A User's Guide to Picat

Version 1.5

Neng-Fa Zhou and Jonathan Fruhman

Copyright ©picat-lang.org, 2013-2015.
Last updated Nov. 23, 2015

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 (e.g., loops) 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.

Picat aims to be a simple, and yet powerful, logic-based programming language 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 system can be used for not only symbolic computations, which is a traditional application domain of declarative languages, but also for scripting and modeling tasks.

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

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

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

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

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

- **T**abling: Tabling can be used to store the results of certain calculations in memory, allowing the program to do a quick table lookup instead of repeatedly calculating a value. As

computer memory grows, tabling is becoming increasingly important for offering dynamic programming solutions for many problems. The `planner` module, which is implemented by the use of tabling, has been shown to be a more efficient tool than ASP and PDDL for solving many planning problems.

The support of explicit unification, explicit non-determinism, tabling, and constraints makes Picat more suitable than functional and scripting languages for symbolic computations. Picat is arguably more expressive than Prolog for scripting and modeling. With arrays, loops, and list and array comprehensions, it is not rare to find problems for which Picat requires an order of magnitude fewer lines of code to describe than Prolog. Picat is more scalable than Prolog. The use of pattern-matching rather than unification facilitates indexing of rules. Picat is also more reliable than Prolog. In addition to explicit non-determinism, explicit unification, and a simple static module system, the lack of cuts, dynamic predicates, and operator overloading also improves the reliability of the language. Picat is not as powerful as Prolog for metaprogramming and it's impossible to write a meta-interpreter for Picat in Picat itself. Nevertheless, this weakness can be remedied with library modules for implementing domain-specific languages.

The Picat implementation is based on the B-Prolog engine. The current implementation is already ready for many kinds of applications. It will also serve as a foundation for new additions, including external language interfaces with C, Java, and Python, an external language interface with MySql, threads, sockets, Web services, and language processing modules. Anybody is welcome to contribute. The C source code is available to registered developers and users. Please contact picat@picat-lang.org.

Acknowledgements

The initial design of Picat was published in December 2012, and the first alpha version was released in May 2013. The following people have contributed to the project by reviewing the ideas, the design, the implementation, and/or the documentation: Roman Bartak, Nikhil Barthwal, Lei Chen, Veronica Dahl, Agostino Dovier, Julio Di Egidio, Christian Theil Have, Hakan Kjellerstrand, Nuno Lopes, Richard O'Keefe, Lorenz Schiffmann, Paul Tarau, and Jan Wielemaker. Special thanks to Hakan Kjellerstrand, who has been programming in Picat and blogging about Picat since May 2013. The system wouldn't have matured so quickly without Hakan's hundreds of programs. The `picat-lang.org` web page was designed by Bo Yuan (Bobby) Zhou. The initial stage of the Picat project was supported in part by the NSF under grant number CCF1018006.

The Picat implementation is based on the B-Prolog engine. It uses the following public domain modules: `token.c` by Richard O'Keefe; `getline.c` by Chris Thewalt; `bigint.c` by Matt McCutchen; `Lingeling` by Armin Biere; `MiniSAT` by Niklas Sorensson; `GLPK` by Andrew Makhorin; `Espresso` (by Berkeley).

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`.

$OSPromt$ `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. Users can also use the `picat` command to run a program directly as follows:

$OSPromt$ `picat` $File\ \ Arg_1\ \ Arg_2\ \ \ldots\ \ Arg_n$

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

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`,...,`t_n`]` or a *structure* in the form `$`s`(`t_1`,...,`t_n`)` where s stands for a structure name, n is called the *arity* of the structure, and each t_i ($1 \leq i \leq n$) is a *term* which is a variable or a value. The preceeding dollar symbol is 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, where each pair has the form $Key = Val$. The function `new_set(`S`)` returns a *map set* that initially contains the elements in list S. A *map-set* is a map in which every key is mapped to the atom `not_a_value`. The function

`new_array(`$I_1, I_2, \ldots, I_n$`)` returns an n-dimensional array, where each I_i is an integer expression specifying the size of a dimension. An n-dimensional array is a one-dimensional array where the arguments are (n-1)-dimensional arrays.

Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Picat allows function calls in arguments. For this reason, it requires structures to be preceded with a dollar symbol in order for them to be treated as data. Without the dollar symbol, the

command `S=point(1.0,2.0)` would call the function `point(1.0,2.0)` and bind `S` to its return value. In order to ensure safe interpretation of meta-terms in higher-order calls, Picat forbids the creation of terms that contain structures with the name `'.'`, index notations, array comprehensions, list comprehensions, 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:

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

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

An array comprehension takes the following form:

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

It is the same as:

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

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
Len = 3

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

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

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

Picat also allows OOP notations for accessing attributes and for calling predicates and functions. The notation $A_1.f(A_2,\ldots,A_k)$ is the same as $f(A_1,A_2,\ldots,A_k)$, unless A_1 is an atom, in which case A_1 must be a module qualifier for f. The notation $A.Attr$, where $Attr$ is not in the form $f(\ldots)$, 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> X = 5.to_binary_string().length
X = 3

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
Len = 3

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



```
D2 = 3

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

1.2 Defining Predicates

A predicate call either succeeds or fails, unless an exception occurs. A predicate call can return multiple answers through backtracking. The built-in predicate `true` always succeeds, and the built-in predicate `fail` (or `false`) always fails. A *goal* is made from predicate calls and statements, including conjunction (A, B and A `&&` B), disjunction ($A; B$ and 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`.[1] 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.

[1] Picat also accepts Prolog-style if-then-else in the form `(If->Then;Else)` and mandates the presence of the else-part.

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` `(`$M_{11},\ldots,M_{1n}$`)` `...` `(`$M_{m1},\ldots,M_{mn}$`)` where each M_{ij} is either $+$ (meaning indexed) or $-$ (meaning not indexed). For each index pattern `(`$M_{i1},\ldots,M_{in}$`)`, 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`,...,`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 $E_1 \in D_1, \ldots, E_n \in D_n$ that satisfies the conditions $Cond_1, \ldots, Cond_n$. 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*.

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

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

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

Example

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

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

read_list=List =>
```



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

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

L=`[]`
L_1=`[`e_1`|`L`]`
L_2=`[`e_2`|`L_1`]`
...
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) = fib(N-1)+fib(N-2).
```

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

Users can also give table modes to instruct the system on what answers to table. Mode-directed tabling is especially useful for dynamic programming problems. In mode-directed tabling, a plus-sign (+) indicates input, a minus-sign (-) indicates output, `max` indicates that the corresponding variable should be maximized, `min` indicates that the corresponding variable should be minimized, and `nt` indicates that the corresponding argument is not tabled.[2]

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.
```

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

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

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

Example

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

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

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

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

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

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

```
Picat> load("test_my_sum")

Picat> go
10
```

The command `load(File)` loads a module file into the system. If the file has not been compiled, then the `load` command compiles the file before loading it. If this module is dependent on other modules, then the other modules are loaded automatically if they are not yet in the system.[3] When a module is loaded, all of its public predicates and functions become visible to the interpreter.

The Picat module system is static, meaning that the binding of normal calls to their definitions takes place at compile time. For higher-order calls, however, Picat may need to search for their definitions at runtime. Several built-in modules are imported by default, including `basic`, `io`, `math`, and `sys`. For a normal call that is not higher-order in a module, the Picat compiler searches modules for a definition in the following order:

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

1.7 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`.[4]

Example

```
import cp.

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

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

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

[3] Dependent modules must be in a path that is specified by the environment variable PICATPATH.

[4] The `mip` module is not included yet.


```
import cp.

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

instance(N,A) =>
    N=9,
    A={{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 thrown by the system include `divide_by_zero`, `file_not_found`, `number_expected`, `interrupt`, and `out_of_range`. The exception `interrupt(keyboard)` is raised when `ctrl-c` is typed during a program's execution. The built-in predicate `throw` $Exception$ throws $Exception$.

All exceptions, including those raised by built-ins and interruptions, can be caught by catchers. A catcher is a call in the form: `catch(`$Goal$`,`$Exception$`,`$Handler$`)` which is equivalent to $Goal$, except when an exception is raised during the execution of $Goal$ that unifies

ExceptionPattern. When such an exception is raised, all of the bindings that have been performed on variables in *Goal* will be undone, and *Handler* will be executed to handle the exception.

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

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 `find_all` 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`)` returns a list of all possible solutions of `call(`S`)` in the form of *Template*. Other higher-order predicates include `call_cleanup/2`, `catch/3`, `count_all`, `freeze/2`, `not/1`, `maxof/2-3`, `maxof_inc/2-3`, `minof/2-3`, `minof_inc/2-3`, `once/1`, `time/1`, `time2/1`, and `time_out/3`. All of these higher-order predicates are defined in the `basic` module, except for `time/1`, `time2/1`, and `time_out/3`, which are defined in the `sys` module. Higher-order calls cannot contain assignments or loops.

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

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

The 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 `apply` is assumed to invoke a definition in a pre-imported built-in module, the enclosing module in which `apply` occurs, an imported module of the enclosing module, or the global module. Due to the overhead of runtime search, the use of higher-order calls is discouraged. Whenever possible, recursion, loops, or list and array comprehensions should be used instead.

1.10 Action Rules

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

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

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

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

Example

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

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

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

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

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

1.11 Prebuilt Maps

Three maps are prebuilt: a *heap* map, a *global* map, and a *table* map. The heap map is created on the heap immediately after the system is started. The built-in function `get_heap_map()` returns the heap map. The heap map is like a normal map. Users use `put` to add key-value pairs into the map. Users use `get` to retrieve a value that is associated with a key in the map. Changes to the map up to a choice point are undone when execution backtracks to that choice point.

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 table map is created in the table area when the Picat system is started. The built-in function `get_table_map()` returns this map. Like the global map, changes to the table map are not undone upon backtracking. Unlike the global map, however, keys and values are *hash-consed* so that common ground sub-terms are not replicated in the table area.

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

For the call `go`, the output is:


```
heap map has no key
global map has key
table 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 and the pair added to the table map remain.

1.12 Programming Exercises

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

Chapter 2

How to Use the Picat System

2.1 How to Use the Picat Interpreter

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

2.1.1 How to Enter and Quit the Picat Interpreter

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

OSPrompt `picat`

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

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

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

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

2.1.2 How to Run a Program Directly

The `picat` command can also be used to run a program directly. Consider the following program:

```
main =>
    printf("hello picat%n").

main(Args) =>
    printf("hello picat"),
    foreach(Arg in Args)
        printf(" %s",Arg)
    end,
    nl.
```


Assume the program is stored in a file named `hello.pi` in the directory `C:\work`. The following shows two runs, one executing `main/0` and the other executing `main/1`.

```
C:\work> picat hello.pi
hello picat

C:\work> picat hello.pi 6 12 2013
hello picat 6 12 2013
```

If the command line contains arguments after the file name, then `main/1` is executed and all the arguments are passed to the predicate as a list of strings.

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

`picat [-path` *Directories*`][-log] [-g` *InitGoal*`]` *File* Arg_1 Arg_2 ... Arg_n

Directories is a semicolon-separated and double-quoted list of directories, and will be set as the value of the environment variable `PICATPATH` before the execution of the program. The Picat system will look for the file and the related modules in these directories. The option `-log` makes the system print log information and warning messages. The option `-g` makes Picat execute a specified initial query *InitGoal* rather than the default `main` predicate. The file name *File* can have the extension `.pi`, and can contain a path of directories.

2.1.3 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.

2.1.4 How to Compile and Load Programs

A Picat program is stored in one or more text files with the extension name `pi`. A file name is a string of characters. Picat treats both `'/'` and `'\'` as 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 directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`. For $FileName$`.pi`, the `cl` command loads the generated byte-codes without creating a byte-code file.

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

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` 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 directory in which $FileName$`.qi` resides or the directories that are stored in the environment variable `PICATPATH`. If the byte-code file $FileName$`.qi` does not exist but the source file $FileName$`.pi` exists, then this built-in compiles the source file and loads the byte codes without creating a `qi` file.

2.1.5 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

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

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

Consider, for example, the following program:

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

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

```
Picat> debug

{Trace mode}
Picat> cl(myprog)

{Trace mode}
Picat> p(X),q(Y)
   Call: (1) p(_328) ?
   Exit: (1) p(a)
   Call: (2) q(_378) ?
   Exit: (2) q(1)
X = a
Y = 1 ?;
   Redo: (2) q(1) ?
   Exit: (2) q(2)
X = a
Y = 2 ?;
   Redo: (1) p(a) ?
   Exit: (1) p(b)
   Call: (3) q(_378) ?
   Exit: (3) q(1)
X = b
Y = 1 ?;
   Redo: (3) q(1) ?
   Exit: (3) q(2)
X = b
Y = 2 ?;
no
```

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

`spy $`$Name/N$

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

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

Chapter 3

Data Types, Operators, and Built-ins

Picat is a dynamically-typed language, in which type checking occurs at runtime. A variable gets a type once it is bound to a value. In Picat, variables and values are terms. A value can be *primitive* or *compound*. A primitive value can be an *integer*, a *real number*, or an *atom*. A compound value can be a *list* or a *structure*. Strings, arrays, 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 `&&`), and disjunction (`;` and `||`), will be described in Chapter 4 on Predicates and Functions. The constraint operators (the ones that begin with `#`) will be described in Chapter 12 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 $A.f(A_2, \ldots, A_k)$ is the same as $f(A_1, A_2, \ldots, A_k)$, unless A_1 is an atom, in which case A_1 must be a module qualifier for f. If an atom needs to be passed as the first argument to a function or a predicate, then this notation cannot be used. The notation $A.Attr$, 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 `X.f().g()` is the same as `g(f(X))`.

The following functions are provided for all terms:

- `copy_term(`$Term_1$`) = `$Term_2$: This function copies $Term_1$ into $Term_2$. If $Term_1$ is an attributed variable, then $Term_2$ will not contain any of the attributes.
- `hash_code(`$Term$`) = `$Code$: This function returns the hash code of $Term$. If $Term$ is a variable, then the returned hash code is always 0.
- `to_codes(`$Term$`) = `$Codes$: This function returns a list of character codes of $Term$.
- `to_fstring(`$Format$`, `$Args\ldots$`)`: This function converts the arguments in the $Args\ldots$ parameter into a string, according to the format string $Format$, and returns the string. The number of arguments in $Args\ldots$ cannot exceed 10. Format characters are described in Chapter 10.
- `to_string(`$Term$`) = `$String$: This function returns a string representation of $Term$.

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

Table 3.1: Operators in Picat

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

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
```

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.
- `dvar(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable.
- `bool_dvar(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable whose lower bound is 0 and whose upper bound is 1.
- `dvar_or_int(`$Term$`)`: This predicate is true if $Term$ is an attributed domain variable or an integer.
- `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.
- `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.
- `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.
- `put(`X`, `Key`)`: This predicate call is the same as `put(`X`, `Key`,not_a_value)`.

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.
- `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.
- `atomic(`$Term$`)`: This predicate is true if $Term$ is an atom or a number.
- `char(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom is made of one character.
- `chr(`$Code$`)` = $Char$: This function returns the UTF-8 character of the code point $Code$.
- `digit(`$Term$`)`: This predicate is true if $Term$ is an atom and the atom is made of one digit.
- `len(`$Atom$`)` = Len: This function returns the number of characters in $Atom$. Note that this function is overloaded in such a way that the argument can also be an array, a list, or a structure.
- `length(`$Atom$`)` = Len: This function is the same as `len(`$Atom$`)`.
- `ord(`$Char$`)` = Int: This function returns the code point of the UTF-8 character $Char$. It throws an error if $Char$ is not a single-character atom.

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: Arithmetic Operators

X `**` Y	power
`+`X	same as X
`-`X	sign reversal
`~`X	bitwise complement
X `*` Y	multiplication
X `/` Y	division
X `//` Y	integer division, truncated
X `/>` Y	integer division (ceiling(X / Y))
X `/<` Y	integer division (floor(X / Y))
X `div` Y	integer division, floored
X `mod` Y	modulo, same as X - floor(X div Y) * Y
X `rem` Y	remainder (X - (X // Y) * Y)
X `+` Y	addition
X `-` Y	subtraction
X `>>` Y	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

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`)`.
- `float(`$Term$`)`: This predicate is true if $Term$ is a real number.
- `integer(`$Term$`)`: This predicate is true if $Term$ is an integer.
- `max(`X`, `Y`)` = Val: This function returns the maximum of X and Y, where X and Y are terms.
- `min(`X`, `Y`)` = Val: This function returns the minimum of X and Y, where X and Y are terms.
- `number(`$Term$`)`: This predicate is true if $Term$ is a number.
- `number_chars(`Num`)` = $String$: This function returns a list of characters of Num. This function is the same as `to_fstring("%d",`Num`)` if Num is an integer, and 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()`.
- `real(`$Term$`)`: This predicate is the same as `float(`$Term$`)`.
- `to_binary_string(`Int`)` = $String$: This function returns the binary representation of the integer Int as a string.
- `to_hex_string(`Int`)` = $String$: This function returns the hexadecimal representation of the integer Int as a string.
- `to_integer(`NS`)` = Int: This function is the same as `truncate(`NS`)` in the `math` module if NS is a number, and the same as `parse_term(`NS`)` if NS is a string.
- `to_oct_string(`Int`)` = $String$: This function returns the octal representation of the integer Int as a string.
- `to_real(`NS`)` = $Real$: This function is the same as NS`*1.0` if NS is a number, and the same as `parse_term(`NS`)` if NS is a string.

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.

3.4.1 Lists

A list takes the form `[`t_1`,...,`t_n`]`, where each t_i ($1 \leq i \leq n$) is a term. Let L be a list. The expression L`.length`, which is the same as 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:

- $List_1$ `++` $List_2$ = $List$: This function returns the concatenated list of $List_1$ and $List_2$. Note that this function is overloaded for arrays.
- `append(`X`, `Y`, `Z`)` (nondet): This predicate is true if appending Y to X can create Z. This predicate may backtrack if X is not a complete list.[1]
- `append(`W`, `X`, `Y`, `Z`)` (nondet): This predicate is defined as:

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

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

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

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

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

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

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

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

- `insert_ordered_down(`*List*`,`*Term*`)`: This function inserts *Term* into the descendantly ordered list *List*, such that the resulting list remains sorted down.

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

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

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

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

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

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

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

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

- `new_list(`*N*`) = `*List*: This function creates a new list that has *N* free variable arguments.

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

- `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(`*List*`,`*KeyIndex*`) = `*SList*: This function sorts the elements of *List* by the key index *KeyIndex* in ascending order, returning the result in *SList*. The elements of *List* must be compound values and *KeyIndex* must be a positive integer that does not exceed the length of any of the elements of *List*. This function is defined as follows:

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

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

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

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

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

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

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

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

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

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

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

- `slice(`*List*`,`*From*`,`*To*`) = `*SList*: This function returns the sliced list of *List* from index *From* through index *To*. *From* must not be less than 1. Note that this function is overloaded for arrays.

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

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

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

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

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

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

3.4.2 Strings

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

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

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

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

3.4.3 Structures

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

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

Picat disallows creation of the following types of structures:

Dot notations:	`math.pi, my_module.f(a)`
Index notations:	`X[1]+2, X[Y[I]]`
Assignments:	`X:=Y+Z, X:=X+1`
Ranges:	`1..10, 1..2..10`
List comprehensions:	`[X : X in 1..5]`
Array comprehensions:	`{X : X in 1..5}`
If-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 term constructor.

The following built-ins are provided for structures:

- `arity(`$Struct$`) = `$Arity$: This function returns the arity of $Struct$, which must be a structure.

- `new_struct(`$Name$`, `$IntOrList$`) = `$Struct$: This function creates a structure that has the name $Name$. If $IntOrList$ is an integer, N, then the structure has N free variable arguments. Otherwise, if $IntOrList$ is a list, then the structure contains the elements in the list.

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

- `to_list(`$Struct$`) = `$List$: This function returns a list of the components of the structure $Struct$.

3.4.4 Arrays

An *array* takes the form $\{t_1, \ldots, t_n\}$, 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:

- $Array_1$ `++` $Array_2$ `=` $Array$: This function returns the concatenated array of $Array_1$ and $Array_2$. Note that this function is overloaded for lists.

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

- `first(`$Array$`) = `$Term$: This function returns the first element of $Array$.

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

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

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

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

- `slice(`$Array$`,`$From$`,`To`) = `$SArray$: This function returns the sliced array of $Array$ from index $From$ through index To. $From$ must not be less than 1. Note that this function is overloaded for lists.

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

  `slice(`$Array$`,`$From$`,`$Array$`.length)`

3.4.5 Maps

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

- `clear(`Map`)`: This predicate clears the map Map. It throws an error if Map is not a map.

- `get(`Map`, `Key`) = `Val: This function returns Val of the key-value pair Key=Val in Map. It throws an error if Map does not contain the key Key.
- `get(`Map`, `Key`, `$DefaultVal$`) = `Val: This function returns Val of the key-value pair Key=Val in Map. It returns $DefaultVal$ if Map does not contain Key.
- `has_key(`Map`, `Key`)`: This predicate is true if Map contains a pair with Key.
- `keys(`X`) = `$List$: This function returns the list of keys of the pairs in Map.
- `map(`$Term$`)`: This predicate is true if $Term$ is a map.
- `map_to_list(`Map`) = `$PairsList$: This function returns a list of Key=Val pairs that constitute Map.
- `new_map(`$IntOrPairsList$`) = `Map: This function creates a map with an initial capacity or an initial list of pairs.
- `new_map(`N`,`$PairsList$`) = `Map: This function creates a map with the initial capacity N, the initial list of pairs $PairsList$, where each pair has the form Key=Val.
- `put(`Map`, `Key`, `Val`)`: This predicate attaches the key-value pair Key=Val to Map, where Key is a non-variable term, and Val is any term.
- `put(`Map`, `Key`)`: This predicate is the same as `put(`Map`, `Key`, not_a_value)`.
- `values(`Map`) = `$List$: This function returns the list of values of the pairs in Map.
- `size(`Map`) = `$Size$: This function returns the number of pairs in Map.

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

3.4.6 Sets

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

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

3.5 Equality Testing, Unification, and Term Comparison

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

The predicate `bind_vars(`$Term$`,`Val`)` This predicate binds all of the variables in $Term$ to Val.

Example

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

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

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

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

3.6 Expressions

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

- An expression can contain dot notations, such as `math.pi`.
- An expression can contain index notations, such as `X[I]`.
- An expression can contain ranges, such as `1..2..100`.

- An expression can contain list comprehensions, such as `[X : X in 1..100]`.
- An expression can contain array comprehensions, such as {`X : X in 1..100`}.

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

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

3.7 Higher-order Predicates and Functions

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

- `apply(`S`,` Arg_1`, ...,` Arg_n`)` `=` Val: S is an atom or a structure. This function calls the function that is named by S with the arguments that are specified in S, together with extra arguments $Arg_1, \ldots, Arg_n$. This function returns the value that S returns.
- `call(`S`,` Arg_1`, ...,` Arg_n`)`: S is an atom or a structure. This predicate calls the predicate that is named by S with the arguments that are specified in S, together with extra arguments $Arg_1, \ldots, Arg_n$.
- `call_cleanup(`$Call$`,` $Cleanup$`)`: This predicate is the same as `call(`$Call$`)`, except that $Cleanup$ is called when $Call$ succeeds determinately (i.e., with no remaining choice point), when $Call$ fails, or when $Call$ raises an exception.
- `catch(`$Call$`,` $Exeption$`,` $Handler$`)`: This predicate is the same as $Call$, except when an exception that matches $Exception$ is raised during the execution of $Call$. When such an exception is raised, all of the bindings that have been performed on variables in $Call$ will be undone, and $Handler$ will be executed to handle the exception.
- `findall(`$Template$`,`$Call$`)` `=` $Answers$: This function returns a list of all possible instances of `call(`$Call$`)` that are true in the form of $Template$. Note that $Template$ is assumed to be a term without function calls, and that $Call$ is assumed to be a predicate call whose arguments can contain function calls. Also note that, like a loop, `findall` forms a name scope. For example, in `findall(f(X),p(X,g(Y)))`, `f(X)` is a term even though it is not preceded with `$`; `g(Y)` is a function call; the variables `X` and `Y` are assumed to be local to `findall` if they do not occur before in the outer scope.
- `find_all(`$Template$`,`$Call$`)` `=` $Answers$: This function is the same as the above function.
- `count_all(`$Call$`)` `=` $Count$: This function returns the number of all possible instances of `call(`$Call$`)` that are true. For example, `count_all(member(X,[1,2,3]))` returns 3.
- `freeze(`X`,` $Call$`)`: This predicate delays the evaluation of $Call$ until X becomes a non-variable term.
- `map(`$FuncOrList$`,` $ListOrFunc$`)` `=` $ResList$: This function applies a function to every element of a given list and returns a list of the results. One of the arguments is a function, and the other is a list. The order of the arguments is not important.

- `map(`$Func$`,` $List1$`,` $List2$`)` `=` $ResList$: Let $List1$ be `[`A_1`, ..., `A_n`]` and $List2$ be `[`B_1`, ..., `B_n`]`. This function applies the function $Func$ to every pair of elements (A_i, B_i) by calling `apply(`$Func$`,` A_i`,` B_i`)`, and returns a list of the results.
- `maxof(`$Call$`,` $Objective$`)`: This predicate finds a satisfiable instance of $Call$, such that $Objective$ has the maximum value. Here, $Call$ is used as a generator, and $Objective$ is an expression to be maximized. For every satisfiable instance of $Call$, $Objective$ must be a ground expression. For `maxof`, search is restarted with a new bound each time that a better answer is found.
- `maxof(`$Call$`,` $Objective$`,` $ReportCall$`)`: This is the same as `maxof(`$Call$`,` $Objective$`)`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.
- `maxof_inc(`$Call$`,` $Objective$`)`: This is the same as `maxof(`$Call$`,` $Objective$`)`, except that search continues rather than being restarted each time that a better solution is found.
- `maxof_inc(`$Call$`,` $Objective$`,` $ReportCall$`)`: This is the same as the previous predicate, except that `call(`$ReportCall$`)` is executed each time that an answer is found.
- `minof(`$Call$`,` $Objective$`)`: This predicate finds a satisfiable instance of $Call$, such that $Objective$ has the minimum value.
- `minof(`$Call$`,` $Objective$`,` $ReportCall$`)`: This is the same as `minof(`$Call$`,` $Objective$`)`, except that `call(`$ReportCall$`)` is executed each time that an answer is found.
- `minof_inc(`$Call$`,` $Objective$`)`: This predicate is the same as `minof(`$Call$`,` $Objective$`)`, except that search continues rather than being restarted each time that a better solution is found.
- `minof_inc(`$Call$`,` $Objective$`,` $ReportCall$`)`: This predicate is the same as the previous one, except that `call(`$ReportCall$`)` is executed each time that an answer is found.
- `reduce(`$Func$`,` $List$`)` `=` Res: If $List$ contains only one element, this function returns the element. If $List$ contains at least two elements, then the first two elements A_1 and A_2 are replaced with `apply(`$Func$`,` A_1`,` A_2`)`. This step is repeatedly applied to the list until the list contains a single element, which is the final value to be returned.
- `reduce(`$Func$`,` $List$`,` $InitVal$`)` `=` Res: This function is the same as `reduce(`$Func$`, [`$InitVal$`|`$List$`])`.

3.8 Other Built-ins in the `basic` Module

- `acyclic_term(`$Term$`)`: This predicate is true if $Term$ is acyclic, meaning that $Term$ does not contain itself.
- `and_to_list(`$Conj$`)` `=` $List$: This function converts $Conj$ in the form `(`a_1`, ..., `a_n`)` into a list in the form `[`a_1`, ..., `a_n`]`.
- `compare_terms(`$Term_1$`,` $Term_2$`)` `=` Res: This function compares $Term_1$ and $Term_2$. If $Term_1 < Term_2$, then this function returns -1. If $Term_1 == Term_2$, then this function returns 0. Otherwise, $Term_1 > Term_2$, and this function returns 1.
- `different_terms(`$Term_1$`,` $Term_2$`)`: This constraint ensures that $Term_1$ and $Term_2$ are different. This constraint is suspended when the arguments are not sufficiently instantiated.

- `get_global_map()` = Map: This function returns the global map, 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.
- `get_table_map()` = Map: This function returns the current thread's table map. Each thread has its own table map. The table map is stored in the table area and both keys and values are hash-consed (i.e., common sub-terms are shared).
- `ground(`$Term$`)`: This predicate is true if $Term$ is ground. A *ground* term does not contain any variables.
- `list_to_and(`$List$`)` = $Conj$: This function converts $List$ in the form `[`a_1`,...,`a_n`]` into a term in the form `(`a_1`,...,`a_n`)`.
- `number_vars(`$Term$`,` 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.
- `parse_term(`$String$`,` $Term$`,` $Vars$`)`: 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.
- `parse_term(`$String$`)` = $Term$: This function converts $String$ to a term.
- `second(`$Compound$`)` = $Term$: This function returns the second argument of the compound term $Compound$.
- `subsumes(`$Term_1$`,` $Term_2$`)`: This predicate is true if $Term_1$ subsumes $Term_2$.
- `variant(`$Term_1$`,` $Term_2$`)`: This predicate is true if $Term_2$ is a variant of $Term_1$.
- `vars(`$Term$`)` = $Vars$: This function returns a list of variables that occur in $Term$.

Chapter 4

Predicates and Functions

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

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

and the *backtrackable* rule

$$Head, Cond \texttt{ ?=> } Body.$$

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)).

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


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

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

4.3 Patterns and Pattern-Matching

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

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

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

A pattern can contain *as-patterns* in the form V`@`$Pattern$, 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.
- `false`: This goal is the same as `fail`.
- $p(t_1, \ldots, t_n)$: This goal is a predicate call. The arguments $t_1, \ldots, t_n$ are evaluated in the given order, and the resulting call is resolved using the rules in the predicate p/n. If the call succeeds, then variables in the call may get instantiated. Many built-in predicates are written in infix notation. For example, `X=Y` is the same as `'='(X,Y)`.
- P`,` Q: This goal is a conjunction of goal P and goal Q. It is resolved by first resolving P, and then resolving Q. The goal is true if both P and Q are true. Note that the order is important: (P, Q) is in general not the same as (Q, P).
- P `&&` Q: This is the same as `(`P`,` Q`)`.
- P`;` Q: This goal is a disjunction of goal P and goal Q. It is resolved by first resolving P. If P is true, then the disjunction is true. If P is false, then Q is resolved. The disjunction is true if Q is true. The disjunction is false if both P and Q are false. Note that a disjunction can succeed more than once. Note also that the order is important: (P; Q) is generally not the same as (Q; P).
- P `||` Q: This is the same as `(`P`;` Q`)`.
- `not` P: This goal is the negation of P. It is false if P is true, and true if P is false. Note a negation goal can never succeed more than once. Also note that no variables can get instantiated, no matter whether the goal is true or false.
- `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.

- `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 and rules cannot co-exist in predicate definitions and facts must be ground. A predicate definition that consists of facts must 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). Facts are translated into pattern-matching rules before they are compiled.

Example

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

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

```
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.
```

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).

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

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

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

Chapter 5

Assignments and Loops

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

5.1 Assignments

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

Example

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

The compiler needs to give special consideration to the *scope* of a variable. The scope of a variable 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 (E in L)
        println(E)
    end.

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

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

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

Example

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

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

Example

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

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

Example:

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

5.2.2 Foreach Loops with Multiple Iterators

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

Example:

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

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

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

```
loop_ex5 =>
    L = [2, 3, 5, 10],
    foreach(I in L)
        foreach(J in 1..10)
            if J mod I != 0 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 and Array 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.

An array comprehension takes the following form:

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

It is the same as:

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

Example

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

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

5.4 Compilation of Loops

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

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

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

Consider the example:

Example

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

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

unable to create new variables for this assignment. In order to solve this problem, the compiler compiles while loops into tail-recursive predicates.

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

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

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

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

Suppose that a while loop modifies a variable that is then used outside of the while loop. For each modified variable `V` that is used after the while loop, predicate `p` is passed two arguments, `Vin` and `Vout`. Then, a predicate that has the body `true` is not sufficient to terminate the compiled while loop. 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 is also visible in the inner loop. However, variables that are local to the inner loop are not visiable to the outer loop.

For example, consider the nested loops:

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

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

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

5.4.1 List Comprehensions

List comprehensions are compiled into foreach loops.

Example

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

This list comprehension is compiled to:

```
comp_ex =>
    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:[1]

- `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.

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

- `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

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

```
catch(Goal,ExceptionPattern,Recovergoal)
```

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

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

Chapter 7

Tabling

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

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

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

Another issue with recursion is redundancy. Consider the following problem: *Starting in the top left corner of 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. The last mode M_n can be `nt`, which indicates that the argument is not tabled. Two types of data can be passed to a tabled predicate as an `nt` argument: (1) global data that are the same to all the calls of the predicate, and (2) data that are functionally dependent on the input arguments. 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)
sp(X,Y,Path,W) ?=>
    Path = [(X,Y)],
    edge(X,Y,W).
sp(X,Y,Path,W) =>
    Path = [(X,Z)|Path1],
    edge(X,Z,Wxz),
    sp(Z,Y,Path1,W1),
    W = Wxz+W1.
```

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

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

```
table (+,+,-,min).
sp(X,Y,Path,WL) ?=>
    Path = [(X,Y)],
    WL = (Wxy,1),
```

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

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

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

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

The predicate `action(Action,S,S1)` selects an action and performs the action on state `S` to generate another state, `S1`.

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. The tabling area can be initialized with the following built-in predicate:

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

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

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

Subgoal lookup and registration: This operation is used when a tabled subgoal is encountered during execution. It looks up the subgoal table to see if there is a variant of the subgoal.


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

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

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

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

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

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

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.

Chapter 8

The `planner` Module

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

- `final(`S`)`: This predicate succeeds if S is a final state.
- `final(`S`,`$Plan$`,`$Cost$`)`: A final state can be reached from S by the action sequence in $Plan$ with $Cost$. If this predicate is not given, then the system assumes the following definition:

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

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

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

8.1 Depth-Bounded Search

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

- `plan(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to a plan that can transform state S to a final state that satisfies the condition given by `final/1` or `final/3`. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.
- `plan(`S`,`$Limit$`,`$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `plan/4` predicate, except that the plan's cost is not returned.

- `plan(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is a variable, then this predicate is the same as the `plan/4` predicate, except that the limit is assumed to be 268435455.
- `plan(`S`,`$Plan$`)`: This predicate is the same as the `plan/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `best_plan(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate finds an optimal plan by using the *downward* algorithm. It first calls `plan/4` to find a plan of 0 cost. If no plan is found, then it increases the cost by 1. In this way, the first plan that is found is guaranteed to be optimal.
- `best_plan(`S`,`$Limit$`,`$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan/4` predicate, except that the plan's cost is not returned.
- `best_plan(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan/4` predicate, except that the limit is assumed to be 268435455.
- `best_plan(`S`,`$Plan$`)`: This predicate is the same as the `best_plan/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `best_plan_nondet(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate is the same as `best_plan(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`, except that it allows multiple best plans to be returned through backtracking.
- `best_plan_nondet(`S`,`$Limit$`,`$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_nondet/4` predicate, except that the plan's cost is not returned.
- `best_plan_nondet(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_nondet/4` predicate, except that the limit is assumed to be 268435455.
- `best_plan_nondet(`S`,`$Plan$`)`: This predicate is the same as the `best_plan_nondet/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `best_plan_upward(`S`,`$Limit$`,`$Plan$`,`$Cost$`)` (deprecated):
- `best_plan_bb(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to an optimal plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer. The following branch-and-bound algorithm is used to find an optimal plan: First, call `plan/4` to find a plan. Then, try to find a better plan by imposing a stricter limit. This step is repeated until no better plan can be found. Finally, return the last plan that was found.
- `best_plan_upward(`S`,`$Limit$`,`$Plan$`)` (deprecated)
- `best_plan_bb(`S`,`$Limit$`,`$Plan$`)`: If the second argument is an integer, then this predicate is the same as the `best_plan_bb/4` predicate, except that the plan's cost is not returned.
- `best_plan_upward(`S`,`$Plan$`,`$PlanCost$`)` (deprecated)

- `best_plan_bb(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is a variable, then this predicate is the same as the `best_plan_bb/4` predicate, except that the limit is assumed to be 268435455.
- `best_plan_upward(`S`,`$Plan$`)` (deprecated)
- `best_plan_bb(`S`,`$Plan$`)`: This predicate is the same as the `best_plan_bb/4` predicate, except that the limit is assumed to be 268435455, and that the plan's cost is not returned.
- `current_resource()`=$Amount$: This function returns the current available resource amount of the current node. If the current execution path was not initiated by one of the calls that performs resource-bounded search, then 268435455 is returned. This function can be used to check the heuristics. If the heuristic estimate of the cost to travel from the current state to a final state is greater than the available resource amount, then the current state can be failed.
- `current_plan()`=$Plan$: This function returns the current plan that has transformed the initial state to the current state. If the current execution path was not initiated by one of the calls that performs resource-bounded search, then [] is returned.
- `current_resource_plan_cost(`$Amount$`,`$Plan$`,`$Cost$`)`: This predicate retrieves the attributes of the current node in the search tree, including the resource amount, the path to the node, and its cost.
- `is_tabled_state(`S`)`: This predicate succeeds if the state S has been explored before and has been tabled.

8.2 Depth-Unbounded Search

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

- `plan_unbounded(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to a plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.
- `plan_unbounded(`S`,`$Limit$`,`$Plan$`)`
- `plan_unbounded(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is an integer, then this predicate is the same as the above predicate, except that the plan's cost is not returned. Otherwise, if the second argument is a variable, then this predicate is the same as the above predicate, except that the limit is assumed to be 268435455.
- `plan_unbounded(`S`,`$Plan$`)`: This predicate is the same as the above predicate, except that the limit is assumed to be 268435455.
- `best_plan_unbounded(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`: This predicate, if it succeeds, binds $Plan$ to an optimal plan that can transform state S to a final state. $Cost$ is the cost of $Plan$, which cannot exceed $Limit$, which is a given non-negative integer.
- `best_plan_unbounded(`S`,`$Limit$`,`$Plan$`)`

- `best_plan_unbounded(`S`,`$Plan$`,`$PlanCost$`)`: If the second argument is an integer, then this predicate is the same as the above predicate, except that the plan's cost is not returned. Otherwise, if the second argument is a variable, then this predicate is the same as the above predicate, except that the limit is assumed to be 268435455.

- `best_plan_unbounded(`S`,`$Plan$`)`: This predicate is the same as the above predicate, except that the limit is assumed to be 268435455.

8.3 Example

The program shown in Figure 8.1 solves the Farmer's problem by using the `planner` module. The `best_plan_unbounded(S0,Plan)` searches for a shortest plan in exactly the same way as the program in Figure 7.1.


```
import planner.

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

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

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

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

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

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

Chapter 9

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.

9.1 Module and Import Declarations

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

`module` $Name$.

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

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

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

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

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

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

9.2 Binding Calls to Definitions

The Picat system has a global symbol table for atoms, a global symbol table for structure names, and a global symbol table for modules. For each module, Picat maintains a symbol table for the

public predicate and function symbols defined in the module. Private symbols that are defined in a module are compiled away, and are never stored in the symbol table. While predicate and function symbols can be local to a module, atoms and structures are always global.

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

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

Example

```
% qsort.pi
module qsort.

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

```
% isort.pi
module isort.

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

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

The module `qsort.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]
```

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

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

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

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

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

9.3 Binding Higher-Order Calls

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

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

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

9.4 Library Modules

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

Chapter 10

I/O

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

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

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

Example

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

10.1 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`, or `append`. The `read` atom is used for reading from a file; if the file does not exist, or the program tries to write to the file, then the program will throw an error. The `write` atom is used for reading from a file and writing to a file; if the file already exists, then the file will be overwritten. The `append` atom is similar to the write atom; however, if the file already exists, then data will be appended at the end of the pre-existing file.

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

The `read` functions in the `io` module take a file descriptor as the first parameter. This file descriptor is the same descriptor that the `open` function returns. The parameter can be omitted if it is the standard input file `stdin`.

- `read_int(`FD`) =` Int: This function reads a single integer from the file that is represented by FD. It throws an `input_mismatch` exception if FD is at the end of the file or the next token at FD is not an integer.
- `read_int() =` Int: This function is the same as `read_int(stdin)`.
- `read_real(`FD`) =` $Real$: This function reads a single real number from the file that is represented by FD. It throws an `input_mismatch` exception if FD is at the end of the file or the next token at FD is not a number.
- `read_real() =` $Real$: This function is the same as `read_real(stdin)`.
- `read_char(`FD`) =` Val: This function reads a single UTF-8 character from the file that is represented by FD. It returns `end_of_file` if FD is at the end of the file.
- `read_char() =` Val: This function is the same as `read_char(stdin)`.
- `read_char(`FD`, `N`) =` $String$: This function reads up to N UTF-8 characters from the file that is represented by FD. It returns a string that contains the characters that were read.
- `read_char_code(`FD`) =` Val: This function reads a single UTF-8 character from the file that is represented by FD and returns its code point. It returns -1 if FD is at the end of the file.
- `read_char_code() =` Val: This function is the same as `read_char_code(stdin)`.
- `read_char_code(`FD`, `N`) =` $List$: This function reads up to N UTF-8 characters from the file that is represented by FD. It returns a list of code points of the characters that were read.
- `read_picat_token(`FD`,`$TokenType$`,`$TokenValue$`)`: This predicate reads a single Picat token from the file that is represented by FD. $TokenType$ is the type and $TokenValue$ is the value of the token. $TokenType$ is one of the following: `atom`, `end_of_file`, `end_of_rule`, `integer`, `punctuation`, `real`, `string`, `underscore`, and `var`.
- `read_picat_token(`$TokenType$`,`$TokenValue$`)`: This predicate reads a token from `stdin`.
- `read_picat_token(`FD`) =` $TokenValue$: This function reads a single Picat token from the file that is represented by FD and returns the token value.
- `read_picat_token() =` $TokenValue$: This function is the same as the above, except that it reads from `stdin`.

- `read_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.
- `read_term() =` $Term$: This function is the same as `read_term(stdin)`.
- `read_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 `end_of_file` atom is returned. The newline is not stored in the returned string.
- `read_line() =` $String$: This function is the same as `read_line(stdin)`.
- `readln(`FD`) =` $String$: This function does the same thing as `read_line`.
- `readln() =` $String$: This function is the same as `readln(stdin)`.
- `read_byte(`FD`) =` Val: This function reads a single byte from the file that is represented by FD.
- `read_byte() =` Val: This function is the same as `read_byte(stdin)`.
- `read_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.
- `read_file_bytes(`$File$`) =` $List$: This function reads an entire byte file into a list.
- `read_file_bytes() =` $List$: This function reads an entire byte file from the console into a list.
- `read_file_chars(`$File$`) =` $String$: This function reads an entire character file into a string.
- `read_file_chars() =` $String$: This function reads an entire character file from the console into a string.
- `read_file_codes(`$File$`) =` $List$: This function reads UTF-8 codes of an entire character file into a list.
- `read_file_codes() =` $List$: This function reads UTF-8 codes of an entire character file from the console into a list.
- `read_file_lines(`$File$`) =` $Lines$: This function reads an entire character file into a list of line strings.
- `read_file_lines() =` $Lines$: This function reads an entire character file from the console into a list of line strings.
- `read_file_terms(`$File$`) =` $Lines$: This function reads an entire text file into a list of terms. In the file, each term must be terminated by '.' followed by at least one white space.
- `read_file_terms() =` $Lines$: This function reads an entire text file from the console into a list of terms.

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

- peek_char(FD) = Val
- peek_byte(FD) = Val

10.2.1 End of File

The end of a file is detected through the end_of_file atom. If the input function returns a single value, and the read/write pointer is at the end of the file, then the end_of_file 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 end_of_file 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 end_of_file atom will not be returned until the next input function is called.

Instead of checking for end_of_file, the at_end_of_stream predicate can be used to monitor a file descriptor for the end of a file.

- at_end_of_stream(FD): The at_end_of_stream predicate is demonstrated in the following example.

Example

```
rw =>
     Reader = open("file1.txt"),
     Writer = open("file2.txt", write),
     while (not at_end_of_stream(Reader))
            L := read_line(Reader),
            println(Writer, L),
            flush(Writer)
     end,
     close(Reader),
     close(Writer).
```

The advantage of using the at_end_of_stream predicate instead of using the end_of_file atom is that at_end_of_stream immediately indicates that the end of the file was reached, even if the last read function read values into a list. In the first example in this chapter, which used the end_of_file atom, an extra read_line function was needed before the end of the file was detected. In the above example, which used at_end_of_stream, read_line was only called if there was data remaining to be read.

10.3 Writing to a File

The write and print predicates take a file descriptor as the first parameter. The file descriptor is the same descriptor that the open function returns. If the file descriptor is stdout, then the parameter can be omitted.

- write(FD, $Term$): This predicate writes $Term$ to a file. 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.
- write($Term$): This predicate is the same as write(stdout,$Term$).
- write_byte(FD, $Bytes$): This predicate writes a single byte or a list of bytes to a file.
- write_byte($Bytes$): This predicate is the same as write_byte(stdout,$Bytes$).
- write_char(FD, $Chars$): This predicate writes a single character or a list of characters to a file. The characters are not quoted. When writing a single-character atom $Char$, write_char(FD, $Char$) is the same as print(FD, $Char$), but write_char is faster than print.
- write_char($Chars$): This predicate is the same as write_char(stdout,$Chars$).
- write_char_code(FD, $Codes$): This predicate writes a single character or a list of characters of the given code or list of codes to a file.
- write_char_code($Codes$): This predicate is the same as the above, except that it writes to stdout.
- writeln(FD, $Term$): This predicate writes $Term$ and a newline, meaning that the next write will begin on the next line.
- writeln($Term$): This predicate is the same as writeln(stdout,$Term$).
- writef(FD, $Format$, $Args\ldots$): This predicate is used for formatted writing, where the $Format$ parameter contains format characters that indicate how to print each of the arguments in the $Args$ parameter. The number of arguments in $Args\ldots$ cannot exceed 10.

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

The writef predicate includes a parameter that specifies the string that is to be formatted. The $Format$ parameter is a string that contains format characters. Format characters take the form %[flags][width][.precision]specifier. Only the percent sign and the specifier are mandatory. *Flags* can be used for justification and padding. The *width* is the minimum number of characters that are to be printed. The *precision* is the number of characters that are to be printed after the number's radix point. Note that the width includes all characters, including the radix point and the characters that follow it. The *specifier* indicates the type of data that is to be written. A specifier can be one of the C format specifiers %%, %c,[1] %d, %e, %E, %f, %g, %G, %i, %o, %s, %u, %x, and %X. In addition, Picat uses the specifier %n for newlines, and uses %w for terms. For details, see Appendix F.

[1] The specifier %c can only be used to print ASCII characters. Use the specifier %w to print UTF-8 characters.

Example

```
formatted_print =>
    FD = open("birthday.txt",write),
    Format1 = "Hello, %s.  Happy birthday!  ",
    Format2 =  "You are %d years old today.  ",
    Format3 =  "That is %.2f%% older than you were last year",
    writef(FD, Format1, "Bob"),
    writef(FD, Format2, 7),
    writef(FD, Format3, 7.0 / 6.0),
    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 `print` predicates.

- `print(`FD`,` $Term$`)`: This predicate prints $Term$ to a file. Unlike the `write` predicates, the `print` predicates do not place quotes around strings and atoms.
- `print(`$Term$`)`: This predicate is the same as `print(stdout,`$Term$`)`.
- `println(`FD`,` $Term$`)` This predicate prints $Term$ and a newline.
- `println(`$Term$`)` This predicate is the same as `println(stdout,`$Term$`)`.
- `printf(`FD`,` $Format$`,` $Args\ldots$`)`: This predicate is the same as `writef`, except that `printf` uses `print` to display the arguments in the $Args$ parameter, while `writef` uses `write` to display the arguments in the $Args$ parameter.

The following example demonstrates the differences between the `write` and `print` predicates.

Example

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


10.4 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.

10.5 Standard File Descriptors

The atoms `stdin`, `stdout`, and `stderr` represent the file descriptors for standard input, standard output, and standard error. These atoms allow the program to use the input and output functions of the `io` module to read from and to write to the three standard streams.

Chapter 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`-port of the channel variable X.

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

- `post_event_ins(`X`)`: posts an event to the `ins`-port of the channel X.
- `post_event_bound(`X`)`: posts an event to the `bound`-port of the channel X.
- `post_event_dom(`X`,` T`)`: posts the term T to the `dom`-port of the channel X.
- `post_event_any(`X`,` T`)`: posts the event T to the `any`-port of the channel of X.

The call `post_event(`X`,`T`)` is the same as `post_event_dom(`X`,`T`)`. This means that the `dom`-port of a finite domain variable has two uses: posting exclusions of inner elements from the domain, and posting general term events.

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 be conjunctions of inline test predicates, such as type-checking built-ins (e.g., `integer(`X`)` and `var(`X`)`) and comparison built-ins (e.g., equality test X `==` Y, disequality test X `!==` Y, and arithmetic comparison $X > Y$). This restriction ensures that no variables in an actor can be changed while the condition is executed.

For an actor that is activated by an event, the system searches the definition sequentially from the top for an applicable rule. If no applicable rule is found, then the actor fails. If an applicable rule is found, the system executes the body of the rule. If the body fails, then the actor also fails. The body cannot succeed more than once. The system enforces this by converting $Body$ into `'once` $Body$`'` if $Body$ contains calls to nondeterministic predicates. If the applied rule is an action rule, then the actor is suspended after the body is executed, meaning that the actor is waiting to be activated again. If the applied rule is a normal non-backtrackable rule, then the actor vanishes after the body is executed. For each activation, only the first applicable rule is applied.

For a call and an action rule '$Head, Cond, \{Event\} => Body$', the call is registered as an actor if the call matches $Head$ and $Cond$ evaluates to `true`. The event pattern $Event$ implicitly specifies the ports to which the actor is attached, and the events that the actor watches. The following event patterns are allowed in $Event$:

- `event(`X`,`T`)`: This is the general event pattern. The actor is attached to the `dom`-ports of the variables in X. The actor will be activated by events posted to the `dom`-ports. T must be a variable that does not occur before `event(`X`,`T`)` in the rule.
- `ins(`X`)`: The actor is attached to the `ins`-ports of the variables in X. The actor will be activated when a variable in X is instantiated.
- `bound(`X`)`: The actor is attached to the `bound`-ports of the variables in X. The actor will be activated when the lower bound or upper bound of the domain of a variable in X changes.
- `dom(`X`)`: The actor is attached to the `dom`-ports of the variables in X. The actor will be activated when an inner value is excluded from the domain of a variable in X. The actor is not interested in what value is actually excluded.
- `dom(`X`,`E`)`: This is the same as `dom(`X`)`, except the actor is interested in the value E that is excluded. E must be a variable that does not occur before `dom(`X`,`E`)` in the rule.
- `dom_any(X)`: The actor is attached to the `any`-ports of the variables in X. The actor will be activated when an arbitrary value, including the lower bound value and the upper bound value, is excluded from the domain of a variable in X. The actor is not interested in what value is actually excluded.

- `dom_any(`X`,`E`)`: This is the same as `dom_any(`X`)`, except the actor is interested in the value E that is actually excluded. E must be a variable that does not occur before `dom_any(`X`,`E`)` in the rule.

In an action rule, multiple event patterns can be specified. After a call is registered as an actor on the channels, it will be suspended, waiting for events, unless the atom `generated` occurs in $Event$, in which case the actor will be suspended after $Body$ is executed.

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.

Example

Consider the following action rule:

```
p(X), {event(X,T)} => writeln(T).
```

The following gives a query and its output:

```
Picat> p(X), X.post_event(ping), X.post_event(pong)
ping
pong
```

The call `p(X)` is an actor. After `X.post_event(ping)`, the actor is activated and the body of the action rule is executed, giving the output `ping`. After `X.post_event(pong)`, the actor is activated again, outputting `pong`.

There is no primitive for killing actors or explicitly detaching actors from channels. As described above, an actor never disappears as long as action rules are applied to it. An actor vanishes only when a normal rule is applied to it. Consider the following example.

```
p(X,Flag),
    var(Flag),
    {event(X,T)}
=>
    writeln(T),
    Flag=1.
p(_,_) => true.
```

An actor defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. When a second event is posted, the action rule is no longer applicable, causing the second rule to be selected.

One question arises here: what happens if a second event is never posted to `X`? In this case, the actor will stay forever. If users want to immediately kill the actor after it is activated once, then users have to define it as follows:

```
p(X,Flag),
    var(Flag),
    {event(X,O),ins(Flag)},

=>
    write(O),
    Flag=1.
p(_,_) => true.
```


In this way, the actor will be activated again after `Flag` is bound to 1, and will be killed after the second rule is applied to it.

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:

```
import cp.

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

different_terms(X,Y,B),var(X),{ins(X),ins(Y)} => true.
different_terms(X,Y,B),var(Y),{ins(X)} => 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) =>
    writeln(X), writeln(Y),
    if (X.name !== Y.name; X.length !== Y.length) then
       B=1
    else
       Bs = new_list(X.length),
       foreach(I in 1 .. X.length)
           different_terms(X[I],Y[I],Bs[I])
       end,
       max(Bs) #= B
    end.
different_terms(X,Y,B),X==Y => B=0.
different_terms(X,Y,B) => B=1.
```

The call `different_terms(X,Y,B)` is delayed if either `X` or `Y` is a variable. The delayed call watches `ins(X)` and `ins(Y)` events. Once both `X` and `Y` become non-variable, the action rule becomes inapplicable, and one of the subsequent rules will be applied. If `X` and `Y` are lists, then they are different if the heads are different (`B1`), or if the tails are different (`B2`). This relationship is represented as the Boolean constraint `B #= (B1 #\/ B2)`. If `X` and `Y` are both structures,

then they are different if the functor is different, or if any pair of arguments of the structures is different.

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`:

```
import cp.

x_in_c_y_ac(X,Y,C),var(X),var(Y),
   {dom(Y,Ey)}
    =>
   fd_set_false(X,C-Ey).
x_in_c_y_ac(X,Y,C) => true.
```

Whenever an inner element `Ey` is excluded from the domain of `Y`, this propagator is triggered to exclude `C-Ey`, which is the support of `Ey`, from the domain of `X`. For the constraint `X+Y #= C`, users need to generate two propagators, namely, `x_in_c_y_ac(X,Y,C)` and `x_in_c_y_ac(Y,X,C)`, to maintain the arc consistency. Note that in addition to these two propagators, users also need to generate propagators for maintaining interval consistency, because `dom(Y,Ey)` only captures exclusions of inner elements, and does not capture bounds. The following propagator maintains interval consistency for the constraint:

```
import cp.

x_add_y_eq_c_ic(X,Y,C),var(X),var(Y),
    {generated,ins(X),ins(Y),bound(X),bound(Y)}
=>
    X :: C-Y.max .. C-Y.min,
    Y :: 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.

Chapter 12

Constraints

Picat provides three solver modules, including `cp`, `sat`, and `mip`, for modeling and solving constraint satisfaction and optimization problems (CSPs).[1] All three of these modules implement the same set of basic linear constraints. The `cp` and `sat` modules also implement non-linear and global 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. In order to make the symbols defined in a module available to the top level of the interpreter, users can use the built-in `load` to load the module or any module that imports the module. As the three modules have the same interface, this chapter describes the three modules together. Figure 12.1 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 both the `cp` and `sat` 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 12.1: Constraint operators in Picat

Precedence	Operators
Highest	`::`, `notin`, `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, `#>=`
	`#~`
	`#/\`
	`#^`
	`#\/`
	`#=>`
Lowest	`#<=>`

A constraint program normally poses a problem in three steps: (1) generate variables; (2) generate constraints over the variables; and (3) call `solve` to find a valuation for the variables that satisfies the constraints and possibly optimizes an objective function.

Example

This program in Figure 12.1 imports the `cp` module in order to solve the N-queens problem. The same program runs with the SAT solver if `sat` is imported, or runs with the LP/MIP solver if `mip` is imported. The predicate `Qs :: 1..N` declares the domains of the variables. The operator

[1] The `mip` module, which only supports linear constraints, is still experimental.

```
import cp.

queens(N) =>
    Qs=new_array(N),
    Qs :: 1..N,
    foreach (I in 1..N-1, J in I+1..N)
        Qs[I] #!= Qs[J],
        abs(Qs[I]-Qs[J]) #!= J-I
    end,
    solve(Qs),
    writeln(Qs).
```

Figure 12.1: A program for N-queens.

`#!=` 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)`.

12.1 Domain Variables

A domain variable is an attributed variable that has a domain attribute. The Boolean domain is treated as a special integer domain where 1 denotes `true` and 0 denotes `false`. Domain variables are declared with the built-in predicate $Vars$ `::` Exp.

- $Vars$ `::` Exp: This predicate restricts the domain or domains of $Vars$ to Exp. $Vars$ can be either a single variable, a list of variables, or an array of variables. For integer-domain variables, Exp must result in a list of integer values. For real-domain variables for the `mip` module, Exp must be an interval in the form $L..U$, where L and U are real values.

Domain variables, when being created, are usually represented internally by using intervals. An interval turns to a bit vector when a hole occurs in the interval. The following built-in predicate can be used to reset the range or access the current range.

- `fd_vector_min_max(`Min`,`Max`)`: When the arguments are integers, this predicate specifies the range of bit vectors; when the arguments are variables, this predicate binds them to the current bounds of the range. The default range is `-3200..3200`.

The following built-ins are provided for domain variables.

- $Vars$ `notin` Exp: This predicate excludes values Exp from the domain or domains of $Vars$, where $Vars$ and Exp are the same as in $Vars$ `::` Exp. This constraint cannot be applied to real-domain variables.
- `fd_degree(`$FDVar$`)` `=` $Degree$: This function returns the number of propagators that are attached to $FDVar$. This built-in is only provided by `cp`.
- `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. This built-in is only provided by `cp`.
- `fd_size(`$FDVar$`)` `=` $Size$: This function returns the size of the domain of $FDVar$, where $FDVar$ is an integer-domain variable.
- `fd_true(`$FDVar$`,` Elm`)`: This predicate is true if the integer Elm is an element in the domain of $FDVar$.
- `new_dvar()` `=` $FDVar$: This function creates a new domain variable with the default domain, which has the bounds `-72057594037927935..72057594037927935` on 64-bit computers and `-268435455..268435455` on 32-bit computers.

12.2 Table constraints

A *table constraint*, or an *extensional constraint*, over a tuple of variables specifies a set of tuples that are allowed (called *positive*) or disallowed (called *negative*) for the variables. A positive constraint takes the form `table_in(`$DVars$`,`R`)`, where $DVars$ is either a tuple of variables $\{X_1, \ldots, X_n\}$ or a list of tuples of variables, and R is a list of tuples of integers in which each tuple takes the form $\{a_1, \ldots, a_n\}$. A negative constraint takes the form `table_notin(`$DVars$`,`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 `ord(`$Char$`)` returns the code of $Char$, and that the function `chr(`$Code$`)` returns the character of $Code$.

```
import cp.

crossword(Vars) =>
    Vars=[X1,X2,X3,X4,X5,X6,X7],
Words2=[{ord('I'),ord('N')},
        {ord('I'),ord('F')},
        {ord('A'),ord('S')},
        {ord('G'),ord('O')},
        {ord('T'),ord('O')}],
Words3=[{ord('F'),ord('U'),ord('N')},
        {ord('T'),ord('A'),ord('D')},
        {ord('N'),ord('A'),ord('G')},
        {ord('S'),ord('A'),ord('G')}],
table_in([{X1,X2},{X1,X3},{X5,X7},{X6,X7}], Words2),
table_in([{X3,X4,X5},{X2,X4,X6}],Words3),
solve(Vars),
writeln([chr(Code) : Code in Vars]).
```

12.3 Arithmetic Constraints

An arithmetic constraint takes the form

$Exp1$ Rel $Exp2$

where $Exp1$ and $Exp2$ are arithmetic expressions, and Rel is one of the constraint operators: `#=`, `#!=`, `#<`, `#=<`, `#<=`, `#>`, or `#>=`. The operators `#=<` and `#<=` are the same, meaning less than or equal to. An arithmetic expression is made from integers, variables, arithmetic functions, and constraints. The following arithmetic functions are allowed: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (truncated integer division), `//` (truncated integer division), `div` (floored integer division), `mod`, `**` (power), `abs`, `min`, `max`, and `sum`. Except for index notations, array comprehensions and list comprehensions, which are interpreted as function calls as in normal expressions, expressions in arithmetic constraints are treated as terms, and it is unnecessary to enclose them with dollar-signs. In addition to the numeric operators, the following functions are allowed in constraints:

- `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 of domain variables.
- `max(`$DVars$`)`: The maximum of $DVars$, where $DVars$ is a list 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 of domain variables.
- `prod(`$DVars$`)`: The product of $DVars$, where $DVars$ is a list of domain variables.

When a constraint occurs in an arithmetic expression, it is evaluated to 1 if it is satisfied and 0 if it is not satisfied.

Example

```
import mip.

go =>
    M={{0,3,2,3,0,0,0,0},
       {0,0,0,0,0,0,5,0},
       {0,1,0,0,0,1,0,0},
       {0,0,2,0,2,0,0,0},
       {0,0,0,0,0,0,0,5},
       {0,4,0,0,2,0,0,1},
       {0,0,0,0,0,2,0,3},
       {0,0,0,0,0,0,0,0}},
    maxflow(M,1,8).

maxflow(M,Source,Sink) =>
    N=M.length,
    X=new_array(N,N),
    foreach(I in 1..N, J in 1..N)
        X[I,J] :: 0..M[I,J]
    end,
    foreach(I in 1..N, I!=Source, I!=Sink)
        sum([X[J,I] : J in 1..N]) #= sum([X[I,J] : J in 1..N])
    end,
    Total #= sum([X[Source,I] : I in 1..N]),
    Total #= sum([X[I,Sink] : I in 1..N]),
    solve([$max(Total)],X),
    writeln(Total),
    writeln(X).
```

This program uses MIP to solve the maximum integer flow problem. Given the capacity matrix `M` of a directed graph, the start vertex `Source`, and the destination vertex `Sink`, the predicate `maxflow(M,Source,Sink)` finds a maximum flow from `Source` to `Sink` over the graph. When two vertices are not connected by an arc, the capacity is given as 0. The first foreach loop specifies the domains of the variables. For each variable `X[I,J]`, the domain is restricted to integers between 0 and the capacity, `M[I,J]`. If the capacity is 0, then the variable is immediately instantiated to 0. The next foreach loop posts the conservation constraints. For each vertex `I`, if it is neither the source nor the sink, then its total incoming flow amount

```
sum([X[J,I] : J in 1..N])
```

is equal to the total outgoing flow amount

```
sum([X[I,J] : J in 1..N]).
```

The total flow amount is the total outgoing amount from the source, which is the same as the total incoming amount to the sink.

12.4 Boolean Constraints

A Boolean constraint takes one of the following forms:

`#~` $BoolExp$
$BoolExp$ `#/\` $BoolExp$
$BoolExp$ `#^` $BoolExp$
$BoolExp$ `#\/` $BoolExp$
$BoolExp$ `#=>` $BoolExp$
$BoolExp$ `#<=>` $BoolExp$

$BoolExp$ is either a Boolean constant (0 or 1), a Boolean variable (an integer-domain variable with the domain [0,1]), an arithmetic constraint, a domain constraint (in the form of Var `::` $Domain$ or Var `notin` $Domain$), or a Boolean constraint. As shown in Table 12.1, 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.

12.5 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. If the `sat` module is used, then global constraints are translated into smaller constraints before being translated further into conjunctive normal form.

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.
- `all_different_except_0(`$FDVars$`)`: This constraint is true if all non-zero values in $FDVars$ are different.
- `assignment(`$FDVars1$`,` $FDVars2$`)`: This constraint ensures that $FDVars2$ is a *dual assignment* of $FDVars1$, i.e., if the ith element of $FDVars1$ is j, then the jth element of $FDVars2$ is i. The constraint can be defined as:

```
assignment(Xs,Ys) =>
    N = Xs.length,
    (var(Ys) -> Ys = new_list(N); true),
    Xs :: 1..N,
    Ys :: 1..N,
    foreach(I in 1..N, J in 1..N)
        X[I] #= J #<=> Y[J] #= I
    end.
```

- `at_least(`N`,`L`,`V`)`: This constraint succeeds if there are at least N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.
- `at_most(`N`,`L`,`V`)`: This constraint succeeds if there are at most N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.
- `circuit(`$FDVars$`)`: Let $FDVars$ be a list of variables $[X_1, X_2, \ldots, X_N]$, where each X_i has the domain $1..N$. A valuation $X_1 = v_1$, $X_2 = v_2$, ..., $X_n = v_n$ satisfies the constraint if `1->`v_1`, 2->`v_2`, ..., n->`v_n forms a Hamiltonian cycle. This constraint ensures that each variable has a different value, and that the graph that is formed by the assignment does not contain any sub-cycles. For example, for the constraint

```
circuit([X1,X2,X3,X4])
```

`[3,4,2,1]` is a solution, but `[2,1,4,3]` is not, because the graph `1->2, 2->1, 3->4, 4->3` contains two sub-cycles.

- `count(`V`,` $FDVars$`,` Rel`,` N`)`: In this constraint, V and N are integer-domain variables, $FDVars$ is a list of integer-domain variables, and Rel is an arithmetic constraint operator (`#=`, `#!=`, `#>`, `#>=`, `#<`, `#=<`, or `#<=`). Let $Count$ be the number of elements in $FDVars$ that are equal to V. The constraint is true iff $Count$ Rel N is true. This constraint can be defined as follows:

```
count(V,L,Rel,N) =>
   sum([V #= E : E in L]) #= Count,
   call(Rel,Count,N).
```

- `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*; however the constraint may be more efficient, because it converts the disjunctive tasks into global constraints. This constraint is only accepted by `cp`.

- `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.

- `exactly(`N`,`L`,`V`)`:: This constraint succeeds if there are exactly N elements in L that are equal to V, where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

- `global_cardinality(`*List*`,` *Pairs*`)`: Let *List* be a list of integer-domain variables `[`X_1`,` ...`,` X_d`]`, and *Pairs* be a list of pairs `[`K_1`-`V_1`,` ...`,` K_n`-`V_n`]`, where each key K_i is a unique integer, and each V_i is an integer-domain variable. The constraint is true if every element of *List* is equal to some key, and, for each pair K_i–V_i, exactly V_i elements of *List* are equal to K_i. This constraint can be defined as follows:

```
global_cardinality(List,Pairs) =>
    foreach($Key-V in Pairs)
        sum([B : E in List, B#<=>(E#=Key)]) #= V
    end.
```

- `lex_le(`L_1`,`L_2`)`: The sequence (an array or a list) L_1 is lexicographically less than or equal to L_2.

- `lex_lt(`L_1`,`L_2`)`: The sequence (an array or a list) L_1 is lexicographically less than L_2.

- `matrix_element(`*Matrix*`,`I`,`J`,`V`)`: This constraint is true if the entry at <I,J> in *Matrix* is V, where I, J, and V are integer-domain variables, and *Matrix* is an two-dimensional array of integer-domain variables.

- `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. This constraint is only accepted by `cp`.

- `nvalue(`N`,` *List*`)`: The number of distinct values in *List* is N, where *List* is a list of integer-domain variables.

- `regular(`$L, Q, S, M, Q0, F$`)`: Given a finite automaton (DFA or NFA) of Q states numbered 1, 2, ..., Q with input 1..S, transition matrix M, initial state $Q0$ ($1 \leq Q0 \leq Q$), and a list of accepting states F, this constraint is true if the list L is accepted by the automaton. The transition matrix M represents a mapping from $1..Q \times 1..S$ to $0..Q$, where 0 denotes the *error* state. For a DFA, every entry in M is an integer, and for an NFA, entries can be a list of integers.

- `scalar_product(`A`,` X`,` *Product*`)`: The scalar product of A and X is *Product*, where A and X are lists or arrays of integer-domain variables, and *Product* is an integer-domain variable. A and X must have the same length.

- `scalar_product(`A`,` X`,` *Rel*`,` *Product*`)`: The scalar product of A and X has the relation *Rel* with *Product*, where *Rel* is one of the following operators: `#=`, `#!=`, `#>=`, `#>`, `#=<` (`#<=`), and `#<`.

- `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. This constraint is only accepted by `cp` and `sat`.

12.6 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` and `sat`, this predicate can be called multiple times for a problem, and each call can backtrack in order to find multiple solutions. However, for `mip`, this predicate can be called only once for a problem, and this call can backtrack in order to find multiple solutions.

- `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*`])`.

- `solve_all(`*Options*`,` *Vars*`)` `=` *Solutions*: This function returns all the solutions that satisfy the constraints.

- `solve_all(`*Vars*`)` `=` *Solutions*: This function is the same as `solve_all([],`*Vars*`)`.

- `indomain_down(`*Var*`)`: This predicate is the same as `solve([down],` `[`*Var*`])`. It is only accepted by `cp`.

12.6.1 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.
- `$report(`$Call$`)`: Execute $Call$ each time a better answer is found while searching for an optimal answer. This option cannot be used if the `mip` module is used.

12.6.2 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]`.
- `label(`$CallName$`)`: This option informs the CP solver that once a variable V is selected, the user-defined call $CallName$`(`V`)` is used to label V, where $CallName$ must be defined in the same module, an imported module, or the global module.
- `leftmost`: The same as `forward`.
- `max`: First, select a variable whose domain has the largest upper bound, breaking ties by selecting a variable with the smallest domain.
- `min`: First, select a variable whose domain has the smallest lower bound, breaking ties by selecting a variable with the smallest domain.
- `rand`: Both variables and values are randomly selected when labeling.
- `rand_var`: Variables are randomly selected when labeling.
- `rand_val`: Values are randomly selected when labeling.
- `reverse_split`: Bisect the variable's domain, excluding the lower half first.
- `split`: Bisect the variable's domain, excluding the upper half first.
- `updown`: Values are assigned to variables from the values that are nearest to the middle of the domain.

12.6.3 Solving Options for `sat`

- `dump`: dumps the CNF code to `stdout`.
- `dump(`$File$`)`: dumps the CNF code to $File$.
- `$nvars(`$NVars$`)`: the number of variables in the CNF code is $NVars$.
- `$ncls(`$NCls$`)`: the number of clauses in the CNF code is $NCls$.

12.6.4 Solving Options for `mip`

- `dump`: dumps the constraints in CPLEX format to `stdout`.

Chapter 13

The `os` Module

Picat has an `os` module for manipulating files and directories. In order to use any of the functions or predicates, users must import the module.

13.1 The *Path* Parameter

Many of the functions and predicates in this module have a *Path* parameter. This parameter is a string or an atom, representing the path of a file or directory. This path can be an absolute path, from the system's root directory, or a 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*

13.2 Directories

The `os` module includes functions for reading and modifying directories. The following example shows how to list all of the files in a directory tree, using a depth-first directory traversal.

Example

```
import os.

traverse(Dir), directory(Dir) =>
   List = listdir(Dir),
   printf("Inside %s%n",Dir),
   foreach(File in List)
       printf("    %s%n",File)
   end,
   foreach (File in List, File != ".", File != "..")
       FullName = Dir ++ [separator()] ++ File,
       traverse(FullName)
   end.
traverse(_Dir) => true.
```

The following function can be used to read the contents of a directory:

- `listdir(`*Path*`) =` *List*: This function returns a list of all of the files and directories that are contained inside the directory specified by *Path*. If *Path* is not a directory, then an error is thrown. The returned list contains strings, each of which is the name of a file or directory.

The above example also uses the `directory` predicate, which will be discussed in Section 13.4.

13.2.1 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*.

13.3 Modifying Files and Directories

13.3.1 Creation

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

- `mkdir(`*Path*`)`: This predicate creates a new directory at location *Path*. The directory will be created with a default permission list of `[rwu, rwg, ro]`. If the program does not have permission to write to the parent directory of *Path*, this predicate will throw an error. An error will also occur if the parent directory does not exist.
- `rename(`*Old*`,` *New*`)`: This renames a file or a directory from *Old* to *New*. This predicate will throw an error if *Old* does not exist. An error will also occur if the program does not have permission to write to *Old* or *New*.
- `cp(`*FromPath*`,` *ToPath*`)`: This copies a file from *FromPath* to *ToPath*. This predicate will throw an error if *FromPath* does not exist or *FromPath* is a directory. An error will also occur if the program does not have permission to read from *FromPath*, or if it does not have permission to write to *ToPath*.

13.3.2 Deletion

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

- `rm(`*Path*`)`: This deletes a file. An error will be thrown if the file does not exist, if the program does not have permission to delete the file, or if *Path* refers to a directory, a hard link, a symbolic link, or a special file type.
- `rmdir(`*Path*`)`: This deletes a directory. An error will be thrown if the directory does not exist, the program does not have permission to delete the directory, the directory is not empty, or if *Path* does not refer to a directory.

13.4 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 $Path$.

- `readable(`$Path$`)`: Is the program allowed to read from the file?
- `writable(`$Path$`)`: Is the program allowed to write to the file?
- `executable(`$Path$`)`: Is the program allowed to execute the file?
- `size(`$Path$`) =` Int: If $Path$ is not a symbolic link, then this function returns the number of bytes contained in the file to which $Path$ refers. If $Path$ is a symbolic link, then this function returns the path size of the symbolic link. Because the function `size/1` is defined in the `basic` module for returning the size of a map, this function requires an explicit module qualifier `os.size(`$Path$`)`.
- `file_base_name(`$Path$`) =` $List$: 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/`".
- `exists(`$Path$`)`: Is $Path$ an existing file or directory?
- `file(`$Path$`)`: Does $Path$ refer to a regular file? This predicate is true if $Path$ is neither a directory nor a special file, such as a socket or a pipe.
- `file_exists(`$Path$`)`: This tests whether $Path$ exists, and, if it exists, whether $Path$ refers to a regular file.
- `directory(`$Path$`)`: Does $Path$ refer to a directory?

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

Example

```
import os.

test_file(Path) =>
    if (not exists(Path)) then
        printf("%s does not exist %n",Path)
    elseif (directory(Path)) then
        println("Directory")
    elseif (file(Path)) then
        println("File")
    else
        println("Unknown")
    end.
```


13.5 Environment Variables

- `env_exists(`$Name$`)`: This predicate succeeds if $Name$ is an environment variable in the system.
- `getenv(`$Name$`) =` $String$: This function returns the value of the environment variable $Name$ as a string. This function will throw an error if the environment variable $Name$ does not exist.

Appendix A

The `math` Module

Picat provides a `math` module, which has common mathematical constants and functions. The `math` module is imported by default.

A.1 Constants

The `math` module provides two constants.

- `e = 2.71828182845904523536`
- `pi = 3.14159265358979323846`

A.2 Functions

The `math` module contains mathematical functions that serve a number of different purposes. Note that the arguments must all be numbers. If the arguments are not numbers, then Picat will throw an error.

A.2.1 Sign and Absolute Value

The following functions deal with the positivity and negativity of numbers.

- `sign(`X`) = `Val: This function 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`) = (`$IntVal$`,`$FractVal$`)`: This function splits a real number into its integer part and its fractional part.

Example

```
Picat> Val1 = ceiling(-3.2), Val2 = ceiling(3)
Val1 = -3
Val2 = 3
Picat> Val1 = floor(-3.2), Val2 = floor(3)
Val1 = -4
Val2 = 3
Picat> Val1 = round(-3.2), Val2 = round(-3.5), Val3 = round(3.5)
Val1 = -3
Val2 = -4
Val3 = 4
Picat> Val1 = truncate(-3.2), Val2 = truncate(3)
Val1 = -3
Val2 = 3
Picat> IF = modf(3.2)
IF = (3.0 , 0.2)
```

A.2.3 Exponents, Roots, and Logarithms

The following functions provide exponentiation, root, and logarithmic functions. Note that, in the logarithmic functions, if $X \leq 0$, then an error is thrown.

- `pow(`X`, `Y`) = `Val: This function returns X^Y. It does the same thing as $X**Y$.
- `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.
- `log(`X`) = `Val: This function returns $log_e(X)$.
- `log10(`X`) = `Val: This function returns $log_{10}(X)$.
- `log2(`X`) = `Val: This function returns $log_2(X)$.
- `log(`B`, `X`) = `Val: This function returns $log_B(X)$.

Example

```
Picat> P1 = pow(2, 5), P2 = exp(2)
P1 = 32
P2 = 7.38906
Picat> S = sqrt(1)
S = 1.0
Picat> E = log(7), T = log10(7), T2 = log2(7), B = log(7, 7)
E = 1.94591
T = 0.845098
T2 = 2.80735
B = 1.0
```

A.2.4 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.
- `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`].

Example

```
Picat> S = sin(pi), C = cos(pi), T = tan(pi)
S = 0.0
C = -1.0
T = 0.0
Picat> S = asin(0), C = acos(0), T = atan(0), T2 = atan2(-10, 10)
S = 0.0
C = 1.5708
T = 0.0
T2 = -0.785398
```


A.2.5 Random Numbers

The following functions provide access to a random number generator.

- `random() =` Val: This function returns a random integer.
- `random2() =` Val: This function returns a random integer, using an environment-dependent seed.
- `rand_max() =` Val: This function returns the maximum random integer.
- `random(`$Seed$`) =` Val: This function returns a random integer. At the same time, it changes the seed of the random number generator.
- `frand() =` Val: This function returns a random real number between 0.0 and 1.0.

A.2.6 Other Built-ins

- `even(`N`)`: This predicate is true if N is an even integer.
- `gcd(`A`,`B`)`: This function returns the greatest common divisor of integer A and integer B.
- `odd(`N`)`: This predicate is true if N is an odd integer.
- `prime(`N`)`: This predicate is true if N is a prime number.
- `primes(`N`) =` $List$: This function returns a list of prime numbers that are less than or equal to N.

Appendix B

The `sys` Module

The `sys` module, which is imported by default, contains built-ins that are relevant to the Picat system. The built-ins in the `sys` module perform operations that include compiling programs, tracing execution, and displaying statistics and information about the Picat system.

B.1 Compiling and Loading Programs

The `sys` module includes a number of built-ins for compiling programs and loading them into memory.

- `compile(`$FileName$`)`: This predicate compiles the file $FileName$`.pi` and all of its dependent files without loading the generated 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. If $FileName$`.pi` imports modules, then these module files are also compiled. The system searches for these module files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`.

- `compile_bp(`$FileName$`)`: This predicate translates the Picat file $FileName$`.pi` into a B-Prolog file $FileName$`.pl`. If the file is dependent on other Picat files, then those files are compiled using `compile/1`. The destination directory for the B-Prolog file is the same as the source file's directory. If the Picat interpreter does not have permission to write into the directory in which a source file resides, then this built-in throws an exception.

- `load(`$FileName$`)`: This predicate loads the byte-code file $FileName$`.qi` and all of its dependent byte-code files into the system for execution. For $FileName$, the system searches for a byte-code file in the directory specified by $FileName$ or the directories that are stored in the environment variable `PICATPATH`. For the dependent file names, the system searches for a byte-code file in the directory in which $FileName$`.qi` resides or the directories that are stored in the environment variable `PICATPATH`. If the byte-code file $FileName$`.qi` does not exist, but the source file $FileName$`.pi` exists, then this built-in compiles the source file and loads the byte codes without creating a `qi` file. Note that, for the dependent files, if the byte-code file does not exist, but the source file exists, then the source file will be compiled.

- `cl(`$FileName$`)`: This predicate compiles and loads the source file named $FileName$`.pi`. Note that the extension `.pi` does not need to be given. The system also compiles and loads all of the module files that are either directly imported or indirectly imported by the source file. The system searches for such dependent files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`.

- `cl`: This predicate compiles and loads a program from the console, ending when the end-of-file character (`ctrl-z` for Windows and `ctrl-d` for Unix) is typed.

- `cl_facts(`$Facts$`)`: This predicate compiles and loads facts into the system. The argument $Facts$ is a list of ground facts.

- `cl_facts(`$Facts$`, `$IndexInfo$`)`: This predicate compiles and loads facts into the system. The argument $Facts$ is a list of ground facts. The argument $IndexInfo$ is a list of indexing information in the form `p(`M_1`, `M_2`, ..., `M_n`)`. Each M_i can either be `+`, which indicates that the argument is input, or `-`, which indicates that the argument is output.

B.2 Tracing Execution

The Picat system has three execution modes: *non-trace mode*, *trace mode*, and *spy mode*. In trace mode, it is possible to trace the execution of a program, showing every call in every possible stage. In order to trace the execution, the program must be recompiled while the system is in trace mode. In spy mode, it is possible to trace the execution of individual functions and predicates. The following predicates are used to switch between non-trace mode and trace mode.

- `trace`: This predicate switches the execution mode to trace mode.

- `notrace`: This predicate switches the execution mode to non-trace mode.

- `debug`: This predicate switches the execution mode to trace mode.

- `nodebug`: This predicate switches the execution mode to non-trace mode.

- `spy(`$Point$`)`: This predicate places a spy point on $Point$, which is a function or a predicate, optionally followed by an arity. The creation of a spy point switches the Picat system to spy mode.

- `nospy`: This predicate removes all spy points, and switches the execution mode to non-trace mode.

- `abort`: This predicate terminates the current program. This can be used in all three execution modes.

B.2.1 Debugging Commands

In trace mode, the system displays a message when a function or a predicate is entered (`Call`), exited (`Exit`), re-entered (`Redo`) or has failed (`Fail`). After a function or a predicate is entered or re-entered, the system waits for a command from the user. A command is a single letter followed by a carriage-return, or may simply be a carriage-return. The following commands are available:

- `+` : create a spy point.

- `-` : remove a spy point.

- `<` : reset the print depth to 10.

- `< i` : reset the print depth to `i`.

- `a` : abort, quit debugging, moving control to the top level.
- `<cr>` : A carriage return causes the system to show the next call trace.
- `c` : creep, show the next call trace.
- `h` : help, display the debugging commands.
- `?` : help, display the debugging commands.
- `l` : leap, be silent until a spy point is encountered.
- `n` : nodebug, prevent the system from displaying debugging messages for the remainder of the program.
- `r` : repeat, continue to creep or leap without intervention.
- `s` : skip, be silent until the call is completed (`Exit` or `Fail`).
- `t` : backtrace, show the backtrace leading to the current call.
- `t i` : backtrace, show the backtrace from the call numbered `i` to the current call.
- `u` : undo what has been done to the current call and redo it.
- `u i` : undo what has been done to the call numbered `i` and redo it.

B.3 Information about the Picat System

The `sys` module contains a number of built-ins that display information about the Picat system. This information includes statistics about the system, including the memory that is used, and the amount of time that it takes to perform a goal.

B.3.1 Statistics

The following built-ins display statistics about the memory that Picat system uses.

- `statistics`: This predicate displays the number of bytes that are allocated to each data area, and the number of bytes that are already in use.
- `statistics(`Key`,` $Value$`)`: The statistics concerning Key are $Value$. This predicate gives multiple solutions upon backtracking. Keys include `runtime`, `program`, `heap`, `control`, `trail`, `table`, `gc`, `backtracks`, and `gc_time`. The values for most of the keys are lists of two elements. For the key `runtime`, the first element denotes the amount of time in milliseconds that has elapsed since Picat started, and the second element denotes the amount of time that has elapsed since the previous call to `statistics/2` was executed. For the key `gc`, the number indicates the number of times that the garbage collector has been invoked. For the key `backtracks`, the number indicates the number of backtracks that have been done during the labeling of finite domain variables since Picat was started. For all other keys, the first element denotes the size of memory in use, and the second element denotes the size of memory that is still available in the corresponding data area.
- `statistics_all() =` $List$: This function returns a list of lists that are in the form `[Key, Value]`. The list contains all of the keys that `statistics/2` can display, together with their corresponding values.

Example

```
Picat> statistics
       Stack+Heap:    8,000,000 bytes
         Stack in use:    1,156 bytes
         Heap in use:    28,592 bytes

       Program:       8,000,000 bytes
         In use:      1,448,436 bytes
         Symbols:         5,300

       Trail:         4,000,000 bytes
         In use:            936 bytes

       Memory manager:
          GC:         Call(0), Time(0 ms)
          Expansions: Stack+Heap(0), Program(0), Trail(0), Table(0)

Picat> statistics(Key, Value)
       Key = runtime
       Value = [359947,66060]?;

       Key = program
       Value = [1451656,6548344]?;

       Key = heap
       Value = [34112,7964524]?;

       Key = control
       Value = [1360,7964524]?;

       Key = trail
       Value = [1496,3998504]?;

       Key = table
       Value = [0,4000000]?;

       Key = table_blocks
       Value = 1?;

       key = gc
       Value = 0?;

       Key = backtracks
       V = 0 ?;

       Key = gc_time
       Value = 0

Picat> L = statistics_all()
```

```
L = [[runtime, [359947,66060]], [program, [1451656,6548344]],
     [heap, [34112,7964524]], [control, [1360,7964524]],
     [trail, [1496,3998504]], [table, [0,4000000]],
     [table_blocks,1],[gc, 0], [backtracks, 0], [gc_time, 0]]
```

B.3.2 Time

The following predicates display the amount of CPU time that it takes to perform a goal.

- `time(`*Goal*`)`: This predicate calls *Goal*, and reports the number of seconds of CPU time that were consumed by the execution.
- `time2(`*Goal*`)`: This predicate calls *Goal*, and reports the number of seconds of CPU time that were consumed by the execution, and the number of backtracks that have been performed in labeling finite-domain variables during the execution of *Goal*.
- `time_out(`*Goal*`,` *Limit*`,` *Result*`)`: This predicate is logically equivalent to `once` *Goal*, except that is imposes a time limit, in milliseconds, on the evaluation. If *Goal* is not finished when *Limit* expires, then the evaluation will be aborted, and *Result* will be unified with the atom `time_out`. If *Goal* succeeds within the time limit, then *Result* will be unified with the atom `success`. Note that time-out may be delayed or never occur because of the execution of an external C function.

B.3.3 Other System Information

- `help`: This predicate displays the usages of some of the commands that the system accepts.
- `loaded_modules() =` *List*: This function returns a list of the modules that are currently loaded in the Picat system. This list includes library modules and user-defined modules. By default, this function returns `[basic,sys,io,math]`.
- `nolog`: This predicate turns off the logging flag.
- `picat_path() =` *Path*: This function returns the directories that are stored in the environment variable `PICATPATH`. If the environment variable `PICATPATH` does not exist, then this function throws an error.
- `command(`*String*`) =` *Int*: This function sends the command *String* to the OS and returns the status that is returned from the OS.

B.4 Garbage Collection

Picat incorporates an incremental garbage collector for the control stack and the heap. The garbage collector is active by default. The `sys` module includes the following predicates for garbage collection.

- `garbage_collect`: This predicate starts the garbage collector.
- `garbage_collect(`*Size*`)`: This predicate calls the garbage collector. If there are less than *Size* words on the control stack and heap after garbage collection, then it invokes the memory manager to expand the stack and heap so that there are *Size* words on the control stack and heap.

B.5 Quitting the Picat System

The following predicates can be used to terminate the Picat interpreter.

- `exit`
- `halt`

Appendix C

The `util` Module

The `util` module provides general useful utility functions and predicates. This module is expected to be expanded in the future. This module must be imported before use.

C.1 Utilities on Terms

- `replace(`$Term$`,`Old`,`New`)` = $NewTerm$: This function returns a copy of $Term$, replacing all of the occurrences of Old in $Term$ by New.
- `replace_at(`$Term$`,`$Index$`,`New`)` = $NewTerm$: This function returns a copy of $Term$, replacing the argument at $Index$ by New. $Term$ must be a compound term.
- `find_first_of(`$Term$`,`$Pattern$`)` = $Index$: This function returns the first index at which the argument unifies with $Pattern$. If there is no argument that unifies with $Pattern$, then this function returns -1. $Term$ must be either a list or a structure.
- `find_last_of(`$Term$`,`$Pattern$`)` = $Index$: This function returns the last index at which the argument unifies with $Pattern$. If there is no argument that unifies with $Pattern$, then this function returns -1. $Term$ must be either a list or a structure.

C.2 Utilities on Strings and Lists

- `find(`$String$`,`$SubString$`,`$From$`,`To`)`: This predicate searches for an occurrence of $SubString$ in $String$, and binds $From$ to the starting index and To to the ending index. On backtracking, this predicate searches for the next occurrence of $SubString$.
- `find_ignore_case(`$String$`,`$SubString$`,`$From$`,`To`)`: This predicate is the same as `find(`$String$`,`$SubString$`,`$From$`,`To`)`, except that it is case insensitive.
- `split(`$String$`,`$Separators$`)` = $Tokens$: This function splits $String$ into a list of tokens, using characters in the string $Separators$ as split characters. Recall that a string is a list of characters. A token is a string, so the return value is a list of lists of characters.
- `split(`$String$`)` = $Tokens$: This function is the same as `split(`$String$`," \t\n\r")`, which uses white spaces as split characters.
- `join(`$Tokens$`,`$Separator$`)` = $String$: This function concatenates the tokens $Tokens$ into a string, adding $Separator$, which is a string or an atom, between each two tokens.
- `join(`$Tokens$`)` = $String$: This function is the same as `join(`$Tokens$`," " )`.

- `strip(`$List$`,`$Elms$`)` = $List$: This function returns a copy of $List$ with leading and trailing elements in $Elms$ removed.
- `strip(`$List$`)` = $List$: This function is the same as `strip(`$List$`," \t\n\r")`.
- `lstrip(`$List$`,`$Elms$`)` = $List$: This function returns a copy of $List$ with leading elements in $Elms$ removed.
- `lstrip(`$List$`)` = $List$: This function is the same as `lstrip(`$List$`," \t\n\r")`.
- `rstrip(`$List$`,`$Elms$`)` = $List$: This function returns a copy of $List$ with trailing elements in $Elms$ removed.
- `rstrip(`$List$`)` = $List$: This function is the same as `rstrip(`$List$`," \t\n\r")`.

C.3 Utilities on Matrices

An *array matrix* is a two-dimensional array. The first dimension gives the number of the rows and the second dimension gives the number of the columns. A *list matrix* represents a matrix as a list of lists.

- `matrix_multi(`A`,`B`)`: This function returns the product of matrices A and B. Both A and B must be array matrices.
- `transpose(`A`)` = B: This function returns the transpose of the matrix A. A can be an array matrix or a list matrix. If A is an array matrix, then the returned transpose is also an array. If A is a list matrix, then the returned transpose is also a list.
- `rows(`A`)` = $List$: This function returns the rows of the matrix A as a list.
- `columns(`A`)` = $List$: This function returns the columns of the matrix A as a list.
- `diagonal1(`A`)` = $List$: This function returns primary diagonal of the matrix A as a list.
- `diagonal2(`A`)` = $List$: This function returns secondary diagonal of the matrix A as a list.
- `array_matrix_to_list_matrix(`A`)` = $List$: This function converts the array matrix A to a list matrix.
- `array_matrix_to_list(`A`)` = $List$: This function converts the array matrix A to a flat list, row by row.

C.4 Utilities on Lists and Sets

- `permutation(`$List$`,`P`)`: This predicate generates a permutation P of $List$. This predicate is non-deterministic. On backtracking, it generates the next permutation.
- `permutations(`$List$`)` = Ps: This function returns all the permutations of $List$.
- `nextto(`$E1$`,`$E2$`,`$List$`)`: This predicate is true if $E1$ follows $E2$ in $List$. This predicate is non-deterministic.

Appendix D

The `ordset` Module

An ordered set is represented as a sorted list that does not contain duplicates. The `ordset` module provides useful utility functions and predicates on ordered sets. This module must be imported before use.

- `delete`($OSet$`,`Elm) = $OSet_1$: This function returns of a copy of $OSet$ that does not contain the element Elm.
- `disjoint`($OSet_1$`,`$OSet_2$): This predicate is true when $OSet_1$ and $OSet_2$ have no element in common.
- `insert`($OSet$`,`Elm) = $OSet_1$: This function returns a copy of $OSet$ with the element Elm inserted.
- `intersection`($OSet_1$`,`$OSet_2$)=$OSet_3$: This function returns an ordered set that contains elements which are in both $OSet_1$ and $OSet_2$.
- `new_ordset`($List$) = $OSet$: This function returns an ordered set that contains the elements of $List$.
- `ordset`($Term$): This predicate is true if $Term$ is an ordered set.
- `subset`($OSet_1$`,`$OSet_2$): This predicate is true if $OSet_1$ is a subset of $OSet_2$.
- `subtract`($OSet_1$`,`$OSet_2$)=$OSet_3$: This function returns an ordered set that contains all of the elements of $OSet_1$ which are not in $OSet_2$.
- `union`($OSet_1$`,`$OSet_2$)=$OSet_3$: This function returns an ordered set that contains all of the elements which are present in either $OSet_1$ or $OSet_2$.

Appendix E

The `datetime` Module

Picat's `datetime` module provides built-ins for retrieving the date and time. This module must be imported before use.

- `current_datetime()` = $DateTime$: This function returns the current date and time as a structure in the form

  `date_time`($Year$, $Month$, Day, $Hour$, $Minute$, $Second$)

  where the arguments are all integers, and have the following meanings and ranges.

Argument	**Meaning**	**Range**
$Year$	years since 1900	an integer
$Month$	months since January	0-11
Day	day of the month	1-31
$Hour$	hours since midnight	0-23
$Minute$	minutes after the hour	0-59
$Second$	seconds after the minute	0-60

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

- `current_date()` = $Date$: This function returns the current date as a structure in the form `date`($Year$, $Month$, Day), where the arguments have the meanings and ranges that are defined above.
- `current_day()` = $WDay$: This function returns the number of days since Sunday, in the range 0 to 6.
- `current_time()` = $Time$: This function returns the current time as a structure in the form `time`($Hour$, $Minute$, $Second$), where the arguments have the meanings and ranges that are defined above.

Appendix F

Formats

F.1 Formatted Printing

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

Specifier	Output
`%%`	Percent Sign
`%c`	Character
`%d`	Signed Decimal Integer
`%e`	Scientific Notation, with Lowercase `e`
`%E`	Scientific Notation, with Uppercase `E`
`%f`	Decimal Real Number
`%g`	Shorter of `%e` and `%f`
`%G`	Shorter of `%E` and `%f`
`%i`	Signed Decimal Integer
`%n`	Platform-independent Newline
`%o`	Unsigned Octal Integer
`%s`	String
`%u`	Unsigned Decimal Integer
`%w`	Term
`%x`	Unsigned Lowercase Hexadecimal Integer
`%X`	Unsigned Uppercase Hexadecimal Integer

Appendix G

External Language Interface with C

Picat has an interface with C, through which Picat programs can call deterministic predicates that are written as functions in C. C programs that use this interface must include the file `"picat.h"` in the directory `Picat/Emulator`. In order to make C-defined predicates available to Picat, users have to re-compile Picat's C source code together with the newly-added C functions.

G.1 Term Representation

Picat's C interface provides functions for accessing, manipulating, and building Picat terms. In order to understand these functions, users need to know how terms are represented in Picat's virtual machine.

A term is represented by a word that contains a value and a tag. A word has 32 bits or 64 bits, depending on the underlying CPU and OS. The tag in a word distinguishes the type of the term.

The value of a term is an address, except when the term is an integer (in which case, the value represents the integer itself). The location to which the address points is dependent on the type of the term. In a reference, the address points to the referenced term. An unbound variable is represented by a self-referencing pointer. In an atom, the address points to the record for the atom symbol in the symbol table. In a structure, $f(t_1, \ldots, t_n)$, the address points to a block of $n + 1$ consecutive words, where the first word points to the record for the functor, *f/n*, in the symbol table, and the remaining n words store the components of the structure. Arrays, floating-point numbers, and big integers are represented as special structures. Picat lists are singly-linked lists. In a list, *[H|T]*, the address points to a block of two consecutive words, where the first word stores the car, *H*, and the second word stores the cdr, *T*.

G.2 Fetching Arguments of Picat Calls

A C function that defines a Picat predicate should not take any argument. The following function is used in order to fetch arguments in the current Picat call.

- `TERM picat_get_call_arg(int i, int arity)`: Fetch the `i`th argument, where `arity` is the arity of the predicate, and `i` must be an integer between 1 and `arity`. The validity of the arguments is not checked, and an invalid argument may cause fatal errors.

G.3 Testing Picat Terms

The following functions are provided for testing Picat terms. They return `PICAT_TRUE` when they succeed and `PICAT_FALSE` when they fail.

- `int picat_is_var(TERM t)`: Term `t` is a variable.
- `int picat_is_attr_var(TERM t)`: Term `t` is an attributed variable.
- `int picat_is_dvar(TERM t)`: Term `t` is an attributed domain variable.
- `int picat_is_bool_dvar(TERM t)`: Term `t` is an attributed Boolean variable.
- `int picat_is_integer(TERM t)`: Term `t` is an integer.
- `int picat_is_float(TERM t)`: Term `t` is a floating-point number.
- `int picat_is_atom(TERM t)`: Term `t` is an atom.
- `int picat_is_nil(TERM t)`: Term `t` is nil, i.e., the empty list `[]`.
- `int picat_is_list(TERM t)`: Term `t` is a list.
- `int picat_is_string(TERM t)`: Term `t` is a string.
- `int picat_is_structure(TERM t)`: Term `t` is a structure (but not a list).
- `int picat_is_array(TERM t)`: Term `t` is an array.
- `int picat_is_compound(TERM t)`: True if either `picat_is_list(t)` or `picat_is_structure(t)` is true.
- `int picat_is_identical(TERM t1, TERM t2)`: `t1` and `t2` are identical. This function is equivalent to the Picat call `t1==t2`.
- `int picat_is_unifiable(TERM t1, TERM t2)`: `t1` and `t2` are unifiable. This is equivalent to the Picat call `not(not(t1=t2))`.

G.4 Converting Picat Terms into C

The following functions convert Picat terms to C. If a Picat term does not have the expected type, then the global C variable `exception`, which is of type `Term`, is assigned a term. A C program that uses these functions must check `exception` in order to see whether data are converted correctly. The converted data are only correct when `exception` is `(TERM)NULL`.

- `long picat_get_integer(TERM t)`: Convert the Picat integer `t` into C. The term `t` must be an integer; otherwise `exception` is set to `integer_expected` and 0 is returned. Note that precision may be lost if `t` is a big integer.
- `double picat_get_float(TERM t)`: Convert the Picat float `t` into C. The term `t` must be a floating-point number; otherwise `exception` is set to `number_expected`, and 0.0 is returned.
- `(char *) picat_get_atom_name(TERM t)`: Return a pointer to the string that is the name of atom `t`. The term `t` must be an atom; otherwise, `exception` is set to `atom_expected`, and `NULL` is returned.
- `(char *) picat_get_struct_name(TERM t)`: Return a pointer to the string that is the name of structure `t`. The term `t` must be a structure; otherwise, `exception` is set to `structure_expected`, and `NULL` is returned.

- `int picat_get_struct_arity(TERM t)`: Return the arity of term `t`. The term `t` must be a structure; otherwise, `exception` is set to `structure_expected` and 0 is returned.

G.5 Manipulating and Writing Picat Terms

- `int picat_unify(TERM t1,TERM t2)`: Unify two Picat terms `t1` and `t2`. The result is `PICAT_TRUE` if the unification succeeds, and `PICAT_FALSE` if the unification fails.
- `TERM picat_get_arg(int i,TERM t)`: Return the `i`th argument of term `t`. The term `t` must be compound, and `i` must be an integer that is between 1 and the arity of `t`; otherwise, `exception` is set to `compound_expected`, and the Picat integer `0` is returned.
- `TERM picat_get_car(TERM t)`: Return the car of the list `t`. The term `t` must be a non-empty list; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.
- `TERM picat_get_cdr(TERM t)`: Return the cdr of the list `t`. The term `t` must be a non-empty list; otherwise `exception` is set to `list_expected`, and the Picat integer `0` is returned.
- `void picat_write_term(TERM t)`: Send term `t` to the standard output stream.

G.6 Building Picat Terms

- `TERM picat_build_var()`: Return a free Picat variable.
- `TERM picat_build_integer(long i)`: Return a Picat integer whose value is `i`.
- `TERM picat_build_float(double f)`: Return a Picat float whose value is `f`.
- `TERM picat_build_atom(char *name)`: Return a Picat atom whose name is `name`.
- `TERM picat_build_nil()`: Return an empty Picat list.
- `TERM picat_build_list()`: Return a Picat list whose car and cdr are free variables.
- `TERM picat_build_structure(char *name, int arity)`: Return a Picat structure whose functor is `name`, and whose arity is `arity`. The structure's arguments are all free variables.
- `TERM picat_build_array(int n)`: Return a Picat array whose size is `n`. The array's arguments are all free variables.

G.7 Registering C-defined Predicates

The following function registers a predicate that is defined by a C function.

```
insert_cpred(char *name, int arity, int (*func)())
```

The first argument is the predicate name, the second argument is the arity, and the third argument is the name of the function that defines the predicate. The function that defines the predicate 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 C-defined built-ins of Picat. After registration, the predicate can be called. All C-defined predicates must be explicitly called with the module qualifier `bp`, as in `bp.p(a,X)`.

Example

Consider the Picat predicate:

```
p(a,X) => X = $f(a).
p(b,X) => X = [1].
p(c,X) => X = 1.2.
```

where the first argument is given and the second is unknown. The following steps show how to define this predicate in C, and how to make it callable from Picat.

Step 1 . Write a C function to implement the predicate. The following shows a sample:

```
#include "picat.h"

p(){
  TERM a1, a2, a, b, c, f1, l1, f12;
  char *name_ptr;

  /*   prepare Picat terms */
  a1 = picat_get_call_arg(1, 2);       /* first argument */
  a2 = picat_get_call_arg(2, 2);       /* second argument */
  a = picat_build_atom("a");
  b = picat_build_atom("b");
  c = picat_build_atom("c");
  f1 = picat_build_structure("f", 1);  /* f(a) */
  picat_unify(picat_get_arg(1, f1), a);
  l1 = picat_build_list();             /* [1] */
  picat_unify(picat_get_car(l1), picat_build_integer(1));
  picat_unify(picat_get_cdr(l1), picat_build_nil());
  f12 = picat_build_float(1.2);        /* 1.2 */

  /* code for the rules */
  if (!picat_is_atom(a1))
      return PICAT_FALSE;
```



```
  name_ptr = picat_get_name(a1);
  switch (*name_ptr){
  case 'a':
    return (picat_unify(a1, a) ?
              picat_unify(a2, f1) : PICAT_FALSE);
  case 'b':
    return (picat_unify(a1, b) ?
              picat_unify(a2, l1) : PICAT_FALSE);
  case 'c':
    return (picat_unify(a1, c) ?
              picat_unify(a2, f12) : PICAT_FALSE);
  default: return PICAT_FALSE;
  }
}
```

Step 2 . Insert the following two lines into `Cboot()` in `cpreds.c`:

```
extern int p();
insert_cpred("p", 2, p);
```

Step 3 . Modify the make file, if necessary, and recompile the system. Now, `p/2` is in the group of built-ins in Picat.

Step 4 . Use `bp.p(...)` to call the predicate.

```
picat> bp.p(a,X)
X = f(a)
```

Appendix H

Appendix: Tokens

```
/* Picat lexical rules
      [...] means optional
      {...} means 0, 1, or more occurrences
      "..." means as-is
      /*  ... */ comment

   Tokens to be returned:

      Token-type     lexeme
     ====================
      ATOM           a string of chars of the atom name
      VARIABLE       a string of chars of the variable name
      INTEGER        an integer literal
      FLOAT          a float literal
      STRING         a string of chars
      OPERATOR       a string of chars in the operator
      SEPARATOR      one of "(" ")" "{" "}"  "[" "]"
*/
line_terminator ->
    the LF character, also known as "newline"
    the CR character, also known as "return"
    the CR character followed by the LF character

input_char ->
    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 ->
    decimal_digit [decimal_digits_and_underscores]

decimal_digits_and_underscores ->
    decimal_digit_or_underscore
    decimal_digits_and_underscores decimal_digit_or_underscore

decimal_digit_or_underscore ->
    decimal_digit
    "_"
```

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

hex_digits ->
    hex_digit [hex_digits_and_underscores]

hex_digits_and_underscores ->
    hex_digit_or_underscore
    hex_digits_and_underscores hex_digit_or_underscore

hex_digit_or_underscore ->
    hex_digit
    "_"

octal_numeral ->
    "0O" octal_digits
    "0o" octal_digits

octal_digits ->
    octal_digit [octal_digits_and_underscores]

octal_digits_and_underscores ->
    octal_digit_or_underscore
    octal_digits_and_underscores octal_digit_or_underscore

octal_digit_or_underscore ->
    octal_digit
    "_"

binary_numeral ->
    "0b" binary_digits
    "0B" binary_digits

binary_digits:
    binary_digit [binary_digits_and_underscores]

binary_digits_and_underscores ->
    binary_digit_or_underscore
    binary_digits_and_underscores binary_digit_or_underscore

binary_digit_or_underscore:
    binary_digit
    "_"

real_literal ->
    decimal_numeral "." decimal_numeral [exponent_part]

exponent_part ->
    exponent_indicator signed_integer

exponent_indicator ->
    "e"
    "E"

signed_integer ->
    [sign] decimal_numeral

sign ->
    "+"
    "-"

separator ->
    one of "(" ")" "{" "}"  "[" "]"

operator ->
    one of
      "=" "!=" ">" ">=" "<" "<=" "=<" ".." "!"
      "," ";" ":" "::" "." ". " (dot-whitespace)
```



```
      "=>" "?=>" "==" "!==" ":=" "|" "$" "@"
      "/\" "\/" "~" "^" "<<" ">>"
      "+" "-" "*" "**" "/" "/>" "/<" "^"
      "#=" "#!=" "#>" "#>=" "#<" "#<=" "#=<"
      "#/\" "#\/" "#~" "#^" "#=>" "#<=>"
      "@>" "@>=" "@<" "@<=" "@=<"

small_letter ->
    one of "a" "b" ... "z"

capital_letter ->
    one of "A" "B" ... "Z"

decimal_digit ->
    one of "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"

hd ->
    hex_digit

hex_digit ->
    one of
       "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
       "a" "b" "c" "d" "e" "f" "A" "B" "C" "D" "E" "F"

octal_digit ->
    one of "0" "1" "2" "3" "4" "5" "6" "7"

binary_digit ->
    one of "0" "1"

escape_sequence ->
    "\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 \ */
    unicode_escape

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

Appendix I

Appendix: Grammar

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

program_body ->
    {predicate_definition | function_definition | actor_definition}

module_declaration ->
    "module" atom eor

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

import_item ->
    atom

predicate_definition ->
    {predicate_directive} predicate_rule_or_fact {predicate_rule_or_fact}

function_definition ->
    {function_directive} function_rule_or_fact {function_rule_or_fact}

actor_definition ->
    ["private"] action_rule {(action_rule
                              | nonbacktrackable_predicate_rule)}

function_directive ->
    "private"
    "table"

predicate_directive ->
    "private"
    "table" ["(" table_mode {"," table_mode} ")" ]
    "index" index_declaration {"," index_declaration}
```



```
index_declaration ->
    "(" index_mode {"," index_mode} ")"

index_mode ->
   "+"
   "-"

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

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 "=" expression ["," condition] "=>" body eor

function_fact ->
    head "=" argument eor

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

event_pattern ->
    term {',' term}

condition -> goal

body -> goal

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 ")" ["do"] goal "end"
   "do" goal "while" "(" goal ")"
   expression {bin_rel_op expression}

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

expression ->
    concat_expression

concat_expression ->
    range_expression ["++" concat_expression]

range_expression ->
   or_expression [".." or_expression [".." or_expression]]

or_expression  ->
    xor_expression
    or_expression "\/" xor_expression

xor_expression ->
```



```
    and_expression
    xor_expression "^" and_expression              % 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

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] "]"       % subscript notation
    variable "@" term ["@"]                        % as-pattern
    variable
    integer
    float
    atom_or_call
    list_expression
    array_expression
    function_call
    term_constructor
    primary_expression "." atom_or_call            % dot-notation

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 array_expression_suffix "}"

array_expression_suffix ->
    ":" iterator {"," (iterator | condition)}      % array comprehension
    {"," argument}
```

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

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

term_constructor ->
    "$" goal ["$"]

/* a term has the same form as a goal except that it cannot contain loops
   or if-then-else. Note that subscript notations, range expressions, and
   list comprehensions are still treated as functions in term constructors */
```


Appendix J

Appendix: Operators

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

Appendix K

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
- X `@<` Y
- X `@<=` Y
- X `@=<` Y
- X `@>` Y
- X `@>=` Y
- $Term_1$ `++` $Term_2$ = $List$
- `[`X `:` I `in` D`,... ]` = $List$
- L `..` U = $List$
- L `..` $Step$ `..` U = $List$
- `-`X = Y
- `+`X = Y
- X `+` Y = Z
- X `-` Y = Z
- X `*` Y = Z
- X `/` Y = Z
- X `//` Y = Z
- X `div` Y = Z
- X `/<` Y = Z
- X `/>` Y = Z
- X `**` Y = Z
- X `mod` Y = Z
- X `rem` Y = Z
- `~`X = Y
- X `\/` Y = Z
- X `/\` Y = Z
- X `^` Y = Z
- X `<<` Y = Z
- X `>>` Y = Z
- Var`[`$Index_1$`,...,`$Index_n$`]`
- $Goal_1$`,`$Goal_2$
- $Goal_1$ `&&` $Goal_2$
- $Goal_1$`;`$Goal_2$
- $Goal_1$ `||` $Goal_2$
- `acyclic_term(`$Term$`)`
- `and_to_list(`$Conj$`)` = $List$
- `append(`X`,`Y`,`Z`)` (nondet)
- `append(`X`,`Y`,`Z`,`T`)` (nondet)
- `apply(`S`,`Arg_1`,...,`Arg_n`)` = Val
- `arity(`$Struct$`)` = $Arity$
- `array(`$Term$`)`
- `atom(`$Term$`)`
- `atom_chars(`Atm`)` = $String$
- `atom_codes(`Atm`)` = $List$
- `atomic(`$Term$`)`
- `attr_var(`$Term$`)`
- `avg(`$List$`)` = Val
- `bind_vars(`$Term$`,`Val`)`
- `between(`$From$`,`To`,`X`)` (nondet)
- `call(`S`,`Arg_1`,...,`Arg_n`)`
- `call_cleanup(`S`,`$Cleanup$`)`
- `catch(`S`,`$Exception$`,`$Handler$`)`
- `char(`$Term$`)`
- `chr(`$Code$`)` = $Char$
- `clear(`Map`)`
- `compare_terms(`$Term_1$`,`$Term_2$`)` = Res
- `compound(`$Term$`)`
- `copy_term(`$Term_1$`)` = $Term_2$
- `count_all(`$Call$`)` = Int
- `delete(`$List$`,`X`)` = $ResList$
- `delete_all(`$List$`,`X`)` = $ResList$
- `different_terms(`$Term_1$`,`$Term_2$`)`
- `digit(`$Char$`)`
- `dvar(`$Term$`)`
- `dvar_or_int(`$Term$`)`
- `fail`
- `false`
- `find_all(`$Template$`,`$Call$`)` = $List$
- `findall(`$Template$`,`$Call$`)` = $List$
- `first(`$Compound$`)` = $Term$
- `flatten(`$List1$`)` = $List2$
- `float(`$Term$`)`
- `fold(`F`,`ACC`,`$List$`)` = Res
- `freeze(`X`,`$Goal$`)`
- `get(`$MapOrAttrVar$`,`Key`)` = Val
- `get(`$MapOrAttrVar$`,`Key`,`$Default$`)`=Val
- `get_global_map()` = Map
- `get_heap_map()` = Map
- `get_table_map()` = Map
- `ground(`$Term$`)`
- `handle_exception(`$Term$`,`$Term$`)`

- `has_key(`$MapOrAttrVar$`,`Key`)`
- `hash_code(`$Term$`)` = Int
- `head(`$List$`)` = $Term$
- `insert(`$List$`,`$Index$`,`Elm`)` = $ResList$
- `insert_all(`$List$`,`$Index$`,`$AList$`)` = $ResList$
- `insert_ordered(`$List$`,`$Term$`)` = R
- `insert_ordered_down(`$List$`,`$Term$`)` = R
- `integer(`$Term$`)`
- `is(`Exp`,`Exp`)`
- `keys(`$MapOrAttrVar$`)` = $List$
- `last(`$Compound$`)` = $Term$
- `len(`$Term$`)` = Len
- `length(`$Term$`)` = Len
- `list(`$Term$`)`
- `list_to_and(`$List$`)` = $Conj$
- `lowercase(`$Char$`)`
- `map(`F`,`$List$`)` = $ListRes$
- `map(`F`,`$ListA$`,`$ListB$`)` = $ListRes$
- `map(`$Term$`)`
- `map_to_list(`Map`)` = $List$
- `max(`$List$`)` = Val
- `max(`X`,`Y`)` = Val
- `maxof(`$Call$`,`$Objective$`)`
- `maxof(`$Call$`,`$Objective$`,`$ReportCall$`)`
- `maxof_inc(`$Call$`,`$Objective$`)`
- `maxof_inc(`$Call$`,`$Objective$`,`$ReportCall$`)`
- `membchk(`$Term$`,`$List$`)`
- `member(`$Term$`,`$List$`)` (nondet)
- `min(`$List$`)` = Val
- `min(`X`,`Y`)` = Val
- `minof(`$Call$`,`$Objective$`)`
- `minof(`$Call$`,`$Objective$`,`$ReportCall$`)`
- `minof_inc(`$Call$`,`$Objective$`)`
- `minof_inc(`$Call$`,`$Objective$`,`$ReportCall$`)`
- `name(`$Struct$`)` = $Name$
- `new_array(`D_1`,...,`D_n`)` = Arr
- `new_list(`N`)` = $List$
- `new_list(`N`,`$InitVal$`)` = $List$
- `new_map(`Int`,`$PairsList$`)` = Map
- `new_map(`$IntOrPairsList$`)` = Map
- `new_set(`Int`,`$ElmsList$`)` = Map
- `new_set(`$IntOrElmsList$`)` = Map
- `new_struct(`$Name$`,`$IntOrList$`)` = $Struct$
- `nonvar(`$Term$`)`
- `not` $Call$
- `nth(`I`,`$List$`,`Val`)` (nondet)
- `number(`$Term$`)`
- `number_chars(`Num`)` = $String$
- `number_codes(`Num`)` = $List$
- `number_vars(`$Term$`,`N_0`)` = N_1
- `once` $Call$
- `ord(`$Char$`)` = Int
- `parse_term(`$String$`)` = $Term$)
- `parse_term(`$String$`,`$Term$`,`$Vars$`)`
- `post_event(`X`,`$Event$`)`
- `post_event_any(`X`,`$Event$`)`
- `post_event_bound(`X`)`
- `post_event_dom(`X`,`$Event$`)`
- `post_event_ins(`X`)`
- `prod(`$List$`)` = Val
- `put(`$MapOrAttrVar$`,`Key`,`Val`)`
- `put(`$MapOrAttrVar$`,`Key`)`
- `real(`$Term$`)`
- `reduce(`$ListOrFun$`,`$FunOrList$`)` = $Value$
- `reduce(`$ListOrFun$`,`$FunOrList$`,`$InitVal$`)` = $Value$
- `remove_dups(`$List$`)` = $ResList$
- `repeat` (nondet)
- `reverse(`$List$`)` = $ResList$
- `second(`$Compound$`)` = $Term$
- `select(`X`,`$List$`,`$ResList$`)` (nondet)
- `size(`Map`)` = $Size$
- `slice(`$ListOrArray$`,`$From$`)`
- `slice(`$ListOrArray$`,`$From$`,`To`)`
- `sort(`$List$`)` = $SList$
- `sort(`$List$`,`$KeyIndex$`)` = $SList$
- `sort_down(`$List$`)` = $SList$
- `sort_down(`$List$`,`$KeyIndex$`)` = $SList$
- `sort_down_remove_dups(`$List$`)` = $SList$
- `sort_down_remove_dups(`$List$`,`$KeyIndex$`)` = $SList$
- `sort_remove_dups(`$List$`)` = $SList$
- `sort_remove_dups(`$List$`,`$KeyIndex$`)` = $SList$
- `string(`$Term$`)`
- `struct(`$Term$`)`
- `subsumes(`$Term_1$`,`$Term_2$`)`
- `sum(`$List$`)` = Val
- `tail(`$List$`)` = $Term$
- `throw` E
- `to_array(`$List$`)` = $Array$
- `to_atom(`$String$`)` = $Atom$
- `to_binary_string(`Int`)` = $String$
- `to_codes(`$Term$`)` = $List$
- `to_fstring(`$Format$`,`$Args...$`)` = $String$
- `to_hex_string(`Int`)` = $String$
- `to_integer(`$NumOrCharOrStr$`)` = Int
- `to_list(`$Struct$`)` = $List$
- `to_lowercase(`$String$`)` = $LString$
- `to_oct_string(`Int`)` = $String$
- `to_real(`$NumOrStr$`)` = $Real$
- `to_string(`$Term$`)` = $String$
- `to_uppercase(`$String$`)` = $UString$
- `true`
- `uppercase(`$Char$`)`
- `values(`$MapOrAttrVar$`)` = $List$
- `var(`$Term$`)`
- `variant(`$Term_1$`,`$Term_2$`)`
- `vars(`$Term$`)` = $Vars$
- `zip(`$List_1$`,`$List_2$`)` = $List$
- `zip(`$List_1$`,`$List_2$`,`$List_3$`)` = $List$
- `zip(`$List_1$`,`$List_2$`,`$List_3$`,`$List_4$`)` = $List$

odule `math` (imported by default)

- `abs(`X`) = `Val
- `acos(`X`) = `Val
- `asin(`X`) = `Val
- `atan(`X`) = `Val
- `atan2(`X`,`Y`) = `Val
- `ceiling(`X`) = `Val
- `cos(`X`) = `Val
- `e() = 2.71828182845904523536`
- `even(`Int`)`
- `exp(`X`) = `Val
- `gcd(`A`,`B`) = `Val
- `floor(`X`) = `Val
- `frand() = `Val
- `log(`X`) = `Val
- `log(B,`X`) = `Val
- `log10(`X`) = `Val
- `log2(`X`) = `Val
- `modf(`X`) = (`$IntVal$`,`$FractVal$`)`
- `odd(`Int`)`
- `pi() = 3.14159265358979323846`
- `pow(`X`,`Y`) = `Val
- `prime(`Int`)`
- `primes(`Int`) = `$List$
- `rand_max() = `Val
- `random = `Val
- `random(`$Seed$`) = `Val
- `random2() = `Int
- `round(`X`) = `Val
- `sign(`X`) = `Val
- `sin(`X`) = `Val
- `sqrt(`X`) = `Val
- `tan(`X`) = `Val
- `truncate(`X`) = `Val

odule `io` (imported by default)

- `at_end_of_stream(`FD`)`
- `close(`FD`)`
- `flush(`FD`)`
- `flush()`
- `nl(`FD`)`
- `nl()`
- `open(`$Name$`) = `FD
- `open(`$Name$`,`$Mode$`) = `FD
- `peek_byte(`FD`) = `Val
- `peek_char(`FD`) = `Val
- `print(`FD`,`$Term$`)`
- `print(`$Term$`)`
- `printf(`FD`,`$Format$`,`$Args$`...)`
- `println(`FD`,`$Term$`)`
- `println(`$Term$`)`
- `read_atom(`FD`) = `$Atom$
- `read_atom() = `$Atom$
- `read_byte(`FD`) = `Val
- `read_byte(`FD`,`N`) = `$List$
- `read_byte() = `Val
- `read_char(`FD`) = `Val
- `read_char(`FD`,`N`) = `$String$
- `read_char() = `Val
- `read_char_code(`FD`) = `Val
- `read_char_code(`FD`,`N`) = `$List$
- `read_char_code() = `Val
- `read_file_bytes(`$File$`) = `$List$
- `read_file_bytes() = `$List$
- `read_file_chars(`$File$`) = `$String$
- `read_file_chars() = `$String$
- `read_file_codes(`$File$`) = `$List$
- `read_file_codes() = `$List$
- `read_file_lines(`$File$`) = `$List$
- `read_file_lines() = `$List$
- `read_file_terms(`$File$`) = `$List$
- `read_file_terms() = `$List$
- `read_int(`FD`) = `Int
- `read_int() = `Int
- `read_line(`FD`) = `$String$
- `read_line() = `$String$
- `read_number(`FD`) = `$Number$
- `read_number() = `$Number$
- `read_picat_token(`FD`) = `$TokenValue$
- `read_picat_token(`FD`,`$TokenType$`,`$TokenValue$`)`
- `read_picat_token(`$TokenType$`,`$TokenValue$`)`
- `read_picat_token() = `$TokenValue$
- `read_real(`FD`) = `$Real$
- `read_real() = `$Real$
- `read_term(`FD`) = `$Term$
- `read_term() = `$Term$
- `readln(`FD`) = `$String$
- `readln() = `$String$
- `write(`FD`,`$Term$`)`
- `write(`$Term$`)`
- `write_byte(`$Bytes$`)`
- `write_byte(`FD`,`$Bytes$`)`
- `write_char(`$Chars$`)`
- `write_char(`FD`,`$Chars$`)`
- `write_char_code(`$Codes$`)`
- `write_char_code(`FD`,`$Codes$`)`
- `writef(`FD`,`$Format$`,`$Args$`...)`
- `writeln(`FD`,`$Term$`)`
- `writeln(`$Term$`)`

Module `ordset`

- `delete(`$OSet$`,`Elm`) = `$OSet1$
- `disjoint(`$OSet1$`,`$OSet2$`)`
- `insert(`$OSet$`,`Elm`) = `$OSet1$
- `intersection(`$OSet1$`,`$OSet2$`)=`$OSet3$
- `new_ordset(`$List$`)`
- `ordset(`$Term$`)`
- `subset(`$OSet1$`,`$OSet2$`)`
- `subtract(`$OSet1$`,`$OSet2$`)=`$OSet3$
- `union(`$OSet1$`,`$OSet2$`)=`$OSet3$

Module `os`

- `cd(`$Path$`)`
- `chdir(`$Path$`)`
- `cp(`$FromPath$`,`$ToPath$`)`
- `cwd() = `$Path$
- `directory(`$Path$`)`
- `dir`
- `env_exists(`$Name$`)`
- `executable(`$Path$`)`
- `exists(`$Path$`)`
- `file(`$Path$`)`
- `file_base_name(`$Path$`) = `$String$
- `file_directory_name(`$Path$`) = `$String$
- `file_exists(`$Path$`)`
- `getenv(`$EnvString$`) = `$String$
- `listdir(`$Path$`) = `$List$
- `ls`
- `mkdir(`$Path$`)`
- `pwd() = `$Path$
- `readable(`$Path$`)`
- `rename(`Old`,`New`)`
- `rm(`$Path$`)`
- `rmdir(`$Path$`)`
- `separator() = `Val
- `size(`$Path$`) = `Int
- `writable(`$Path$`)`

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

- `#~`X
- X `#!=` Y
- X `#/\` Y
- X `#<` Y
- X `#<=` Y
- X `#<=>` Y
- X `#=` Y
- X `#=<` Y
- X `#=>` Y
- X `#>` Y
- X `#>=` Y
- X `#\/` Y
- X `#^` Y
- $Vars$ `::` Exp
- $Vars$ `notin` Exp
- `all_different(`$FDVars$`)`
- `all_different_except_0(`$FDVars$`)`
- `all_distinct(`$FDVars$`)`
- `assignment(`$FDVars1$`,`$FDVars2$`)`
- `at_least(`N`,`L`,`V`):`
- `at_most(`N`,`L`,`V`):`
- `circuit(`$FDVars$`)`
- `count(`V`,`$FDVars$`,`Rel`,`N`)`
- `cumulative(`Ss`,`Ds`,`Rs`,`$Limit$`)`
- `diffn(`$RectangleList$`)`
- `disjunctive_tasks(`$Tasks$`)` (cp only)
- `element(`I`,`$List$`,`V`)`
- `exactly(`N`,`L`,`V`):`
- `fd_degree(`$FDVar$`) = `$Degree$` (cp only)
- `fd_disjoint(`$DVar1$`,`$DVar2$`)`
- `fd_dom(`$FDVar$`) = `$List$
- `fd_false(`$FDVar$`,`Elm`)`
- `fd_max(`$FDVar$`) = `Max
- `fd_min(`$FDVar$`) = `Min
- `fd_min_max(`$FDVar$`,`Min`,`Max`)`
- `fd_next(`$FDVar$`,`Elm`) = `$NextElm$
- `fd_prev(`$FDVar$`,`Elm`) = `$PrevElm$
- `fd_set_false(`$FDVar$`,`Elm`)` (cp only)
- `fd_size(`$FDVar$`) = `$Size$
- `fd_true(`$FDVar$`,`Elm`)`
- `fd_vector_min_max(`Min`,`Max`)`
- `global_cardinality(`$List$`,`$Pairs$`)`
- `indomain(`Var`)` (nondet) (cp only)
- `indomain_down(`Var`)` (nondet) (cp only)
- `lex_le(`L_1`,`L_2`)`
- `lex_lt(`L_1`,`L_2`)`
- `matrix_element(`$Matrix$`,`I`,`J`,`V`)`
- `neqs(`$NeqList$`)` (cp only)
- `new_dvar() = `$FDVar$
- `new_fd_var() = `$FDVar$
- `regular(`X`,`Q`,`S`,`D`,`$Q0$`,`F`)`
- `scalar_product(`A`,`X`,`$Product$`)`
- `scalar_product(`A`,`X`,`Rel`,`$Product$`)`
- `serialized(`$Starts$`,`$Durations$`)`
- `solve(`$Options$`,`$Vars$`)` (nondet)
- `solve(`$Vars$`)` (nondet)
- `solve_all(`$Options$`,`$Vars$`) = `$List$
- `solve_all(`$Vars$`) = `$List$
- `subcircuit(`$FDVars$`)` (cp only)
- `table_in(`$DVars$`,`R`)`
- `table_notin(`$DVars$`,`R`)`

Module `planner`

- `best_plan(`S`,`$Limit$`,`$Plan$`)`
- `best_plan(`S`,`$Plan$`,`$PlanCost$`)`
- `best_plan(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`
- `best_plan(`S`,`$Plan$`)`
- `best_plan_bb(`S`,`$Limit$`,`$Plan$`)`
- `best_plan_bb(`S`,`$Plan$`,`$PlanCost$`)`
- `best_plan_bb(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`
- `best_plan_bb(`S`,`$Plan$`)`
- `best_plan_nondet(`S`,`$Limit$`,`$Plan$`)` (nondet)
- `best_plan_nondet(`S`,`$Plan$`,`$PlanCost$`)` (nondet)
- `best_plan_nondet(`S`,`$Limit$`,`$Plan$`,`$Cost$`)` (nondet)
- `best_plan_nondet(`S`,`$Plan$`)` (nondet)
- `best_plan_unbounded(`S`,`$Limit$`,`$Plan$`)`
- `best_plan_unbounded(`S`,`$Plan$`,`$PlanCost$`)`
- `best_plan_unbounded(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`
- `best_plan_unbounded(`S`,`$Plan$`)`
- `current_plan()=`$Plan$
- `current_resource()=`$Amount$
- `current_resource_plan_cost(`$Amount$`,`$Plan$`,`$Cost$`)`
- `is_tabled_state(`S`)`
- `plan(`S`,`$Limit$`,`$Plan$`)`
- `plan(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`
- `plan(`S`,`$Plan$`)`
- `plan_unbounded(`S`,`$Limit$`,`$Plan$`)`
- `plan_unbounded(`S`,`$Plan$`,`$PlanCost$`)`
- `plan_unbounded(`S`,`$Limit$`,`$Plan$`,`$Cost$`)`
- `plan_unbounded(`S`,`$Plan$`)`

odule `datetime`

- `current_datetime()` = *DateTime*
- `current_day()` = *WDay*
- `current_date()` = *Date*
- `current_time()` = *Time*

odule `sys` (imported by default)

- `abort`
- `cl(`*File*`)`
- `cl_facts(`*Facts*`)`
- `cl`
- `command(`*String*`)`
- `compile(`*File*`)`
- `debug`
- `exit`
- `garbage_collect(Size)`
- `garbage_collect`
- `halt`
- `initialize_table`
- `load(`*File*`)`
- `nodebug`
- `nospy`
- `notrace`
- `spy` *Functor*
- `statistics(`*Name*`,`*Value*`)` (nondet)
- `statistics`
- `time(`*Goal*`)`
- `time2(`*Goal*`)`
- `time_out(`*Goal*`,`*Limit*`,`*Res*`)`
- `trace`

Module `util`

- `array_matrix_to_list(`*Matrix*`)` = *List*
- `array_matrix_to_list_matrix(`*AMatrix*`)` = *LMatrix*
- `find(`*String*`,`*SubString*`,`*From*`,`*To*`)` (nondet)
- `find_first_of(`*Term*`,`*Pattern*`)` = *Index*
- `find_ignore_case(`*String*`,`*SubString*`,`*From*`,`*To*`)` (nondet)
- `find_last_of(`*Term*`,`*Pattern*`)` = *Index*
- `join(`*Words*`)` = *String*
- `join(`*Words*`,`*Separator*`)` = *String*
- `list_matrix_to_array_matrix(`*LMatrix*`)` = *AMatrix*
- `lstrip(`*List*`)` = *List*
- `lstrip(`*List*`,`*Elms*`)` = *List*
- `matrix_multi(`*MatrixA*`,`*MatrixB*`)` = *MatrixC*
- `permutation(`*List*`,`*Perm*`)` (nondet)
- `permutations(`*List*`)` = *Lists*
- `power_set(`*List*`)` = *Lists*
- `replace(`*Term*`,`*Old*`,`*New*`)` = *NewTerm*
- `replace_at(`*Term*`,`*Index*`,`*New*`)` = *NewTerm*
- `rstrip(`*List*`)` = *List*
- `rstrip(`*List*`,`*Elms*`)` = *List*
- `split(`*List*`)` = *Words*
- `split(`*List*`,`*Separators*`)` = *Words*
- `strip(`*List*`)` = *List*
- `strip(`*List*`,`*Elms*`)` = *List*
- `transpose(`*Matrix*`)` = *Transposed*

Index