Learning Programming in Picat by Examples from Google Code Jam

Neng-Fa Zhou

February, 2016

Abstract

Picat (picat-lang.org) is a logic-based multi-paradigm programming language that integrates logic programming, functional programming, constraint programming, dynamic programming with tabling, and scripting. Picat is underpinned by the same logic programming concepts as Prolog, but it is more efficient, reliable, and convenient than Prolog. Picat integrates features from other languages, including arrays, maps, functions, list and array comprehensions, loops, and assignments. Picat's support for explicit unification, explicit non-determinism, tabling, and constraints makes Picat more suitable than functional languages (such as Haskell and F#) and scripting languages (such as Python and Ruby) for symbolic computations. Picat can give a competitive edge for many applications. This article provides a quick introduction to Picat using examples from Google Code Jam (GCJ).

1.1 Introduction

Picat is a simple and yet powerful logic-based multi-paradigm programming language for general-purpose applications. Picat's core is underpinned by logic programming concepts, including *logic variables*, *unification*, and *backtracking*. Logic variables, like variables in mathematics, are value holders. A logic variable can be bound to any term, including another logic variable. Figure 1.1 gives the types of terms in Picat. Picat is a dynamically-typed language, which means that type checking occurs at runtime.

A variable name is an identifier that begins with a capital letter or the underscore; for example, `X1` and `_abc` are variable names. The underscore itself `_` is used for *anonymous variables*, and each occurrence of the underscore indicates a different variable.

An *atomic* value can be an *atom* or a *number*. An *atom* is a constant symbol. An atom name can be either unquoted or quoted. 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 atom is a single-quoted sequence of arbitrary characters. For example, `x1`, `x_1`, `'_abc'`, and `'a+b'` are atom names.

A *compound* value can be a *list* or a *structure*; for example, `[a,b,c]` is a list and

`f(a,b,c)` is a structure.[1] Lists are singly-linked lists. A string is a list of characters; for example, `"a+b"` is the same as `[a,'+',b]`. An array is a special structure; for example, `{a,b,c}` is an array. A *map* is a special structure that contains a set of key-value pairs, and a *set* is a special map that contains only keys; both are hash tables.

Each type provides a set of built-in functions and predicates. Each of the type names in Figure 1.1, except `term` and `set`, is a type-checking predicate. For example, `list(`L`)` tests if L is a list. Let L be a compound term. The index notation L`[`I`]` is a special function that returns the Ith component of list L, with $L[1]$ referring to the first element of L. An index notation can take multiple subscripts. The *cons* operator `[`H`|`T`]` builds a new list by adding H to the front of T. The *concatenation* operator L_1 `++` L_2 returns the concatenated list of L_1 and L_2.

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable* rule $Head, Cond$ `=>` $Body$, and the *backtrackable* rule $Head, Cond$ `?=>` $Body$. In a predicate definition, the $Head$ takes the form $p(t_1,\ldots,t_n)$, where p is a predicate name, and n is the arity. 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, 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. In a function definition, the $Head$ takes the form $f(t_1,\ldots,t_n) = Term$, where f is a function name and $Term$ is a result to be returned.

Picat supports tabling for dynamic programming solutions. Both predicates and functions can be tabled. In order to have all calls and answers of a predicate or function tabled, you just need to add the keyword `table` before the first rule. For

[1] A structure requires a proceding dollar symbol, as in `$f(a,b,c)`, to distinguish the structure from a function call, unless the structure is special, or it occurs in a special context.

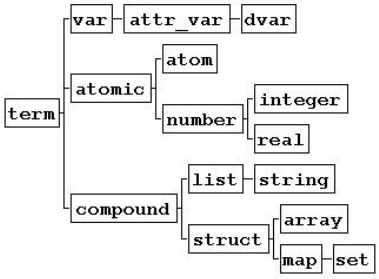

Figure 1.1: Picat's data types

a predicate definition, the keyword `table` can be followed by a tuple of table modes, including `+` (input), `-` (output), `min`, `max`, and `nt` (not tabled). For a predicate with a table mode declaration that contains `min` or `max`, Picat tables one optimal answer for each tuple of the input arguments.

Picat supports loops for describing repetitions, and comprehensions for constructing lists and arrays using properties. Other features of Picat include assignments, global maps for storing permanent data, higher-order functions, action rules for defining event-driven actors, and modules for modeling and solving constraint satisfaction problems with CP, SAT, and MIP.

This article gives programs in Picat for several Google Code Jam (GCJ) practice problems. The objective is to familiarize you with Picat's language features and well-used built-ins. More details of the Picat language can be found in the User's Guide.[2] The constraint programming and planning modules are detailed in the book "Constraint Solving and Planning with Picat" by N.-F. Zhou, H. Kjellerstrand, and J. Fruhman, Springer, 2015; in the book, Agostino Dovier gives a short account of the history of logic programming that led to the design of Picat. Solutions in Picat for several GCJ problems that utilize tabling and constraints can be found in "Declaratively Solving Google Code Jam Problems with Picat" by S. Dymchenko and M. Mykhailova in PADL'15. Many more programs for GCJ problems can be found at:

```
http://picat-lang.org/gcj/index.html
```

You need to download the Picat system from `picat-lang.org` and install it. Binary executables are available for most popular platforms, including Windows, Linux, and MacOS. The C-source code is also available, so executables can be made for other platforms. All of the programs given in this article contain a `main` predicate, and can be run from a command line. Let `prog.pi` be a Picat program and `practice.in` be an input file. You can run the program on the input using the following command:

```
picat prog < practice.in > practice.out
```

The output is stored in a file named `practice.out`.

1.2 Reverse Words

Reverse Words is an easy practice problem.[3] Given a list of space separated words, reverse the order of the words.

```
import util.

main =>
    T = read_line().to_int(),
    foreach (TC in 1..T)
        Words = read_line().split(),
```

[2] *A User's Guide to Picat* by N.-F. Zhou and J. Fruhman (`http://picat-lang.org`)

[3] `https://code.google.com/codejam/contest/351101/dashboard#s=p1`

```
        printf("Case #%w: %s\n", TC, Words.reverse().join())
    end.
```

```
import util.

main =>
    T = read_line().to_int(),
    foreach (TC in 1..T)
        Words = read_line().split(),
        printf("Case #%w: %s\n", TC, Words.reverse().join())
    end.
```

This program imports the `util` module, from which the functions `split` and `join` are used. Several modules, including `basic`, `math`, `io`, and `sys`, are preloaded once the Picat system is started, and built-ins defined in these modules can be used without import.

The `main` predicate is defined by one rule. The first line in the body of `main` reads `T`, the number of test cases, from the standard input `stdin`. The function `read_line()`, which is defined in the `io` module, reads the next line from `stdin`, and returns it as a string. The function `to_int()` converts the string into an integer. In Picat, the dot (.) operator can be utilized to chain function calls. The chain of calls `read_line().to_int()` is the same as `to_int(read_line())`.

The *foreach* loop iterates over the test case numbers in the *range* `1..T`. For each number, `TC`, the loop reads a line of words and prints out a line of the same words in reversed order. The range `1..T` is the same as the list `[1,2,...,T]`. The Picat compiler translates the loop into tail recursion such that the list is not actually constructed.

A `foreach` loop statement has the following general format:

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

where each E_i `in` D_i is an *iterator* (E_i is an iterating pattern and D_i is an expression that gives a compound value), and each $Cond_i$ is an optional condition on the patterns E_1 through E_i. The `foreach` loop executes *Goal* once for every possible combination of values in the iterators that satisfies the conditions.

Each 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. In the above example, the variable `Words` is local to the loop, meaning that for each `TC` there is a list of words `Words`.

The `split(`*String*`)` function takes a string and splits it into a list of tokens, using white spaces as split characters. For example, the call `split("Hello Picat World")` returns `["Hello", "Picat", "World"]`. The `reverse(`*List*`)` function takes a list and returns a reversed list. The `join(`*Tokens*`)` function concatenates

Tokens into a string, adding a white space between each two tokens. For example, the call `join(["Hello", "Picat", "World"])` returns the string `"Hello Picat World"`.

The built-in predicate `printf`, which performs formatted printing, is similar to the `printf` function in the C language. The format specifier `%s` is for printing strings, and the specifier `%w` is for printing a term of any type.

Assume that the program is stored in a file named `reverse_words.pi`, and that the file `reverse_words.in` contains the following sample input:

```
3
this is a test
foobar
all your base
```

The command

```
picat reverse_words < reverse_words.in
```

produces the following required output:

```
Case #1: test a is this
Case #2: foobar
Case #3: base your all
```

1.3 Store Credit

Store Credit is another easy practice problem.[4] A test case consists of an integer C, which is the store credit you receive, and a sequence of integers, which are prices of the available items. The output for a test case consists of the indices, i and j ($i < j$), of the two items whose prices add up to the store credit. It is assumed that each test case will have exactly one solution.

```
main =>
    T = read_int(),
    foreach (TC in 1..T)
        C = read_int(),
        N = read_int(),
        Items = {read_int() : _ in 1..N},
        do_case(TC, C, Items)
    end.

do_case(TC, C, Items),
   between(1, len(Items)-1, I),
   between(I+1, len(Items), J),
```

[4] https://code.google.com/codejam/contest/351101/dashboard#s=p0

```
   C == Items[I]+Items[J]
=>
   printf("Case #%w: %w %w\n", TC, I, J).
```

The function `read_int()` reads an integer from the standard input `stdin`. The `main` predicate reads `T`, the number of test cases. For each test case number `TC`, the `foreach` loop reads the store credit `C`, the number of available items `N`, and the sequence of prices of the items `Items`. For each test case, the predicate `do_case` searches for two indices, `I` and `J` (`I` $<$ `J`), that satisfies the condition `C == Items[I]+Items[J]`, and prints out the answer.

The expression {`read_int() : _ in 1..N`} is called an *array comprehension*, which returns an array consisting of `N` integers read from `stdin`. An array comprehension has the following general format:

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

where *Exp* is an expression, and the iterators and conditions have the same format as those used in the `foreach` loop. An array comprehension is a special functional notation for creating arrays. It includes *Exp* as an element in the array for each possible combination of values in the iterators that satisfies the conditions. Like a loop, an array comprehension also forms a name scope.

The array comprehension {`read_int() : _ in 1..N`} is equivalent to `[read_int() : _ in 1..N].to_array()`, which creates a list using a list comprehension, and converts the list to an array. Since the array comprehension creates a temporary list, the following code is more efficient:

```
Items = new_array(N),
foreach (I in 1..N)
    Items[I] = read_int()
end,
```

The function `new_array(N)` returns a new array of `N` elements. Initially, all the elements are distinct variables. The `foreach` loop fills in the array with integers from the input.

In Picat, the function `len(L)` returns the length of `L`, and the index operator `L[I]` returns the `I`th element of `L`. While `len(L)` and `L[I]` take constant time when `L` is an array, they take linear time when `L` is a list. For this reason the program uses an array, rather than a list, to store the prices.

The `do_case` predicate uses a *failure-driven loop* to enumerate `I`, over the range `1..len(Items)-1`, and `J`, over the range `I+1..len(Items)`, until a pair of indices is found that satisfies the condition `C == Items[I]+Items[J]`. This predicate implements a basic search technique, called *generate-and-test*. The predicate call `between(`*From*`,`*To*`,`*X*`)` is a *choice point*, which nondeterministically selects a value from the range *From*..*To* for *X*. It first binds *X* to *From*. When execution backtracks to the call, it binds *X* to *From* + 1 if *From* + 1 is not greater than *To*. Execution can backtrack to the call as long as there are untried values in the range. The call fails

when execution backtracks to it, and all values have been tried. When this calls fails, execution will continue to backtrack to another call that is a choice point.

The operator == tests if two terms are identical, and the operator = performs *unification* on two terms. The *unification* T_1 = T_2 is true if term T_1 and term T_2 can be made identical by binding some of the variables to values.

The do_case predicate can be implemented as follows using a foreach loop:

```
do_case(TC, C, Items) =>
    foreach(I in 1..len(Items)-1, J in I+1..len(Items))
        if (C == Items[I]+Items[J]) then
            printf("Case #%w: %w %w\n", TC, I, J)
        end
    end.
```

Nevertheless, this implementation is not as preferable as the failure-driven loop, because the foreach loop continues to check all the remaining pairs, even after a satisfying pair has been found. Picat does not provide statements like the break or return statements in procedural languages that can terminate loops early.

In this example, all three occurrences of %w in printf can be safely replaced by %d, because the integers all fit in 32 bits. However, for printing big integers, the specifier %w must be used.

The above program takes $O(n^2)$ time, where n is the number of items. It can be improved by using a map to speed up search. The following gives an improved version:

```
main =>
    T = read_int(),
    foreach (TC in 1..T)
        C = read_int(),
        N = read_int(),
        Items = {read_int() : _ in 1..N},
        Map = new_map(),
        foreach (I in N..-1..1)
            Is = Map.get(Items[I], []),
            Map.put(Items[I],[I|Is])
        end,
        do_case(TC, C, Items, Map)
    end.

do_case(TC, C, Items, Map),
   between(1, len(Items)-1, I),
   Js = Map.get(C-Items[I], []),
   member(J, Js),
   I < J
=>
   printf("Case #%w: %w %w\n", TC, I, J).
```

The function `new_map()` returns a new map. The function `put(`*Map*`, `*Key*`, `*Value*`)` puts the pair (*Key*, *Value*) into *Map*. The function `get(`*Map*`, `*Key*`, `*DefaultVal*`)` returns the value associated with *Key* in *Map*; it returns *DefaultVal* if *Map* does not contain *Key*.

The `foreach` loop below `new_map()` inserts a key-value pair for each price into the map, where the key is the price, and the value is a list of indices at which the price occurs in the array. Note that the loop iterates over the indices from `N` down to `1`. The indices associated with each price are added to the front of the list, from the largest to the smallest. In this way, the resulting list of indices for each price will be sorted in ascending order.

The `do_case` predicate does the following: For each `I` in `1..len(Items)-1`, and for each `J` in `Js` (which is the list of indices associated with the value `C-Items[I]`), if `I < J`, then (I, J) is a satisfying pair of indices. The call `member(J, Js)` nondeterministically selects a value from `Js` for `J`.

1.4 Minimum Scalar Product

This problem is from Round 1A 2008.[5] Given two vectors $v_1 = (x_1, x_2, \ldots, x_n)$ and $v_2 = (y_1, y_2, \ldots, y_n)$, the problem is to choose a permutation of v_1 and a permutation of v_2 such that the scalar product of these two permutations is the smallest possible, and output that minimum scalar product.

Like many other GCJ problems, this problem requires insightful reasoning. The brute-force approach that enumerates all of the permutations cannot be scaled to handle large vectors. Let $v_1 = (x_1, x_2)$ and $v_2 = (y_1, y_2)$. Assume $x_1 \leq x_2$ and $y_1 \leq y_2$. It is not difficult to prove that

$$x_1 \times y_2 + x_2 \times y_1 \leq x_1 \times y_1 + x_2 \times y_2.$$

In general, in order to get the minimum product, we can sort v_1 in ascending order and sort v_2 is descending order, and multiply the sorted vectors.

```
main =>
    T = read_int(),
    foreach (I in 1..T)
        do_case(I)
    end.

do_case(TC) =>
    N = read_int(),
    V1 = [read_int() : _ in 1..N].sort(),
    V2 = [read_int() : _ in 1..N].sort_down(),
    Prod = sum([E1*E2 : {E1,E2} in zip(V1,V2)]),
    printf("Case #%w: %w%n", TC, Prod).
```

[5] `https://code.google.com/codejam/contest/32016/dashboard#s=p0`

The `sort(L)` function returns a sorted list of L in ascending order, and the `sort_down(L)` function returns a sorted list of L is descending order. These sort functions can be utilized to sort a list of any terms. The expression

```
sum([E1*E2 : {E1,E2} in zip(V1,V2)])
```

gives the product of the two vectors `V1` and `V2`. The function `zip(V1,V2)` returns a zipped list of pairs from `V1` and `V2`. For example, `zip([1,2],[3,4])` returns `[{1,3},{2,4}]`. This expression sums `E1*E2` for each pair `{E1,E2}` in the zipped list of `V1` and `V2`. For this expression, the Picat compiler generates code for evaluating the expression without actually creating a zipped list or a list for the list comprehension.

Let's see how to implement the brute-force algorithm for the problem. We don't need to try all permutations of both vectors. We can fix v_1 and choose a permutation of v_2 such that the product of v_1 and the permutation is minimum. This brute-force algorithm is not efficient, but it can handle the small test.

```
import util.

main =>
    T = read_int(),
    foreach (I in 1..T)
        do_case(I)
    end.

do_case(Case) =>
    N = read_int(),
    V1 = [read_int() : _ in 1..N],
    V2 = [read_int() : _ in 1..N],
    minof(scalar_prod(V1,V2,Prod),Prod),
    printf("Case #%w: %w%n", Case, Prod).

scalar_prod(V1,V2,Prod) =>
   permutation(V2,V22),
   Prod = sum([E1*E2 : {E1,E2} in zip(V1,V22)]).
```

The predicate `permutation(V2,V22)`, which is defined in the `util` module, nondeterministically binds `V22` to a permutation of `V2`. For a permutation `V22` of `V2`, the predicate `scalar_prod(V1,V2,Prod)` binds `Prod` to the product of `V1` and `V22`. Since the `permutation` predicate is nondeterministic, the `scalar_prod` predicate is also nondeterministic. The built-in predicate `minof(scalar_prod(V1,V2,Prod),Prod)` returns an instance of the predicate call `scalar_prod(V1,V2,Prod)` that has the smallest `Prod`.[6]

[6]The `minof` predicate, which takes another predicate call as the first argument, is called a higher-order predicate. Picat provides several higher-order built-ins. For example, `maxof(`*Goal*`,`*Exp*`)`, `find_all(`*Template*`,`*Goal*`)`, and `count_all(`*Goal*`)`.

It is also possible to iterate over all of the permutations to find the best permutation that gives the minimum product.[7] Nevertheless, the backtracking-based approach is more memory efficient than the iterative approach, since it does not use any memory to store all the permutations.

The following shows how the `permutation` predicate is implemented in Picat:

```
permutation([], P) => P = [].
permutation(L, P) =>
    P = [X|P1],
    select(X, L, L1),
    permutation(L1, P1).

select(X, [Y|L], L1) ?=> Y = X, L1 = L.
select(X, [Y|L], L1) => L1 = [Y|L2], select(X, L, L2).
```

This implementation utilizes pattern-matching rules, where the heads contain non-variable patterns. The first rule states that the permutation of `[]` is `[]`. For a non-empty list `L`, the second rule is applied. The call `P = [X|P1]` binds `P` to the list constructed by the *cons* operator `[X|P1]`. The call `select(X, L, L1)` nondeterministically selects an element `X` from `L`, resulting in a new list `L1`. The last call `permutation(L1, P1)` generates a permutation `P1` of `L1`.

The implementation of `select` uses a *backtrackable* rule, as denoted by the operator `?=>`. Because of the use of this backtrackable rule, this predicate becomes *nondeterministic*, and it is able to return multiple answers. For example:

```
Picat> select(X,[1,2,3],L1)
X = 1
L1 = [2,3] ?;
X = 2
L1 = [1,3] ?;
X = 3
L1 = [1,2] ?;
no
```

After Picat returns an answer, you can type a semicolon immediately after the answer to let the system backtrack; the system reports `no` if no answer remains.

1.5 Alien Numbers

This is Problem A in the set of practice problems.[8] The objective of the problem is to convert a number from one alien numeral system, called the source language, to another alien numeral system, called the target language. Each numeral system consists of a set of "digits", and the size of the set is the base.

For a number in the source language, the conversion can be done in two steps:

[7] The function `permutations(`*L*`)`, which is defined in the `util` module, returns a list of permutations of *L*.

[8] `https://code.google.com/codejam/contest/32003/dashboard#s=p0`

first convert the number to a decimal number, and then convert the decimal number to the target language. For each language, we use a map in order to map the digits to their values, the first digit to 0, the second digit to 1, and so on.

```
import util.  % use split

main =>
    T = to_int(read_line()),
    foreach (TC in 1..T)
        [Num,SDs,TDs] = read_line().split(),
        do_case(TC, Num, SDs, TDs)
    end.

do_case(TC, Num, SDs, TDs) =>
    SMap = new_map(),
    SBase = len(SDs),
    foreach ({D, DVal} in zip(SDs, 0..SBase-1))
        SMap.put(D,DVal)
    end,
    source_to_decimal(Num, SBase, SMap, 0, SVal),
    %
    TMap = new_map(),
    TBase = len(TDs),
    foreach ({D, DVal} in zip(TDs, 0..TBase-1))
        TMap.put(DVal,D)
    end,
    decimal_to_target(SVal, TBase, TMap, TNum),
    printf("Case #%w: %s\n", TC, TNum).

source_to_decimal([], _Base, _Map, Val0, Val) => Val = Val0.
source_to_decimal([D|Ds], Base, Map, Val0, Val) =>
    source_to_decimal(Ds, Base, Map, Val0*Base+Map.get(D), Val).

decimal_to_target(0, _Base, Map, Num) => Num = [Map.get(0)].
decimal_to_target(Val, Base, Map, Num) =>
    Ds = [],
    while (Val !== 0)
        DVal := Val mod Base,
        Val := Val div Base,
        Ds := [Map.get(DVal)|Ds]
    end,
    Num = Ds.
```

Each test case consists of a number string `Num`, a list of digits `SDs` in the source language, and a list of digits `TDs` in the target language. For the source language, the program uses `SMap` to map the digits to the values, and stores the base in `SBase`.

Let $[D_{n-1}, D_{n-2}, \ldots, D_1, D_0]$ be a number string of the source language that has

the base B. This string represents the decimal value: $D_{n-1} * B^{n-1} + D_{n-2} * B^{n-2} + \ldots + D_1 * B^1 + D_0$. The predicate `source_to_decimal(Num, Base, Map, Val0, Val)` uses tail recursion to convert the number string `Num` into decimal: If `Num` is empty, then the result `Val` is bound to the accumulator `Val0`; otherwise, if `Num` is a list `[D|Ds]`, then it recurses on `Ds` using `Val0*Base+Map.get(D)` as the new accumulator value. The accumulator value in the initial call to `source_to_decimal` is 0.

Let $D_{n-1} * B^{n-1} + D_{n-2} * B^{n-2} + \ldots + D_1 * B^1 + D_0$ be the decimal value and B be the base of the target language. The digits can be extracted using the *divide-by-base* algorithm. When the value is divided by the base B, the remainder is D_0, and the quotient is $D_{n-1} * B^{n-2} + D_{n-2} * B^{n-3} + \ldots + D_1$. This division step is repeatedly applied to the value until the value becomes 0. The predicate `decimal_to_target(Val, Base, Map, Num)` converts the decimal value `Val` to a number string of the target language. If `Val` is 0, then the string only consists of the 0-value digit. Otherwise, the predicate uses the divide-by-base algorithm to extract the digits.

The predicate `decimal_to_target` illustrates the use of the `while` loop and the assignment operator `:=` in Picat. A while loop takes the form

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

It repeatedly executes *Goal* as long as *Cond* succeeds. For the assignment X `:=` *Exp*, Picat introduces a new variable to hold the value of *Exp*; after that, this new variable replaces all of the occurrences of X in the scope. Because of variable cloning, no values can be returned using assignments. For example, if the unification `Num = Ds` in the `decimal_to_target` predicate were changed to `Num := Ds`, then the result would never be returned to the caller through the variable `Num`.

1.6 Alien Language

Alien Language involves matching words in an alien language against patterns.[9] A pattern consists of tokens, where each token is either a single lowercase letter or a group of unique lowercase letters surrounded by parentheses `(` and `)`. For example: `(ab)d(dc)` means the first letter is either `a` or `b`, the second letter is definitely `d`, and the last letter is either `d` or `c`. Therefore, the pattern `(ab)d(dc)` can stand for any one of these 4 possibilities: `add`, `adc`, `bdd`, `bdc`. Each test case is a pattern. The output for the case indicates how many of the given words match the pattern.

The problem can be solved by pattern matching. For a letter in a word, if the token is also a letter, then the match succeeds iff the two letters are identical; otherwise, if the token is a group, then the match succeeds iff the letter is included in the group.

[9] `https://code.google.com/codejam/contest/90101/dashboard#s=p0&a=1`

```
import util.

main =>
    [_L,D,T] = [to_int(W) : W in read_line().split()],
    Words = [read_line() : _ in 1..D],
    foreach(TC in 1..T)
        do_case(TC, Words)
    end.

do_case(TC, Words) =>
    trans_pattern(read_line(), P),
    printf("Case #%w: %w%n", TC,
                              sum([1 : Word in Words, match(Word, P)])).

trans_pattern([], P) => P = [].
trans_pattern(['('|S], P) =>
    P = [G|PR],
    trans_pattern_group(S, SR, G),
    trans_pattern(SR, PR).
trans_pattern([X|S], P) =>
    P = [X|PR],
    trans_pattern(S, PR).

trans_pattern_group([')'|S], SR, G) =>
    G = [], S = SR.
trans_pattern_group([X|S], SR, G) =>
    G = [X|GR],
    trans_pattern_group(S, SR, GR).

match([], []) => true.
match([A|As], [A|Ps]) =>
    match(As, Ps).
match([A|As], [L|Ps]), member(A,L) =>
    match(As, Ps).
```

The first line in the body of the `main` predicate reads three integers from the input: the length of each of the words `_L`, the number of words `D`, and the number of test cases `T`. The value `_L` is not used later in the program.[10] The list comprehension `[read_line() : _ in 1..D]` reads `D` lines into a list. For each test case, the `do_case` predicate reads the pattern, transforms the pattern into a list, and counts the words that match the pattern.

The predicate `trans_pattern(S, P)` transforms the pattern string `S` into a list `P`. A letter is copied into the list. For a group that is surrounded by parentheses, the call `trans_pattern_group(S, SR, G)` extracts the letters from the group and puts them into the list `G`; `SR` holds the remainder of `S` after the extraction. For example, for the pattern `"(ab)d(dc)"`, the list obtained after transformation is

[10] Picat does not issue singleton variable warnings for variable names that begin with the underscore _.

`[[a,b],d,[d,c]]`. The matching of a word against a pattern is done by the `match` predicate.

Since a group is represented as a list, it takes $O(n)$ time to check if a letter is in a group of size n. The above program can be improved by using a set for each group.

```
trans_pattern([], P) => P = [].
trans_pattern(['('|S], P) =>
    P = [G|PR],
    G = new_set(),
    trans_pattern_group(S, SR, G),
    trans_pattern(SR, PR).
trans_pattern([X|S], P) =>
    P = [X|PR],
    trans_pattern(S, PR).

trans_pattern_group([')'|S], SR, _G) =>  S = SR.
trans_pattern_group([X|S], SR, G) =>
    G.put(X),
    trans_pattern_group(S, SR, G).

match([], []) => true.
match([A|As], [P|Ps]), atom(P) =>
    A == P,
    match(As, Ps).
match([A|As], [G|Ps]), G.has_key(A) =>
    match(As, Ps).
```

The call `new_set()` returns a new empty set. For each pattern group that begins with `'('`, the call `trans_pattern_group(S, SR, G)` adds every letter `X` in the group into set `G` using the function `G.put(X)`. The `match` predicate uses `G.has_key(A)` to test if letter `A` is in group `G`.

1.7 Egg Drop

Egg Drop is an optimization problem that involves three parameters: the number of floors F in a building, the number of drops D that you are allowed to perform, and the number of eggs B that you can break.[11] You are assumed to have at least D eggs. All eggs are identical in terms of the shell's strength. If an egg breaks when dropped from floor i, then all eggs are guaranteed to break when dropped from any floor $j \geq i$. Likewise, if an egg doesn't break when dropped from floor i, then all eggs are guaranteed to never break when dropped from any floor $j \leq i$. For each floor in the building, you want to know whether or not an egg dropped from that floor will break.

[11] `https://code.google.com/codejam/contest/32003/dashboard#s=p2`

The problem can be posted in three different ways, depending on which parameter is to be optimized. The first variant is to determine the maximum number of floors that can be examined when D and B are given. If $D = 0$ or $B = 0$, then no floors can be examined, so $F = 0$. If $B = 1$, then what you can do is to try the floors, starting at floor 1, until the egg breaks or you have dropped D times; so $F = D$. In general, let $f(D,B)$ be the number of floors that can be examined with D drops and B breaks. There are two possible outcomes when dropping an egg from floor k, an optimal floor number to start. If the egg breaks, then the $k-1$ floors that are below floor k need to be examined, and the number of remaining breaks becomes $B-1$. If the egg does not break, then the floors above floor k need to be examined, and the number of remaining breaks remains to be B. The function can be defined recursively as:

```
f(0, _) = 0.
f(_, 0) = 0.
f(D, B) = f(D-1, B) + f(D-1, B-1) + 1.
```

This function grows exponentially, and dynamic programming can be used to speed up the computation. Since the problem requires outputting -1 if the value is greater than or equal to 2^{32} for given B and D, calls with large arguments are guaranteed to return -1, and therefore do not need to be tabled.

The second variant of the problem is to find the minimum number of drops D given F and B, and the third variant is to find the minimum number of breaks B given F and D. These variants can also be solved using dynamic programming. However, since the input values can be as large as 2 billion, the dynamic programming approach is not feasible. A more efficient approach is to use binary search to find the smallest value for which the F floors can be examined.

```
main =>
    T = read_int(),
    foreach (TC in 1..T)
        F = read_int(), D = read_int(), B = read_int(),
        do_case(TC, F, D, B)
    end.

do_case(TC, F, D, B) =>
    MF = max_f(D, B),
    min_d(F, MD, B),
    min_b(F, D, MB),
    printf("Case #%w: %w %w %w\n", TC, MF, MD, MB).

% maximize F for given D and B
max_f(D, B) = F, D >= 100000, B >= 2 =>  F = -1.
max_f(D, B) = F, D >= 10000, B >= 3 =>  F = -1.
max_f(D, B) = F, D >= 1000, B >= 4 =>  F = -1.
max_f(D, B) = F, B > D => F = max_f(D, D).
max_f(D, B) = f(D, B).
```

```
table
f(_, 0) =  0.
f(D, 1) =  D.
f(0, _) =  0.
f(1, _) =  1.
f(D, B) =  F =>
    F1 = f(D-1,B),
    F2 = f(D-1,B-1),
    if F1 == -1 ; F2 == -1 then
        F = -1
    else
        F0 = F1+F2+1,
        F = cond(F0 >= 2**32, -1, F0)
    end.

% minimize D for given F and B
min_d(F, D, B) =>
    bsearch_d(0, F, F, D, B).

bsearch_d(From, To, F, D, B), From >= To =>
    D = cond((max_f(From, B) >= F ; max_f(From, B) == -1), From, From+1).
bsearch_d(From, To, F, D, B) =>
    Mid = (From+To) div 2,
    if max_f(Mid, B) == F then
        D = Mid
    elseif max_f(Mid, B) == -1 ; max_f(Mid, B) > F then
        bsearch_d(From, Mid-1, F, D, B)
    else
        bsearch_d(Mid+1, To, F, D, B)
    end.

% minimize B for given F and D
min_b(F, D, B) =>
    bsearch_b(0, F, F, D, B).

bsearch_b(From, To, F, D, B), From >= To =>
    B = cond((max_f(D, From) >= F ; max_f(D, From) == -1), From, From+1).
bsearch_b(From, To, F, D, B) =>
    Mid = (From+To) div 2,
    if max_f(D, Mid) == F then
        B = Mid
    elseif max_f(D, Mid) == -1 ; max_f(D, Mid) > F then
        bsearch_b(From, Mid-1, F, D, B)
    else
        bsearch_b(Mid+1, To, F, D, B)
    end.
```

The function `max_f(D, B)` returns the maximum number of floors that can be ex-

amined with D drops and B breaks. It returns -1 for certain combinations of values of D and B, where f(D, B) $\geq 2^{32}$. It is necessary to filter out these cases because the function f(D, B) takes O(D×B) table space.

In the implementation of function f(D, B), the values from f(D-1, B) and f(D-1, B-1) are combined in such a way that -1 is returned if either value is -1 or if f(D-1, B) + f(D-1, B-1) + 1 is greater than or equal to 2^{32}.

The min_d(F, D, B) and min_b(F, D, B) predicates implement binary search for finding the minimum D and the minimum B, respectively. In Picat, $(A;B)$ is a disjunction, (A,B) is a conjunction, and cond(C, A, B) is a conditional expression, which gives the value of A if C if true and the value of B if C is false. Since ';' has lower precedence than ',', C must be parenthesized if it is a disjunction.

1.8 Always Turn Left

This is Problem B in the set of practice problems.[12] Unlike a typical maze problem, where the objective is to find a path from the entrance to the exit in a given maze, the objective of this problem is to figure out the configuration of the maze for two given "always-turn-left" paths: one path is from the entrance to the exit, and the other is from the exit to the entrance. It is known that the maze is "perfect", meaning that it can always be solved using the "always turn left" algorithm. It is also known that the maze is a rectangular grid of rooms and the entrance is on the north wall. However, the size of the grid, the column number of the entrance, and the position of the exit are unknown. Your program has to compute these pieces of missing information, in addition to each room's configuration.

A given path is described with three characters: 'W' means to walk forward into the next room, 'L' means to turn left (or counterclockwise) 90 degrees, and 'R' means to turn right (or clockwise) 90 degrees. Initially, you are assumed to face south, since the entrance is on the north wall. For example, for the two paths: "WRWWLWWLWWLWLWRRWRWWWRWWRWLW" and "WWRRWLWLWWLWWLWWRWWRWWLW", the maze will have the configuration shown in Figure 1.2.

We can solve the problem by walking along the paths and recording the configurations of the visited rooms. The size of the maze is unknown, so we need to use a size that is big enough for the tests. The column number of the entrance is also unknown, so we need to guess it.

```
import util.

main =>
    T = to_int(read_line()),
    foreach (TC in 1..T)
        [FPath,BPath] = read_line().split(),
        once do_case(TC, FPath, BPath)
```

[12] https://code.google.com/codejam/contest/32003/dashboard#s=p1

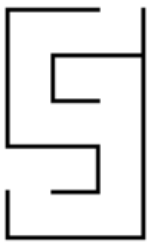

Figure 1.2: A maze

```
    end.

do_case(TC, [_|FPath], [_|BPath]) =>
    Maze = new_array(100,100),
    between(1, 100, C),
    record_room(Maze[1,C], north),
    walk(Maze, 1, C, EndR, EndC, south, Dir, FPath),   % begin at <1,C>
    walk(Maze, EndR, EndC, 1, C, opp(Dir), _, BPath),  % walk back
    printf("Case #%w:\n", TC),
    output(Maze).

walk(Maze, R, C, EndR, EndC, Dir0, Dir, [_]) =>  % the last walk
    record_room(Maze[R,C], Dir0),
    EndR = R, EndC = C, Dir = Dir0.
walk(Maze, R, C, EndR, EndC, Dir0, Dir, ['W'|Path]) =>
    Dir1 = opp(Dir0),
    next_pos(R, C, Dir0, R1, C1),
    record_room(Maze[R,C], Dir0),
    record_room(Maze[R1,C1], Dir1),
    walk(Maze, R1, C1, EndR, EndC, Dir0, Dir, Path).
walk(Maze, R, C, EndR, EndC, Dir0, Dir, ['L'|Path]) =>
    Dir1 = left(Dir0),
    walk(Maze, R, C, EndR, EndC, Dir1, Dir, Path).
walk(Maze, R, C, EndR, EndC, Dir0, Dir, [_|Path]) =>   % turn right
    Dir1 = right(Dir0),
    walk(Maze, R, C, EndR, EndC, Dir1, Dir, Path).

left(north) = west.
left(south) = east.
left(west) = south.
left(east) = north.

right(Dir) = opp(left(Dir)).

opp(north) = south.
```

```
opp(south) = north.
opp(west) = east.
opp(east) = west.

next_pos(R, C, south, R1, C1) => R1 = R+1, C1 = C, R1 =< 100.
next_pos(R, C, north, R1, C1) => R1 = R-1, C1 = C, R1 >= 1.
next_pos(R, C, west, R1, C1) => R1 = R, C1 = C-1, C1 >= 1.
next_pos(R, C, east, R1, C1) => R1 = R, C1 = C+1, C1 =< 100.

record_room(Room, north) => Room = [y,_,_,_].
record_room(Room, south) => Room = [_,y,_,_].
record_room(Room, west) => Room = [_,_,y,_].
record_room(Room, east) => Room = [_,_,_,y].

output(Maze) =>
    foreach(R in 1..len(Maze))
        foreach(C in 1..len(Maze[1]))
            Room = Maze[R,C],
            if nonvar(Room) then
                bind_vars(Room, n),   % change vars to n
                printf("%w", maze_code(Room))
            end
        end,
        if nonvar(Maze[R,1]) then nl end
    end.

maze_code([y,n,n,n]) = 1.
maze_code([n,y,n,n]) = 2.
maze_code([y,y,n,n]) = 3.
maze_code([n,n,y,n]) = 4.
maze_code([y,n,y,n]) = 5.
maze_code([n,y,y,n]) = 6.
maze_code([y,y,y,n]) = 7.
maze_code([n,n,n,y]) = 8.
maze_code([y,n,n,y]) = 9.
maze_code([n,y,n,y]) = a.
maze_code([y,y,n,y]) = b.
maze_code([n,n,y,y]) = c.
maze_code([y,n,y,y]) = d.
maze_code([n,y,y,y]) = e.
maze_code([y,y,y,y]) = f.
```

The two paths are read as two strings, `FPath` and `BPath`. The first step in the paths is always `'W'`, which indicates entering the maze, and the last step is also always `'W'`, which indicates stepping out the maze. These two steps are disregarded. The `do_case` predicate creates a two-dimensional array of size 100 by 100, which is big enough for passing the large test. Initially, all the entries of the array are variables. The call `between(1, 100, C)` selects a column number `C` in the range from 1 to

100. Execution will backtrack to this call if either of the walks fails because a step crosses the boundaries of the maze. The call `record_room(Maze[1,C], north)` records the fact that the first room is open in the north.

The predicate `walk(Maze, R, C, EndR, EndC, Dir0, Dir, Path)` processes `Path`, starting at the position `(R,C)` and the direction `Dir0`. After the predicate succeeds, `EndR` and `EndC` will be bound to the row and column numbers, respectively, of the last room that was visited, and `Dir` will be bound to the direction in which the last room was entered. For each step, if it is `'W'`, then the predicate `next_pos` computes the position of the next room, and the walk continues from that position in the same direction. Note that the walk cannot continue if the position is outside of the array bounds. In that case, the program will backtrack to `between(1, 100, C)` to select the next column number `C`. If the step is `'L'` or `'R'`, then the walk continues from the same room but in a new direction.

After processing the two paths, the `do_case` predicate calls `output` to print out the room configurations in the required format. For each row number `R` and column number `C`, the entry `Maze[R,C]` indicates a visited room if it is a list of the form `[N,S,W,E]`, where each element is either `y`, indicating that there is a door, or a free variable, indicating that there is no door in that direction. The call `bind_vars(Room, n)` binds all the free variables in `Room` to `n`. The `maze_code` function converts a room configuration to the required code using pattern matching.

1.9 Acknowledgement

Two of the examples (**Minimum Scalar Product** and **Alien Language**) were originally written by Mike Bionchik. The author would like to thank Sergii Dymchenko for bring GCJ to his attention, and the following people for giving very helpful comments on early drafts of this article: Roman Barták, Peter Bernschneider, Mike Bionchik, Jonathan Fruhman, Håkan Kjellerstrand, Annie Liu, Claudio Cesar de Sá, and Bo Yuan (Bobby) Zhou.