialized.

A query can be a string or a Picat term, and a query can return one or more solutions.

- `int picat_call_string(char *goal)`: This function executes the Picat call, as represented by the string `goal`. The return value is `PICAT_TRUE` if the call succeeds, `PICAT_FALSE` if the call fails, and `PICAT_ERROR` if an exception occurs. Examples:

```
picat_call_string("load(myprog)")
picat_call_string("X is 1+1")
picat_call_string("p(X,Y), q(Y,Z)")
```

- `picat_call_term(TERM goal)`: This function is similar to `picat_call_string`, except that it executes the Picat call, as represented by the term `goal`. While `picat_call_string` cannot return any bindings for variables, this function can return results through the Picat variables in `goal`. Example:

```
TERM call = picat_build_structure("p", 2);
picat_call_term(call);
```

- `picat_mount_query_string(char *goal)`: Mount `goal` as the next Picat goal to be executed.
- `picat_mount_query_string(TERM goal)`: Mount `goal` as the next Picat goal to be executed.
- `picat_next_solution()`: Retrieve the next solution of the current goal. If no goal is mounted before this function, then the exception `illegal_predicate` will be raised, and `PICAT_ERROR` will be returned as the result. If no further solution is available, then the function returns `PICAT_FALSE`. Otherwise, the next solution is found.

Example

This example program retrieves all of the solutions of the query `member(X,[1,2,3])`.

```
#include "bprolog.h"

main(int argc, char *argv[])
{
  TERM query;
  TERM list0, list;
  int res;

  initialize_bprolog(argc, argv);
  /* build the list [1,2,3] */
  list = list0 = picat_build_list();
  picat_unify(picat_get_car(list), picat_build_integer(1));
  picat_unify(picat_get_cdr(list), picat_build_list());
  list = picat_get_cdr(list);
  picat_unify(picat_get_car(list), picat_build_integer(2));
```

```
  picat_unify(picat_get_cdr(list), picat_build_list());
  list = picat_get_cdr(list);
  picat_unify(picat_get_car(list), picat_build_integer(3));
  picat_unify(picat_get_cdr(list), picat_build_nil());

  /* build the call member(X,list) */
  query = picat_build_structure("member", 2);
  picat_unify(picat_get_arg(2, query), list0);

  /* invoke member/2 */
  picat_mount_query_term(query);
  res = picat_next_solution();
  while (res == PICAT_TRUE) {
    picat_write(query);
    printf("\n");
    res = picat_next_solution();
  }
}
```

In order to run the program, users need to replace the content of the file `main.c` in `$BPDIR/Emulator` with this program, and then recompile the system. The newly compiled system will give the following output when it is started.

```
member(1,[1,2,3])
member(2,[1,2,3])
member(3,[1,2,3])
```

Appendix A

Appendix: Math

Picat provides a `math` module, which has common mathematical constants and functions. In order to use the examples in this chapter, first type `import math` on the command line.

A.1 Constants

The `math` module provides four constants.

- `e = 2.71828`
- `pi = 3.14159`
- `inf`: This represents positive infinity.
- `ninf`: This represents negative infinity.

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 determines whether X is positive or negative. If X is positive, then this function returns 1. If X is negative, then this function returns -1. If X is 0, then this function returns 0.
- `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`) = (`$FractVal$`, `$IntVal$`)`: This function splits a real number into its fractional part and its integer 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),
       Val3 = round(-3.5), Val4 = round(3.5)
Val1 = -3
Val2 = 3
Val3 = -4
Val4 = 4
picat> Val1 = truncate(-3.2), Val2 = truncate(3)
Val1 = -3
Val2 = 3
picat> (F1, I1) = modf(3.2), (F2, I2) = modf(3)
F1 = 2
I1 = 3
F2 = 0
I2 = 3
```

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 $X * *Y$.
- `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 $X < 0$, then this function throws an error.

- `cbrt(`X`) = `Val: This function returns the cube root of X.
- `nthrt(`N`, `X`) = `Val: This function returns the Nth root of X. Note that, if N is even, and $X < 0$, then 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), C = cbrt(27), N = nthrt(5, 32)
S = 1.0
C = 3.0
N = 2.0
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.

- `radians(`$Degree$`) = `$Radian$: This function converts from degrees to radians.
- `degrees(`$Radian$`) = `$Degree$: This function converts from radians to degrees.

Example

```
picat> R = radians(180)
R = 3.14159
picat> D = 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. If the tangent is undefined, such as at `pi / 2`, then this function throws an error.

- `sec(`X`)` = Val: This function returns the secant of X, where X is given in radians. If $cos(X)$ is 0, then $sec(X)$ 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 $csc(X)$ throws an error.
- `cot(`X`)` = Val: This function returns the cotangent of X, where X is given in radians. If $tan(X)$ is 0, or if $tan(X)$ is undefined, then $cot(X)$ 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 cosineof 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(`X`,` Y`)` = 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`]. Note that, if both X and Y are 0, then this function throws an error.

Example

```
picat> S = sin(pi), C = cos(pi), T = tan(pi)
S = 0.0
C = -1.0
T = 0.0
picat> S = sec(pi / 4), C = csc(pi / 4), T = cot(pi / 4)
S = 1.41421
C = 1.41421
T = 1.0
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
```

A.2.6 Hyperbolic Functions

The `math` module provides the following hyperbolic functions.

- `sinh(`X`)` = Val: This function returns the hyperbolic sine of X, where X is given in radians.
- `cosh(`X`)` = Val: This function returns the hyperbolic cosine of X, where X is given in radians.
- `tanh(`X`)` = Val: This function returns the hyperbolic tangent of X, where X is given in radians.

- `sech(`X`) = `Val: This function returns the hyperbolic secant of X, where X is given in radians. If $cosh(X)$ is 0, then $sech(X)$ throws an error.
- `csch(`X`) = `Val: This function returns the hyperbolic cosecant of X, where X is given in radians.
- `coth(`X`) = `Val: This function returns the hyperbolic cotangent of X, where X is given in radians. If $tanh(X)$ is 0, then $coth(X)$ throws an error.
- `asinh(`X`) = `Val: This function returns the arc hyperbolic sine of X, in radians.
- `acosh(`X`) = `Val: This function returns the arc hyperbolic cosine of X, in radians. If $X \leq 1$, then this function throws an error.
- `atanh(`X`) = `Val: This function returns the arc hyperbolic tangent of X, in radians. X must be in the range $(-1, 1)$; 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 number, in the range [0, 1).
- `random(`$Seed$`) = `Val: This function returns a random number, in the range [0, 1). At the same time, it changes the seed of the random number generator.
- `randrange(`$From$`, `To`) = `Val: This function returns a random integer in the range $[From, To)$.
- `randrange(`$From$`, `$Step$`, `To`) = `Val: This function returns a random integer in the range $[From, To)$. The integer will be equal to $From + K * Step$, for some integer K.

Appendix B

Appendix: Date and Time

Picat's `datetime` module provides built-ins for manipulating and retrieving the date and time.

B.1 Representing Date and Time

Picat represents the date and time as a structure of integers. The structure has the form `$date_time(`$Year$`,` $Month$`,` Day`,` $Hour$`,` $Minute$`,` $Second$`,` $MilliSecond$`)$`.

Variable	Range of Values
$Year$	Four-digit year
$Month$	1-12
Day	1-31
$Hour$	0-23
$Minute$	0-59
$Second$	0-60
$MilliSecond$	0-999

In the $Month$ variable, 1 represents January, and 12 represents December. In the $Hour$ variable, 0 represents 12 AM, and 23 represents 11 PM. In the $Second$ variable, the value 60 represents a leap second.

The following function creates a new datetime object, which is initialized to the date and time at which the object is created.

- `current_datetime() =` $DateTime$

The following example creates a `date_time` structure at 10:56:23.899 PM on October 24, 2014.

Example

```
picat> import datetime.

picat> D = current_datetime().
D = date_time(2014, 10, 24, 22, 56, 23, 899)
```

B.2 Extracting Values

The following functions extract a single field from the `date_time` structure.

- `year(`$DateTime$`) = `$Year$
- `month(`$DateTime$`) = `$Month$
- `day(`$DateTime$`) = `Day
- `hour(`$DateTime$`) = `$Hour$
- `minute(`$DateTime$`) = `$Minute$
- `second(`$DateTime$`) = `$Second$
- `millisecond(`$DateTime$`) = `$MilliSecond$

Each of these functions returns an integer.

B.3 Changing the Date and Time

There are two ways to change the date and time: *adding*, and *setting*.

B.3.1 Adding

The following functions add values to a `date_time` in order to create a new `date_time`. There is one function to add (or subtract) values to each variable in the `date_time` structure.

- `add_years(`$DateTime$`, `$Years$`) = `$DateTime$
- `add_months(`$DateTime$`, `$Months$`) = `$DateTime$
- `add_days(`$DateTime$`, `$Days$`) = `$DateTime$
- `add_hours(`$DateTime$`, `$Hours$`) = `$DateTime$
- `add_minutes(`$DateTime$`, `$Minutes$`) = `$DateTime$
- `add_seconds(`$DateTime$`, `$Seconds$`) = `$DateTime$
- `add_milliseconds(`$DateTime$`, `$MilliSeconds$`) = `$DateTime$

The following example creates a `date_time` structure at 10:56:23.899 PM on October 24, 2014, and then adds values to the structure.

```
picat> import datetime.

picat> D = current_datetime(), D1 = add_seconds(D, 5),
          D2 = add_days(D, -2), D3 = add_minutes(D, 20).
D = date_time(2014, 10, 24, 22, 56, 23, 899)
D1 = date_time(2014, 10, 24, 22, 56, 28, 899)
D2 = date_time(2014, 10, 22, 22, 56, 23, 899)
D3 = date_time(2014, 10, 24, 23, 16, 23, 899)
```

$D1$ adds 5 seconds to the time. $D2$ subtracts two days from the date; this is done by passing a negative number to `add_days`. $D3$ adds 20 minutes to the time; note that this also makes the hour change.

B.3.2 Setting

The following predicates modify a `date_time` structure, allowing users to set each variable to a specified value.

- `set_year(`$DateTime$`, `$Year$`)`
- `set_month(`$DateTime$`, `$Month$`)`
- `set_day(`$DateTime$`, `Day`)`: If Day is larger than the number of days in the month, then an error is thrown.
- `set_hour(`$DateTime$`, `$Hour$`)`
- `set_minute(`$DateTime$`, `$Minute$`)`
- `set_second(`$DateTime$`, `$Second$`)`: If $Second$ is set to 61, but $Second$ is not a valid leap second, then an error is thrown.
- `set_millisecond(`$DateTime$`, `$MilliSecond$`)`

Note that the second argument to each predicate must be within the valid integer range for the specified variable. Otherwise, an error is thrown.

The following example creates a `date_time` structure at 10:56:23.899 PM on October 24, 2014, and then sets values in the structure.

Example

```
picat> import datetime.

picat> D = current_datetime(), set_second(D, 5),
          set_day(D, 2), set_minute(D, 20).
D = date_time(2014, 10, 2, 22, 20, 5, 899)
```

B.4 Converting to Strings

The following functions convert sections of the `date_time` structure to a string.

- `day_string(`$DateTime$`) = `$String$: This function returns the day of the week (Sunday, Monday, ...) for the specified date, in the system's locale. This is the same as `dt_to_fstring("%A", `$DateTime$`)`.
- `month_string(`$DateTime$`) = `$String$: This function returns the month of the year (January, February, ...) for the specified date, in the system's locale. This is the same as `dt_to_fstring("%A", `$DateTime$`)`.
- `time_string(`$DateTime$`) = `$String$: This function returns the 12-hour time in the format `HH:MM AM/PM`.
- `dt_to_fstring(`$Format$`, `$DateTime$`) = `$String$: This function converts the `date_time` structure to a string, depending on the $Format$ parameter, which is a string that contains format characters in the form `%specifier`. For details, see Appendix E.

The following examples create a `date_time` structure at 10:56:23.899 PM on October 24, 2014, and convert the structure to strings.

Example

```
picat> import datetime.

picat> D = current_datetime(), Day = day_string(D),
           Month = month_string(D), Time = time_string(D).
D = date_time(2014, 10, 2, 22, 20, 5, 899)
Day = "Friday"
Month = "October"
Time = "10:56 PM"

picat > D = current_datetime(), DS = dt_to_fstring("%B %d, %Y", D).
D = date_time(2014, 10, 2, 22, 20, 5, 899)
DS = "October 24, 2014"

picat > D = current_datetime(), DS = dt_to_fstring("%d %B %Y", D).
D = date_time(2014, 10, 2, 22, 20, 5, 899)
DS = "24 October 2014"
```

B.5 Other Built-ins

- `compare(`$DateTime1$`, `$DateTime2$`) = `$Result$: This function compares two `date_time` structures. If the first date_time is earlier than the second date_time, then this function returns -1. If the first date_time is later than the second date_time, then this function returns 1. Otherwise, the two date_times are equal, and this function returns 0.
- `is_leap_year(`$DateTime$`)`: This predicate determines whether the $Year$ of the `date_time` structure is a leap year.

Appendix C

Appendix: Lexical Grammar

```
/* Picat lexical grammar 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 ->
    unicode_input_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_token
    separator_token

atom_token ->
    small_letter {alphanumeric_char}
    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 ->
    underscore_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 ->
    "0"
    non_zero_digit [decimal_digits]
    non_zero_digit underscores decimal_digits

decimal_digits ->
    decimal_digit
    decimal_digit [decimal_digits_and_underscores] decimal_digit

non_zero_digit ->
    one of "1" "2" "3" "4" "5" "6" "7" "8" "9"

decimal_digits_and_underscores ->
    decimal_digit_or_underscore
    decimal_digits_and_underscores decimal_digit_or_underscore

decimal_digit_or_underscore ->
    decimal_digit
    "_"

underscores ->
    "_"
    underscores "_"
```

```
hex_numeral ->
    "0x" hex_digits
    "0X" hex_digits

hex_digits ->
    hex_digit
    hex_digit [hex_digits_and_underscores] hex_digit

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" underscores octal_digits

octal_digits ->
    octal_digit
    octal_digit [octal_digits_and_underscores] octal_digit

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_digit [binary_digits_and_underscores] binary_digit

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_digits "." [decimal_digits] [exponent_part]
    "." decimal_digits [exponent_part]
    decimal_digits exponent_part

exponent_part ->
    exponent_indicator signed_integer

exponent_indicator ->
    "e"
    "E"

signed_integer ->
    [sign] decimal_digits

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"

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 ->
    "\b"    /* \u0008: backspace BS */
    "\t"    /* \u0009: horizontal tab HT */
    "\n"    /* \u000a: linefeed LF */
    "\f"    /* \u000c: form feed FF */
    "\r"    /* \u000d: carriage return CR */
    "\""    /* \u0022: double quote " */
    "\'"    /* \u0027: single quote ' */
    "\\"    /* \u005c: backslash \ */
    octal_escape
    unicode_escape

octal_escape ->
    "\" octal_digit octal_digit octal_digit

unicode_escape ->
    "\u" hex_digit hex_digit hex_digit hex_digit
    "\U" hex_digit hex_digit hex_digit hex_digit
         hex_digit hex_digit hex_digit hex_digit
```

Appendix D

Appendix: Syntax Grammar

```
/* Picat syntax grammar rules
      [...] means optional
      {...} means 0, 1, or more occurrences
      (a | b) means choice
      "..." means a token
      %... one-line comment
      input tokens:
           atom
    variable
    integer
    float
            operator
            separator
            eor is "." followed by a white space
*/
program ->
    [module_declaration]
    {import_declaration}
    program_body

program_body ->
    {include_declaration | predicate_definition
     | function_definition | actor_definition}

module_declaration ->
    "module" atom eor

import_declaration ->
    import import_item {"," import_item} eor

import_item ->
    atom ["." atom ["/" integer]]

include_declaration ->
    "include " string {, string} eor
```

```
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_mode {"," index_mode} ")"

index_mode ->
   "+"
   "-"

table_mode ->
   "+"
   "-"
   "min"
   "max"

cardinality_limit ->
   "cardinality" "(" integer ")"

predicate_rule_or_fact ->
    predicate_rule
    predicate_fact

function_rule_or_fact ->
    function_rule
    function_fact

predicate_rule ->
    head ["," condition] ("=>" | "?=>") body eor

nonbacktrackable_predicate_rule ->
    head ["," condition] "=>" body eor

predicate_fact ->
    head eor
```

```
head ->
    atom ["(" [term {"," term}] ")"]

function_rule ->
    head "=" variable ["," condition] "=>" body eor

function_fact ->
    head "=" argument eor

action_rule ->
    head ["," condition] "," "{" event_pattern "}"   => body eor

event_pattern ->
    term {',' term}

condition -> goal

body -> goal

goal ->
    disjunctive_goal

argument ->
    negative_goal

disjunctive_goal ->
    disjunctive_goal ";" conjunctive_goal
    conjunctive_goal

conjunctive_goal ->
    conjunctive_goal ","  negative_goal
    negative_goal

negative_goal ->
   "not" 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" goal "then" goal {"elseif" goal "then" goal} "else" goal "end"
   "foreach" "(" iterator {"," (iterator | condition)} ")" goal "end"
   "while" "(" goal ")" ["loop"] goal "end"
   "loop" goal "while" "(" goal ")"
   "try" goal catch_clause {catch_clause} ["finally" goal] "end"
    expression {bin_rel_op expression}

catch_clause ->
    "catch" "(" exception_pattern ")" goal

exception_pattern ->
    term

bin_rel_op ->
    "="
    "!="
    ":="
    "=="
    "!=="
    ">"
    ">="
    "<"
    "=<"
    "<="
    "in"
    "#="
    "#!="
    "#>"
    "#>="
    "#<"
    "#=<"
    "#<="

expression ->
    range_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   % bit-wise xor

and_expression ->
    shift_expression
    and_expression  "/\" shift_expression

shift_expression ->
    additive_expression
    shift_expr ( "<<" | ">>" | ">>>" ) additive_expression

additive_expression ->
    multiplicative_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    % bit-wise complement

power_expression ->
    primary_expression ["**" unary_expression]

primary_expression ->
    "(" goal ")"
    variable "[" argument ["," argument] "]"
```

```
                % index notation
    variable "@" term "@"
                % as-pattern, can only occur in terms
    variable
    integer
    float
    atom_or_call
    list_expression
    array_expression
    function_call
    lambda_term
    term_constructor
    primary_expression "." atom_or_call
                % dot-notation, primary can't be lambda

atom_or_call ->
    atom ["(" [argument {"," argument}] ")"]

list_expression ->
    "[" argument list_expression_suffix "]"

list_expression_suffix ->
    ":" iterator {"," (iterator | condition)}     % list comprehension
    {"," argument} ["|" argument]

array_expression ->
    "{" argument {"," argument} "}"

function_call ->
    [primary_expression "."] atom "(" [argument {"," argument}] ")"

lambda_term ->
    "lambda" "(" variable_list, argument ")"

variable_list ->
    "[" [variable {"," variable}] "]"

term_constructor ->
    "$" goal "$"
/* a term has the same form as a goal except that
   (1) a term contains no index notations;
   (2) a term contains no dot notations (O.E);
   (3) a term contains no lambda terms;
   (4) a term contains no loops;
   (5) a term contains no range_expressions;
   (6) a term contains no list comprehensions;
   (7) a term can contain an as-pattern in the form Var@Term@.
*/
```

Appendix E

Appendix: Formats

E.1 Formatted Printing

The following table shows the specifiers that can be used in formats for the `writef`, `fwritef`, `printf`, and `fprintf` predicates.

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

E.2 Formatted Date and Time

The following table shows that specifiers that can be used in formats for the `dt_to_fstring` function in the `datetime` module.

Specifier	Output
`%%`	Percent Sign
`%a`	Locale-dependent Weekday Name, Abbreviated
`%A`	Locale-dependent Weekday Name
`%b`	Locale-dependent Month Name, Abbreviated
`%B`	Locale-dependent Month Name
`%c`	Locale-dependent Date and Time
`%d`	Two-digit Day of the Month
`%H`	Two-digit Hour (24-hour Time)
`%I`	Two-digit Hour (12-hour Time)
`%j`	Three-digit Day of the Year
`%m`	Two-digit Month
`%M`	Two-digit Minute
`%p`	`AM` or `PM`
`%S`	Two-digit Second
`%w`	One-digit Day of the Week
`%x`	Locale-dependent Date
`%X`	Locale-dependent Time
`%y`	Two-digit Year
`%Y`	Four-digit Year

Appendix F

Appendix: Socket Options

This appendix contains the list of options for `setsockopt` and `getsockopt`, sorted by level. Note that some options can only be used in `getsockopt`.

Socket Level

- `acceptconn`
- `bindtodevice`
- `broadcast`
- `bspstate`
- `conditionalaccept`
- `connecttime`
- `debug`
- `domain`
- `dontlinger`
- `dontroute`
- `error`
- `exclusiveaddruse`
- `groupid`
- `grouppriority`
- `keepalive`
- `linger`
- `maxmsgsize`
- `oobinline`
- `portscalability`
- `protocolinfo`
- `prototype`
- `pvdconfig`
- `rcvbuf`
- `rcvlowat`
- `rcvtimeo`
- `reuseaddr`
- `sndbuf`
- `sndlowat`
- `sndtimeo`
- `type`
- `updateacceptcontext`

TCP Level

- `nodelay`

IPX Level

- address
- addressnotify
- dstype
- extendedaddress
- filterptype
- getnetinfo
- genetinfonorip
- immediatespxack
- maxadapternum
- maxsize
- ptype
- receivebroadcast
- recvhdr
- reripnetnumber
- spxgetconnectionstatus
- stopfilterptype

IP Level

- addmembership
- dropmembership

IPv6 Level

- multicasthops
- multicastif
- multicastloop
- unicasthops

Appendix G

Appendix: The Library Modules

Module `basic` (imported by default)

- 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
- X `>>>` Y `=` Z
- Var`[`$Index_1$`,...,`$Index_n$`]`
- $Goal_1$`,`$Goal_2$
- $Goal_1$`;`$Goal_2$
- `acyclic_term(`$Term$`)`
- `append(`X`,`Y`,`Z`)` (nondet)
- `apply(`S`,`Arg_1`,...,`Arg_n`)` `=` Val
- `arity(`$Term$`)`
- `array(`$Term$`)`
- `atom(`$Term$`)`
- `atom_chars(`Atm`)` `=` $String$
- `atom_codes(`Atm`)` `=` $List$
- `atomic(`$Term$`)`
- `attr_var(`$Term$`)`
- `avg(`$List$`)` `=` Val
- `between(`$From$`,`To`,`X`)` (nondet)
- `call(`S`,`Arg_1`,...,`Arg_n`)`
- `char(`$Term$`)`
- `char_code(`$Char$`)` `=` Int
- `code_char(`$Code$`)` `=` $Char$
- `compare_terms(`$Term_1$`,`$Term_2$`)` `=` Res
- `compound(`$Term$`)`
- `copy_term(`$Term_1$`)` `=` $Term_2$
- `delete(`$List$`,`X`)` `=` $ResList$
- `delete_all(`$List$`,`X`)` `=` $ResList$
- `different_terms(`$Term_1$`,`$Term_2$`)`
- `digit(`$Char$`)`
- `fail`
- `findall(`$Template$`,`S`,`Arg_1`,...,`Arg_n`)` `=` $List$
- `float(`$Term$`)`
- `flush`
- `freeze(`X`,`$Goal$`)`
- `get(`$MapOrAttrVar$`,`Key`)` `=` Val
- `get_global_map()` `=` Map
- `get_heap_map()` `=` Map
- `ground(`$Term$`)`
- `has_key(`$MapOrAttrVar$`,`Key`)`
- `hash_code(`$Term$`)` `=` Int
- `insert(`$List$`,`$Index$`,`Elm`)` `=` $ResList$
- `insert_all(`$List$`,`$Index$`,`$AList$`)` `=` $ResList$
- `integer(`$Term$`)`

- keys(*MapOrAttrVar*) = *List*
- length(*Compound*) = *Len*
- list(*Term*)
- lowercase(*Char*)
- map(*Term*)
- map_to_list(*Map*) = *List*
- max(*List*) = *Val*
- max(*X*,*Y*) = *Val*
- membchk(*Term*,*List*)
- member(*Term*,*List*) (nondet)
- min(*List*) = *Val*
- min(*X*,*Y*) = *Val*
- name(*Struct*) = *Name*
- new_array(D_1,...,D_n) = *Arr*
- new_list(*N*) = *List*
- new_map(*PairsList*) = *Map*
- new_struct(*Name*,*IntOrList*) = *Struct*
- nonvar(*Term*)
- not *Call*
- number(*Term*)
- number_chars(*Num*) = *String*
- number_codes(*Num*) = *List*
- number_vars(*Term*,N_0) = N_1
- once *Call*
- parse_term(*String*) = *Term*)
- parse_term(*String*,*Term*,*Vars*)
- parse_term(*String*,*Term*,*Vars*,*RString*)
- post_event(*X*,*Event*)
- post_event_any(*X*,*Event*)
- post_event_bound(*X*)
- post_event_dom(*X*,*Event*)
- post_event_ins(*X*)
- print(*Term*)
- printf(*Term*,*Args*...)
- println(*Term*)
- put(*MapOrAttrVar*,*Key*,*Val*)
- read_char(*N*) = *String*
- read_char() = *Val*
- read_int() = *Int*
- read_line() = *String*
- read_real() = *Real*
- read_term() = *Term*
- read_token() = *String*
- read_unicode_char(*N*) = *String*
- read_unicode_char() = *Val*
- readln() = *String*
- real(*Term*)
- remove_dups(*List*) = *ResList*
- repeat (nondet)
- reverse(*List*) = *ResList*
- select(*X*,*List*,*ResList*) (nondet)
- sort(*List*) = *SList*
- sort_down(*List*) = *SList*
- string(*Term*)
- struct(*Term*)
- sublist(*List*,*Start*,*End*) = *SubList*
- subsumes($Term_1$,$Term_2$)
- sum(*List*) = *Val*
- throw *E*
- to_array(*List*) = *Array*
- to_binary_string(*Int*) = *String*
- to_codes(*Term*) = *List*
- to_fstring(*Format*,*Term*) = *String*
- to_hex_string(*Int*) = *String*
- to_integer(*Num*) = *Int*
- to_list(*Struct*) = *List*
- to_lowercase(*String*) = *LString*
- to_oct_string(*Int*) = *String*
- to_real(*Num*) = *Real*
- to_string(*Term*) = *String*
- to_uppercase(*String*) = *UString*
- true
- unnumber_vars($Term_1$) = $Term_2$
- uppercase(*Char*)
- values(*MapOrAttrVar*) = *List*
- var(*Term*)
- variant($Term_1$,$Term_2$)
- vars(*Term*) = *Vars*
- write(*Term*)
- write_byte(*Bytes*)
- writef(*Term*,*Args*...)
- writeln(*Term*)
- zip($List_1$,$List_2$,...,$List_n$) = *List*

Module `math`

- `abs(`X`) = `Val
- `acos(`X`) = `Val
- `acosh(`X`) = `Val
- `asin(`X`) = `Val
- `asinh(`X`) = `Val
- `atan(`X`) = `Val
- `atan2(`X`,`Y`) = `Val
- `atanh(`X`) = `Val
- `cbrt(`X`) = `Val
- `ceiling(`X`) = `Val
- `cos(`X`) = `Val
- `cosh(`X`) = `Val
- `cot(`X`) = `Val
- `coth(`X`) = `Val
- `csc(`X`) = `Val
- `csch(`X`) = `Val
- `degrees(`$Radian$`) = `$Degree$
- `e = 2.71828`
- `exp(`X`) = `Val
- `floor(`X`) = `Val
- `inf`
- `log(`X`) = `Val
- `log(B,`X`) = `Val
- `log10(`X`) = `Val
- `log2(`X`) = `Val
- `modf(`X`) = (`$FractVal$`,`$IntVal$`)`
- `ninf`
- `nthrt(`N`,`X`) = `Val
- `pi = 3.14159`
- `pow(`X`,`Y`) = `Val
- `radians(`$Degree$`) = `$Radian$
- `random = `Val
- `random(`$Seed$`) = `Val
- `randrange(`$From$`,`$Step$`,`To`) = `Val
- `randrange(`$From$`,`To`) = `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
- `truncate(`X`) = `Val

Module `io`

- `at_end_of_stream(`FD`)`
- `close(`FD`)`
- `dup(`FD`) = `$NewFD$
- `dup2(`$FromFD$`,`$ToFD$`)`
- `eof`
- `flush(`FD`)`
- `fprint(`FD`,`$Term$`)`
- `fprintf(`FD`,`$Format$`,`$Args...$`)`
- `fprintln(`FD`,`$Term$`)`
- `fread_byte(`FD`) = `Val
- `fread_byte(`FD`,`N`) = `$List$
- `fread_char(`FD`) = `Val
- `fread_char(`FD`,`N`) = `$String$
- `fread_file_bytes(`FD`) = `$List$
- `fread_file_chars(`FD`) = `$String$
- `fread_int(`FD`) = `Int
- `fread_line(`FD`) = `$String$
- `fread_real(`FD`) = `$Real$
- `fread_term(`FD`) = `$Term$
- `fread_token(`FD`) = `$String$
- `fread_unicode_char(`FD`) = `Val
- `fread_unicode_char(`FD`,`N`) = `$String$
- `freadln(`FD`) = `$String$
- `fwrite(`FD`,`$Term$`)`
- `fwrite_byte(`$Bytes$`)`
- `fwritef(`FD`,`$Format$`,`$Args...$`)`
- `fwriteln(`FD`,`$Term$`)`
- `getpos(`FD`) = `Pos
- `mkfifo(`$Path$`)`
- `mkfifo(`$Path$`,`$Mode$`)`
- `mkpipe() = `FD_Map
- `mktmp() = `FD
- `open(`$Name$`) = `FD
- `open(`$Name$`,`$Mode$`) = `FD
- `peek_byte(`FD`) = `Val
- `peek_char(`FD`) = `Val
- `peek_int(`FD`) = `Int
- `peek_real(`FD`) = `$Real$
- `peek_unicode_char(`FD`) = `Val
- `rewind(`FD`)`
- `seek(`FD`,`$Offset$`,`$From$`)`
- `setpos(`FD`,`Pos`)`
- `sizeof_char() = `$Size$
- `stderr`
- `stdin`
- `stdout`

Module `os`

- `atime(`*Path*`) = `*DateTime*
- `block_special(`*Path*`)`
- `cd(`*Path*`)`
- `char_special(`*Path*`)`
- `chdir(`*Path*`)`
- `chmod(`*Path*`,`*Mode*`)`
- `cp(`*Path*`1,`*Path*`2)`
- `create(`*Path*`)`
- `create(`*Path*`,`*Mode*`)`
- `ctime(`*Path*`) = `*DateTime*
- `cwd() = `*Path*
- `dev_id(`*Path*`) = `*Int*
- `directory(`*Path*`)`
- `directory_exists(`*Path*`)`
- `executable(`*Path*`)`
- `exists(`*Path*`)`
- `fifo(`*Path*`)`
- `file(`*Path*`)`
- `file_base_name(`*Path*`) = `*String*
- `file_directory_name(`*Path*`) = `*String*
- `file_exists(`*Path*`)`
- `file_type(`*Path*`) = `*Term*
- `gid(`*Path*`) = `*Int*
- `ino(`*Path*`) = `*Int*
- `link(`*Path*`)`
- `link(`*Path*`1,`*Path*`2)`
- `listdir(`*Path*`) = `*List*
- `listdir(`*Path*`,`*REPattern*`) = `*List*
- `message_queue(`*Path*`)`
- `mkdir(`*Path*`)`
- `mkdir(`*Path*`,`*Mode*`)`
- `mkdirs(`*Path*`)`
- `mkdirs(`*Path*`,`*Mode*`)`
- `mode(`*Path*`) = `*String*
- `mode(`*Path*`,`*Value*`)`
- `mtime(`*Path*`) = `*DateTime*
- `mv(`*Path*`1,`*Path*`2)`
- `nlink(`*Path*`) = `*Int*
- `pwd() = `*Path*
- `readable(`*Path*`)`
- `rm(`*Path*`)`
- `rmdir(`*Path*`)`
- `root() = `*Path*
- `semaphore(`*Path*`)`
- `separator() = `*Val*
- `shared_memory(`*Path*`)`
- `shortcut(`*Path*`)`
- `shortcut(`*Path*`1,`*Path*`2)`
- `size(`*Path*`) = `*Int*
- `socket(`*Path*`)`
- `uid(`*Path*`) = `*Int*
- `unlink(`*Path*`)`
- `writable(`*Path*`)`

Modules `cp`, `sat`, and `mip`

- *X* `#=` *Y*
- *X* `#!=` *Y*
- *X* `#>` *Y*
- *X* `#>=` *Y*
- *X* `#<` *Y*
- *X* `#=<` *Y*
- *X* `#<=` *Y*
- `#~`*X*
- *X* `#\/` *Y*
- *X* `#/\` *Y*
- *X* `#^` *Y*
- *X* `#=>` *Y*
- *X* `#<=>` *Y*
- *Vars* `in` *Exp*
- *Vars* `notin` *Exp*
- `all_different(`*FDVars*`)`
- `all_distinct(`*FDVars*`)`
- `assignment(`*FDVars*`1,`*FDVars*`2)`
- `circuit(`*FDVars*`)`
- `count(`*V*`,`*FDVars*`,`*Rel*`,`*N*`)`
- `cumulative(`*Starts*`,`*Durations*`,`*Resources*`,`*Limit*`)`
- `diffn(`*RectangleList*`)`
- `disjunctive_tasks(`*Tasks*`)`
- `element(`*I*`,`*List*`,`*V*`)`
- `fd_degree(`*FDVar*`) = `*Degree*
- `fd_disjoint(`*DVar*`1,`*DVar*`2)`
- `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*`)`
- `fd_size(`*FDVar*`) = `*Size*
- `fd_superset(`*DVar*`1,`*DVar*`2)`
- `fd_true(`*FDVar*`,`*Elm*`)`
- `fd_var(`*Term*`)`
- `global_cardinality(`*List*`,`*Pairs*`)`
- `indomain(`*Var*`)`
- `indomain_down(`*Var*`)`
- `lp_in(`*Vars*`,`*LExp*`,`*UExp*`)`
- `neqs(`*NeqList*`)`
- `new_fd_var() = `*FDVar*
- `serialized(`*Starts*`,`*Durations*`)`
- `solve(`*Options*`,`*Vars*`)`
- `solve(`*Vars*`)`
- `subcircuit(`*FDVars*`)`

Module `thread`

- `acquire_mutex(`$Mutex$`)`
- `broadcast_cv(`CV`)`
- `join(`$Thread$`)`
- `new_cv() =` CV
- `new_mutex() =` $Mutex$
- `new_rwlock() =` $RWLock$
- `new_semaphore() =` $Semaphore$
- `new_semaphore(`N`) =` $Semaphore$
- `new_thread(`$S, Arg_1, \ldots, Arg_n$`) =` $Thread$
- `p_semaphore(`$Semaphore$`)`
- `rdlock(`$RWLock$`)`
- `release_mutex(`$Mutex$`)`
- `rwunlock(`$RWLock$`)`
- `signal_cv(`CV`)`
- `sleep(`$Milliseconds$`)`
- `start(`$Thread$`)`
- `this_thread() =` $Thread$
- `v_semaphore(`$Semaphore$`)`
- `wait_cv(`$CV, Mutex$`)`
- `wrlock(`$RWLock$`)`

Module `timer`

- `get_interval(`$Timer$`) =` $Milliseconds$
- `kill(`$Timer$`)`
- `new_timer(`$Milliseconds$`) =` $Timer$
- `set_interval(`$Timer, Milliseconds$`)`
- `start(`$Timer$`)`
- `stop(`$Timer$`)`

Module `process`

- `exec(`$S, Arg_1, \ldots, Arg_n$`)`
- `execl(`$S, ArgList$`)`
- `fork() =` ID
- `new_process(`$S, Arg_1, \ldots, Arg_n$`) =` ID
- `pid() =` ID
- `ppid() =` ID
- `wait() =` $StatMap$
- `waitpid(`ID`) =` $StatMap$

Module `socket`

- `accept(`FD`) =` $Client$
- `bind(`$FD, INet, Address, Port$`)`
- `bind(`$FD, Unix, Name$`)`
- `close(`FD`)`
- `connect(`$FD, INet, Address, Port$`)`
- `connect(`$FD, Unix, Name$`)`
- `getaddr(`$Name$`) =` $Addr$
- `getcanonicalname(`$Addr$`) =` $Name$
- `gethostbyaddr(`$Addr$`) =` $Host$
- `gethostbyname(`$Name$`) =` $Host$
- `getservbyname(`$Name$`) =` $Service$
- `getservbyname(`$Name, Type$`) =` $Service$
- `getservport(`$Name$`) =` $Port$
- `getsockopt(`$FD, Level, Option$`) =` $Value$
- `joingroup(`$GroupAddress$`)`
- `leavegroup(`$GroupAddress$`)`
- `listen(`FD`)`
- `listen(`$FD, Backlog$`)`
- `recv(`FD`) =` $Message$
- `recv(`$FD, Flags$`) =` $Message$
- `recvfrom(`$FD, Domain$`) =` $Message$
- `recvfrom(`$FD, Flags, Domain$`) =` $Message$
- `send(`$FD, Message$`) =` $NBytes$
- `send(`$FD, Message, Flags$`) =` $NBytes$
- `sendto(`$FD, Message, Domain, Address, Port$`) =` $NBytes$
- `sendto(`$FD, Message, Flags,` $Domain, Address, Port$`) =` $NBytes$
- `sendto(`$FD, Message, Flags, Name$`) =` $NBytes$
- `sendto(`$FD, Message, Name$`) =` $NBytes$
- `setsockopt(`$FD, Level, Option, Value$`)`
- `socket(`$Domain, Type$`) =` FD
- `tcp_bind(`$FD, Address, Port$`)`
- `tcp_connect(`$FD, Address, Port$`)`
- `tcp_socket() =` FD
- `udp_bind(`$FD, Address, Port$`)`
- `udp_socket() =` FD
- `unix_bind(`$FD, Name$`)`
- `unix_connect(`$FD, Name$`)`
- `unix_socket() =` FD

Module `sys` (imported by default)

- `abort`
- `cl(`*File*`)`
- `compile(`*File*`)`
- `debug`
- `execute(`*CommandString*`) = `*Status*
- `exit`
- `getenv(`*EnvironmentVarNameString*`) = `*String*
- `halt`
- `help`
- `initialize_table`
- `load(`*File*`)`
- `modules() = `*List*
- `nodebug`
- `nospy `*Functor*
- `nospy`
- `notrace`
- `profile(`*Goal*`)`
- `profile_src(`*File*`)`
- `prompt(`*NewPrompt*`)`
- `spy `*Functor*
- `statistics(`*Name*`,`*Value*`)` (nondet)
- `statistics`
- `table_get_all(`*Goal*`) = `*List*
- `table_get_one(`*Goal*`)`
- `trace`

Module `datetime`

- `add_days(`*DateTime*`,`*Days*`) = `*DateTime*
- `add_hours(`*DateTime*`,`*Hours*`) = `*DateTime*
- `add_milliseconds(`*DateTime*`,`*MilliSeconds*`) = `*DateTime*
- `add_minutes(`*DateTime*`,`*Minutes*`) = `*DateTime*
- `add_months(`*DateTime*`,`*Months*`) = `*DateTime*
- `add_seconds(`*DateTime*`,`*Seconds*`) = `*DateTime*
- `add_years(`*DateTime*`,`*Years*`) = `*DateTime*
- `compare(`*DateTime*`,`*DateTime*`) = `*Res*
- `current_datetime() = `*DateTime*
- `day(`*DateTime*`) = `*Day*
- `day_of_week(`*DateTime*`) = `*Atom*
- `day_of_year(`*DateTime*`) = `*Int*
- `day_string(`*DateTime*`) = `*String*
- `dt_to_fstring(`*Format*`,`*DateTime*`) = `*String*
- `hour(`*DateTime*`) = `*Hour*
- `is_leap_year(`*DateTime*`)`
- `millisecond(`*DateTime*`) = `*MilliSecond*
- `minute(`*DateTime*`) = `*Minute*
- `month(`*DateTime*`) = `*Month*
- `month_string(`*DateTime*`) = `*String*
- `second(`*DateTime*`) = `*Second*
- `set_day(`*DateTime*`,`*Day*`)`
- `set_hour(`*DateTime*`,`*Hour*`)`
- `set_millisecond(`*DateTime*`,`*MilliSecond*`)`
- `set_minute(`*DateTime*`,`*Minute*`)`
- `set_month(`*DateTime*`,`*Month*`)`
- `set_second(`*DateTime*`,`*Second*`)`
- `set_year(`*DateTime*`,`*Year*`)`
- `time_string(`*DateTime*`) = `*String*
- `year(`*DateTime*`) = `*Year*

Index