Picat

2019

About

Picat is a simple, and yet powerful, logic-based multi-paradigm programming language aimed for general-purpose applications. Picat is a rule-based language, in which predicates, functions, and actors are defined with pattern-matching rules. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, list comprehensions, constraints, and tabling. Picat also provides imperative language constructs, such as assignments and loops, for programming everyday things.

About

The Picat implementation, which is based on a well-designed virtual machine and incorporates a memory manager that garbage-collects and expands the stacks and data areas when needed, is efficient and scalable. Picat can be used for not only symbolic computations, which is a traditional application domain of declarative languages, but also for scripting and modeling tasks.

Name

The letters in the name summarize Picat's features:

1. **P**attern-matching: Predicates and functions are defined with pattern-matching rules.
2. **I**ntuitive: Picat provides assignment and loop statements for programming everyday things.
3. **C**onstraints: Picat supports constraint programming.
4. **A**ctors: Actors are event-driven calls. An actor can be attached to a channel in order to watch and to process its events.
5. **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.

Data types

Picat is a dynamically-typed language, in which type checking occurs at runtime. 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 value in Picat can be primitive or compound.

Data types

```
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]
```

Data types

```
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
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]
```

Predicates

A predicate is defined with pattern-matching rules. Picat has two types of rules: the nonbacktrackable rule `Head, Cond => Body`, and the backtrackable rule `Head, Cond ?=> Body`. The `Head` takes the form `p(t1, ..., tn)`, where `p` is called the predicate name, and `n` is called the arity. The condition `Cond`, which is an optional goal, specifies a condition under which the rule is applicable.

```
fib(0,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).
```

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(t1,..., tn)` and `X` holds the return value.

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

Conditional Statements and Loops

- **Conditional Expressions**
  ```
  fib(N) = cond((N==0;N==1), 1,
  fib(N-1)+fib(N-2))
  ```
- **Foreach**
  ```
  foreach (E1 in D1, Cond1,..., En in Dn,
  Condn) Goal end
  ```
- **While**
  ```
  while (Cond) Goal end
  ```
- **Do while**
  ```
  do Goal while (Cond)
  ```

Exceptions

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 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)`.

Tabling

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.

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

Without tabling, fib(N) takes exponential time.
With tabling, fib(N) takes linear time.

Higher-Order Calls

A predicate or function is said to be higher-order if it takes calls as arguments.

- `call(S,A1,...,An)`
  Calls the named predicate with the specified arguments
- `apply(S,A1,...,An)`
  Similar to call, except apply returns a value
- `findall(Template, Call)`
  Returns a list of all possible solutions of `Call` in the form `Template`. `findall` forms a name scope like a loop.

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

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=> 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.

```
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.

Picat vs. Prolog

Although Picat is a multi-paradigm language, its core is underpinned by logic programming concepts, including logic variables, unification, and backtracking.

Like in Prolog, logic variables in Picat are value holders. A logic variable can be bound to any term, including another logic variable. Logic variables are single-assignment, meaning that once a variable is bound to a value, the variable takes the identity of the value, and the variable cannot be bound again, unless the value is a variable or contains variables.

In both Prolog and Picat, unification is a basic operation, which can be utilized to unify terms. Unlike Prolog, Picat uses pattern-matching, rather than unification, to select applicable rules for a call.

Picat vs. Prolog

Predicate `membchk/2`, checks whether a term occurs in a list.

```
% Prolog
membchk(X,[X|_]) :- !.
membchk(X,[_|T]) :- membchk(X,T).

% Picat
membchk(X,[X|_]) => true.
membchk(X,[_|T]) => membchk(X,T).
```

For a call `membchk(X,L)`, if both `X` and `L` are ground, then the call has the same behavior under both the Prolog and the Picat definitions. However the call `membchk(a,_)` and the call `membchk(_,[a])` succeed in Prolog, but they fail in Picat.

Picat vs. Prolog

Picat, like Prolog, supports backtracking.

```
% Prolog
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

% Picat
member(X,[Y|_]) ?=> X = Y.
member(X,[_|T]) => member(X,T).
```

For a call `member(X,L)`, if `L` is a complete list, A list is complete if it is empty, or if its tail is complete. If `L` is incomplete, then the call can succeed an infinite number of times under the Prolog definition, because unifying `L` with a cons always succeeds when `L` is a variable. In contrast, pattern matching never changes call arguments in Picat. Therefore, the call `member(X,L)` can never succeed more times than the number of elements in `L`.