Chapter 1: Introduction to programming

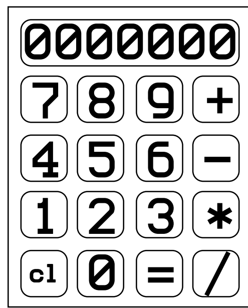
Computational processes

Computer science is the study of computational processes. A process is an entity that exhibits a specific behaviour over time. In computational processes however, time advances in discrete steps. Science in general provides a mechanism for describing the behaviour of a process using some language. Describing a computational process involves specifying how it evolves during its successive discrete time steps. This is generally called programming and the language used is called a programming language. Typically, the behaviour of a computational process can be simulated using some machine to interpret the program describing it.

Consider a simple computational process that consists of evaluating an arithmetic expression:

```
(4+1)*2/5
```

Typically we will be using a pocket calculator:



to interpret a program

cl 4 + 1 * 2 / 5 =

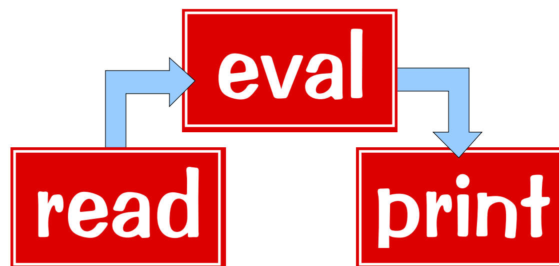
expressed as an acceptable sequence of buttons to be pressed. In doing so we respect the language that the calculator accepts; hence we can simulate the computation by entering this program on the calculator to obtain its value:

0000002

Computer science gives a lot of attention to the expressiveness of programming languages. Typically, we are interested in describing as wide a range of processes as possible while using a language which is as simple as possible. In this chapter we will introduce a simple yet powerful language called Pico[1] to help us explore the concept of computation.

Interpretation of programs

We will first focus on the inner workings of a computer, that is a machine capable of simulating a computational process by interpreting a program that describes it. In order to separate the different concerns, we will view a computer as consisting of three collaborating components:



The core of the computer is the evaluator **eval** which maps expressions onto their values:

eval : { expression } (∅ { value }

The evaluator is the actual embodiment of the *meaning* that must be attributed to a program. We will use the two terms *expression* and *value* to refer to the machine's internal representation of a program and to the result of its interpretation. In the next chapter we shall see that values are exactly those expressions that are neutral to the **eval** function. The **read** and **print** components map text onto expressions and values onto text:

read : { character } (∅ { expression }

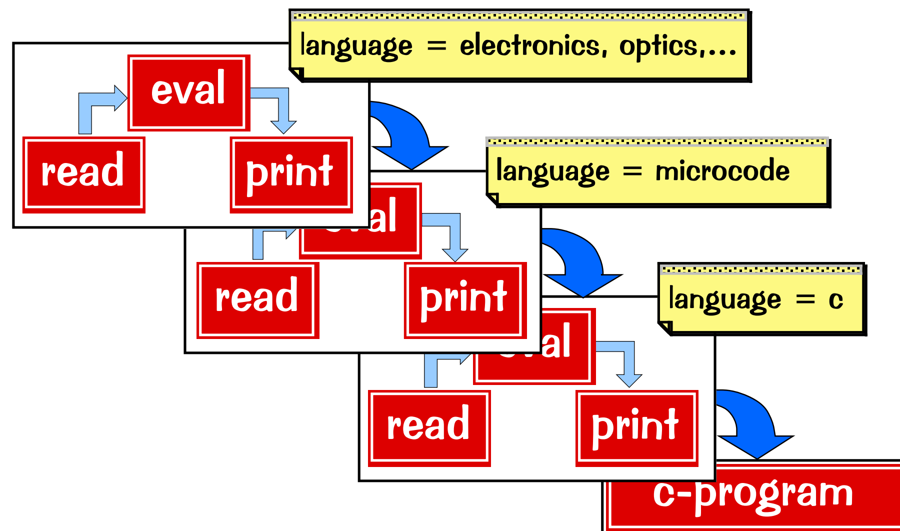
print : { value } (∅ { character }

---

[1] pico stands for 10^{-12} and hence very small; the name has no other meaning

In doing so, we distinguish between how we as human beings represent information —using strings of characters from some alphabet— and how this is done by the machine performing the actual simulation.

In the real world, a computer is not an abstract concept but a physical entity that is itself —strangely enough— a computational process. A pocket calculator for instance contains a minuscule electronic device that is driven by a behaviour as laid down in some program. This leads us to the slightly surprising consideration that any computer that we are physically confronted with sits at the bottom of a tower of machines:



Every level consists of a **read-eval-print** machine (**REP** for short) capable of interpreting a program in a particular language. In modern personal computers we can easily distinguish a tower consisting of three tiers of processes:

- a machine which is described by a geometrical pattern etched into the silicon surface of a microchip; this machine accepts programs typically accessible via languages such as Basic, C or Pascal
- a machine which is described by the composition of electronic switches built out of a large number of transistors; this machine accepts programs written in the language of digital logic
- a machine which is described by the physical laws that govern the behaviour of semiconductor devices.

Each of these three tiers has changed over time and will continue to do so. At one time a typical computer consisted of vacuum tubes wired together in an organised way and governed by how electrons behave in a vacuum when exposed to an electrical field. Future generations of computers will probably exploit

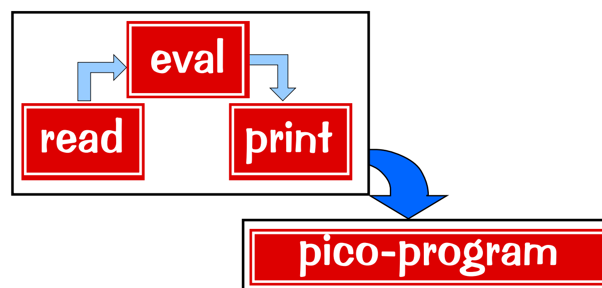
the behaviour of photons inside optical conduits. These examples go to show that it is necessary to make abstraction of the physical implementation of computational devices and to concentrate on the computational processes themselves.

Metacircularity

Programming is the preferred road to understanding computational processes. In order to build programs one has to know the language that is to be used; the most productive approach to do so is understanding the evaluation mechanism introduced in the previous section.

Computer science regularly uses *metacircularity* to define and hence to understand some computational mechanism. In this context metacircularity means that the same language is used to describe both a computational process and the **REP** machine used to simulate it. The prefix *meta* refers to the fact that we use a program that manipulates programs; circularity means that the same language is used on both levels. We will regularly use the qualification meta-level and base-level to make the distinction between these two levels.

Metacircularity is of course a theoretical concept but it enables us to study computational processes using a one-tier tower of evaluators. In the next chapter we shall formally define the programming language called Pico in exactly this fashion: each and every instance of a Pico-expression will be defined by a Pico-program capable of computing its value.



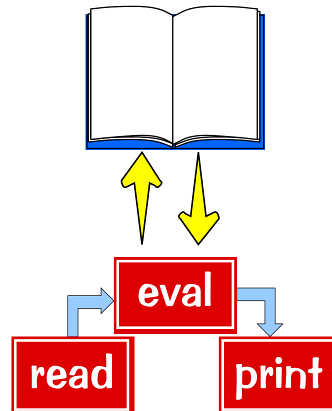
In order to prime our knowledge, we will use the rest of this chapter to give an informal introduction to the language Pico.

Pico basics

In this chapter we will ignore the exact structure of Pico expressions and values and only use their textual representations as accepted and generated by the **read** and **print** components. This textual representation has been

chosen to be as close as possible to conventional notation from calculus, taking into account the limitations on text formats imposed by personal computers and pocket calculators.

Central to the evaluation of Pico programs is the notion of a dictionary. In the schematic below, a dictionary is represented by a double-entry book-like structure that can be consulted, modified and extended by the **eval** component.



The dictionary binds values to variables. As before, a value is the result obtained when an expression is evaluated; a variable is a new concept: it is an name that may be used within a Pico-program. It enables us to identify and reference values as they are computed.

Examples of variable names are:

```
Prime        *        Divide_by_2
```

Examples of values as generated by the **print** component are

```
123   4.75   "Monday"   [1,2,3]   <function f>
```

The exact syntax of variables or values is not relevant at this point.

A dictionary functions as a kind of memory for a computational process. It is generally used to store values identified by the name they are bound to and to retrieve or even change these values as time evolves.

The various operations on dictionaries will be formally defined in the course of next chapter.

Numbers, fractions and text

We will start by having a look at the simplest kinds of expressions: numbers, fractions and text. When submitted to the Pico machine, these expressions —sometimes called *literals*— are identical to their values. For instance:

REP(123) ≡ **123**

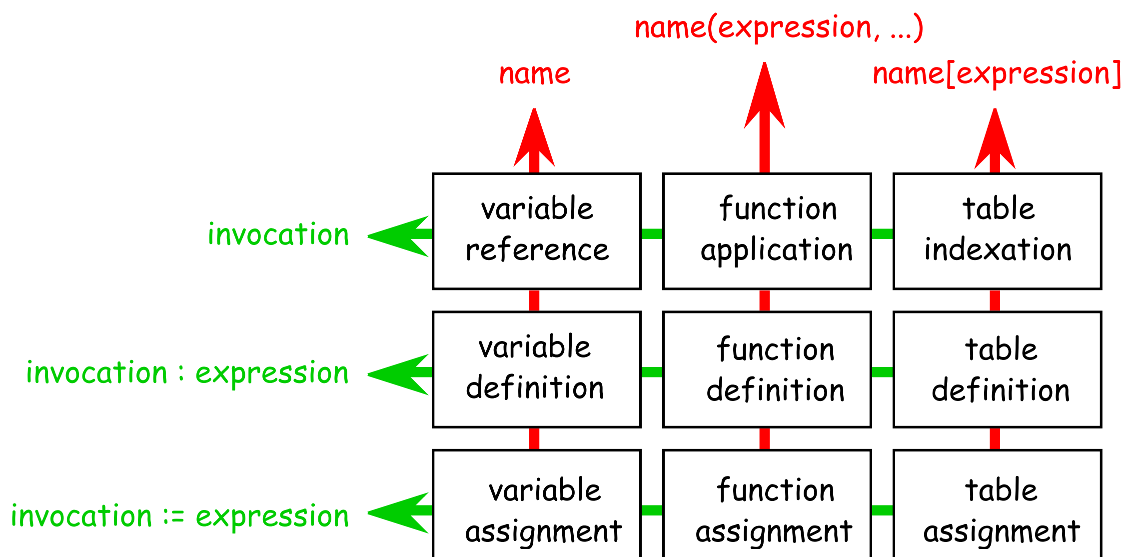
REP(4.75) ≡ **4.75**

REP("Monday") ≡ **Monday**

While the meaning of numbers or fractions is obvious, the notion of text deserves some explanation. Text is the mechanism used to introduce *symbolic* —as opposed to *numeric*— values; any sequence of keystrokes[2] delimited by either apostrophes (') or quotes (") is acceptable as text. Programmers usually have text in their programs to represent symbolic data or to interact with the outside world using symbols whenever numbers are not sufficient; in this case the **print** component strips the delimiters from the text value.

Grammatical rules

In Pico, syntax is kept to a strict minimum. Actually, in addition to the syntax of numbers, fractions and text, only 3x3 = 9 rules need to be memorised in order to master Pico's essential syntax:



[2] actually excluding apostrophe or quote depending on the delimiters used

This table lists the possible *invocations —references*, *applications* and *indexations*— from left to right; from top to bottom is listed how these invocations can be used inside an expression. Each of the nine combinations has a unique label that will be used throughout the rest of this text.

We shall now proceed by describing the meaning of these grammatical rules in an informal way.

Variable definition

The first expression type we will look into is the one allowing us to introduce —we say define— new variables. We are required to state a variable name and an expression which is used to compute a value to be bound to this name. The colon between the name and the expression is a necessary hint for Pico to interpret this expression as a variable definition:

name : expression

As an example consider the following:

```
pi: 4*arctan(1)
```

Its evaluation appends an entry for the variable `pi` and the value `3.14159`… to the current dictionary. The value of a variable definition is identical to the value of the expression following the colon.

Variable reference

Once a new variable has been introduced by means of a variable definition, we are able to reference it:

name

The value of a variable reference is the value bound to the variable name in the dictionary. Therefore the following:

```
pi
```

should result in the retrieval of the value `3.14159`… computed in the previous paragraph. In the absence of a dictionary entry for the variable name an error is generated; in case the same name is used for several variables, the most recent version is used.

Variable assignment

An existing variable can at all times be bound to a new value by means of a variable assignment:

name := expression

The following:

```
pi := 3.14159265358979
```

will result in the originally computed value of `pi` to be replaced by a —possibly— more accurate approximation of π. Clearly, a variable assignment can only take place if the variable has been defined previously; if this is not the case, an error is generated; in case the same name is used for several variables, the most recent version is used.

Using variables

Consider the following *transcript*, resulting from an interaction with a Pico-machine:

```
pi: 4*arctan(1)
:3.14159
x: pi
:3.14159
zero:= 0
undefined identifier: zero
x:= 0
:0
abc: xyz
undefined identifier: xyz
```

transcript

A line preceded by a colon is the actual output from the **print** component. We can see how the variable pi is computed as an approximation to π, and then used to initialise a variable x. Assigning the value 0 to a variable zero fails because zero hasn't been defined yet; the same operation applied to x succeeds. Finally, defining the variable abc fails because the expression to the right of the colon references a non-existent variable xyz. Note that as each of the valid expressions is entered, a value is printed out; the effects on the current dictionary are only noticeable through the availability of new variables as the evaluation proceeds.

Function definition

Functions can be defined by using the colon to affix a function *prescription* to a function invocation. A prescription is a valid expression used to specify the future behaviour of the function; the invocation must specify the function name and a parameter list consisting of any number of parameter names[3] separated by comma's and delimited by ordinary parentheses.

name(name,...): expression

The following example is a valid function definition:

```
f(x,y,z): x*(y+z)
```

We can easily see that a function named f is introduced featuring three parameters called x, y and z. The prescription states that an application of this function will add the values bound to y and z and return this sum multiplied by the value of x. Pico comes with most of the essential functions already defined; these will be enumerated in later paragraphs. Actually, a function definition introduces a new variable in the current dictionary; instead of a number, fraction or text, a value representing a function (or function for short) is bound to this variable. Functions have no grammatical representations, which is why the **print** component displays the value of f[4] as follows:

```
<function f>
```

[3] Until further notice parameters are limited to names

[4] Native functions such as sqrt are printed as <native function sqrt>

Function application

Any function bound to a variable in the current dictionary is candidate to be applied:

name(expression,...)

An application identifies the function by name and further specifies a number of *arguments* separated by comma's and delimited by ordinary parentheses. Any valid expression can be used as argument; their number should correspond to the number of parameters of the function bound to the variable named in the application.

Consider as an example the following function application:

```
f(1+2,3,2*2)
```

which our intuition tells us should produce the value 21. In the next chapter the mechanism of function application will be described formally; for the time being we will limit ourselves to an informal introduction.

Function application starts by looking up the function name in the dictionary; if it isn't found or if its value is different from a function, an error is generated. In the other case the values of arguments are computed and temporarily —that is for the duration of the function application— bound to the corresponding parameters, after which the value of the prescription is computed and returned.

```
f(x,y,z): x*(y+z)
:<function f>
f(1+2,3,2*2)
:21
g(1,2,3)
undefined identifier: g
pi(1,2,3)
not a function: pi
f(1,2)
non-matching argument list: f
```

transcript

In the preceding transcript various transactions involving function application are illustrated. Note that it is impossible to apply a function unless it has been properly defined with the correct number of parameters.

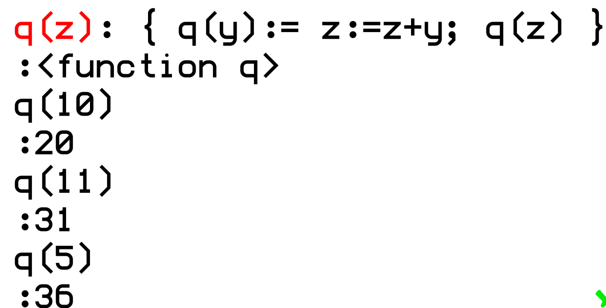
Function assignment

Any previously defined function is accessible via the variable that its name refers to. This variable —and any other variable for that matter— can be changed through ordinary variable assignment or through function assignment:

name(expression,...):= expression

The effect is identical to that of function definition except for the fact that no new variable is introduced. Note that any previous value —function or otherwise— is discarded.

At this stage function assignment is not really an essential feature of Pico. Nevertheless, in order to illustrate future use, the more adventurous reader is invited to examine the following transcript:



Note that the definition of the function q contains a prescription that starts by redefining itself and then applying the new definition of q to the original parameter z (the curly braces and the semicolon serve to group multiple expressions, and will be defined later on). Hence the first application changes the definition of q to:

```
q(y):= z:=z+y
```

and therefore returns the value 20. The next two applications q(11) and q(5) however, use this redefined version of q. The interesting part about

this example is the use of z as a kind of memory for the function q making it behave like an accumulator.

Table definition

Pico allows values to be combined into *tables*. Tables —very similar to vectors from mathematics— are defined by using an indexed invocation:

name[expression]: expression

Tables are named and this name is used to define a variable in the current dictionary. The square brackets are a hint needed by Pico to indicate that a table is considered; they delimit a *size*. Any expression with a non-negative number value can be used as a valid table size. The freshly defined variable will be bound to a table, that is a sequence of value slots. The number of slots is defined by the indicated size (and might be zero). Each slot can hold an arbitrary value, which is initially set to a newly computed value of the expression to the right of the colon. Therefore, the following expression:

```
T[10]: 0
```

introduces a variable named T bound to a table containing ten slots each containing the value 0. The following transcript illustrates a more sophisticated use of tables:

```
T[10]: 0
:[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
n: 5
:5
P[n]: n:=n-1
:[4, 3, 2, 1, 0]
```

transcript

Note that the value of a table definition is the table itself; its slots are displayed by the **print** component as a list delimited by square brackets and separated by comma's. Also note that the slot values of P are computed from left to right using the expression n:=n-1, starting with the value 5 for n.

Table indexation

Every variable bound to a table may be indexed:

name[expression]

The name identifies the variable while the expression should have a number value that is compatible with the size of the table that the variable is bound to. The result is the value contained within the slot indexed by the value of the expression (it should range from one to the size of the table).

The following transcript illustrates the use of indexation:

```
n: 20
:20
k: 1
:1
p[n]: k:=2*k
:[2, 4, 8, 16, 32, 64, 128, 256, 512,
1024, 2048, 4096, 8192, 16384, 32768,
65536, 131072, 262144, 524288, 1048576]
log2(x,m): if(x<p[m],m,log2(x,m+1))
:<function log2>
log2(1000,1)
:10
log2(5000,1)
:13
```

transcript

First of all a table `p` is created containing the first 20 positive integer powers of 2. Note that the value of `p` is printed out in the prescribed format. Next a function `log2` is defined taking two parameters `x` and `m`. This function traverses `p` in order to find the first slot `p[m]` that contains a value smaller than `x`. Note that `log2` keeps applying itself to the same value of `x` and the next integer value of `m` in order to perform this traversal. The function `if` is one of the predefined functions that will be explained in the coming paragraphs; intuitively it uses the validity of `x<p[m]` to select either the value of `m` or the value of `log2(x,m+1)`.

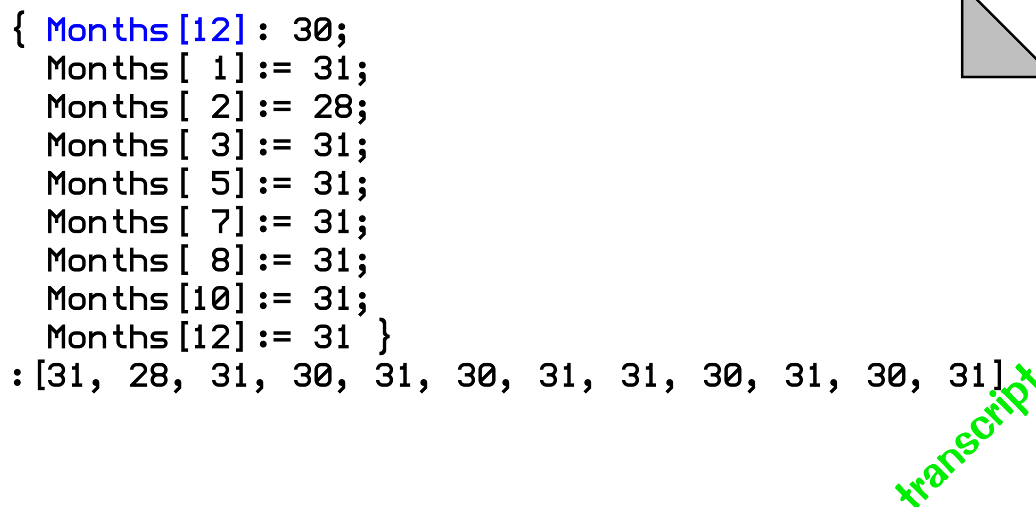
The transcript demonstrates that `log2` correctly computes the (rounded) logarithm of base 2 of the numbers `1000` and `5000`.

Table assignment

Tables bound to variables can have a slot value changed by a table assignment:

name[expression]:= expression

The name should refer to a variable that has a table bound to it in the current dictionary. The value of the expression between square brackets must be a number compatible with the size of the table and will result in the index of a slot. The value of this slot will be replaced by the value of the expression on the right-hand side; the updated table will be returned as the value of the table assignment.



In the above transcript table assignment is used to initialise a table `Months` holding the number of days in each of the twelve months. Note that the table index —ranging from 1 through 12— identifies the month.

Operators

Functions with names consisting of the following characters:

! # $ % ^ & * + - * / \ | < > = ~

are called *operators*. They can be used to define, apply or assign functions.

For instance:

```
$(p,q): if(p>q,p,q)

!(z): if(z<0,-z,z)
```

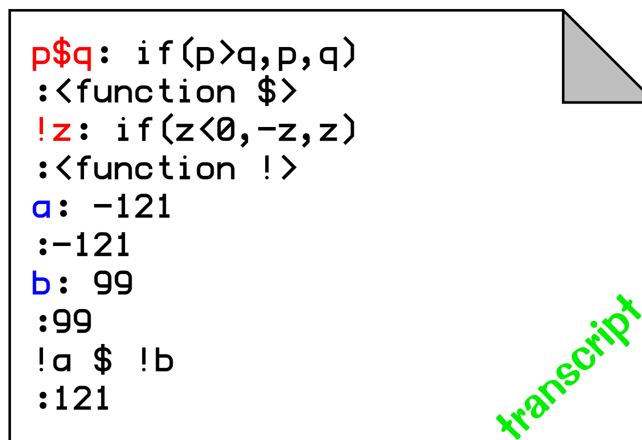
define perfectly valid functions that can be used in the regular way:

```
$(!(a),!(b))
```

resulting in maximum value of the absolute values of `a` and `b`. It would however be much more convenient to use standard operator notation:

```
!a $ !b
```

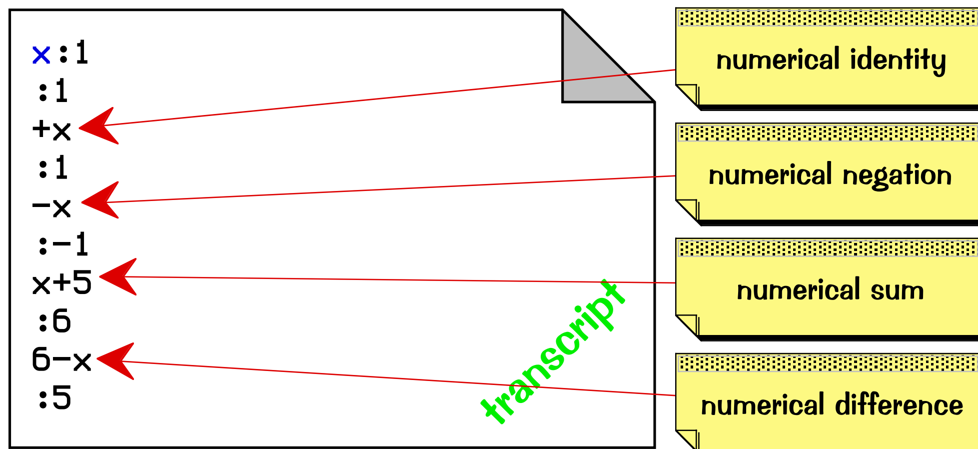
Pico supports operator notation as a (non essential) extension of its syntax:



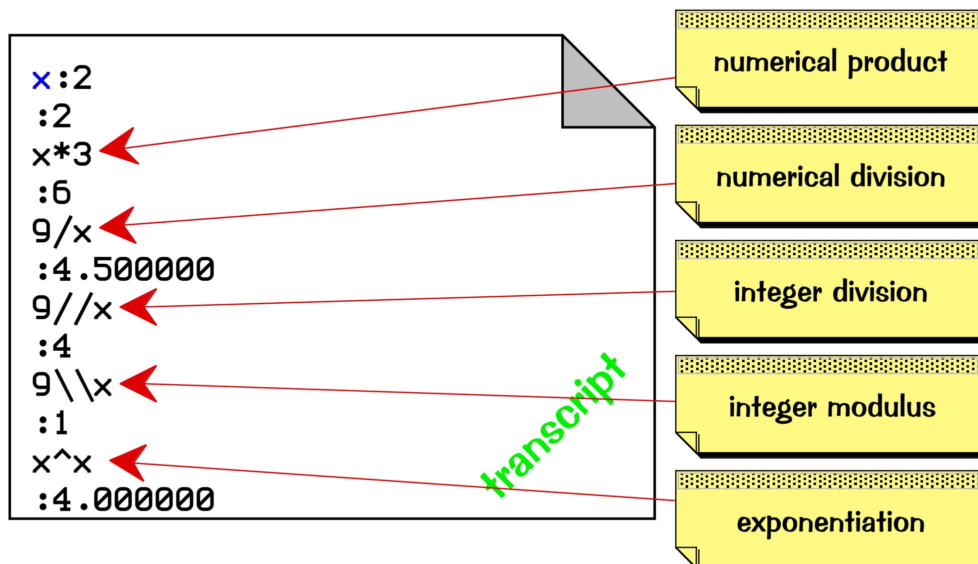
Calculus teaches us that infix and prefix notation involving operators requires precedence rules and a means to change these rules by using parentheses. We refer to the next chapter for a strict definition of these rules. For the moment, it is sufficient to use an intuitive approach to operators and their relative precedence.

Arithmetic functions

Pico comes with the standard complement of *native* arithmetic functions and operators for addition, subtraction, multiplication, division and exponentiation. The following transcript illustrates the use of the two native additive operators:



Note that both the + and the - operator can be used as unary and binary operator. The multiplicative operators *, /, //, \\ and ^ are always binary:



A number of rules apply to the types of arguments of these operators:

- the arguments of *, /, and ^ are restricted to numbers and fractions
- the arguments of // and \\ are restricted to numbers

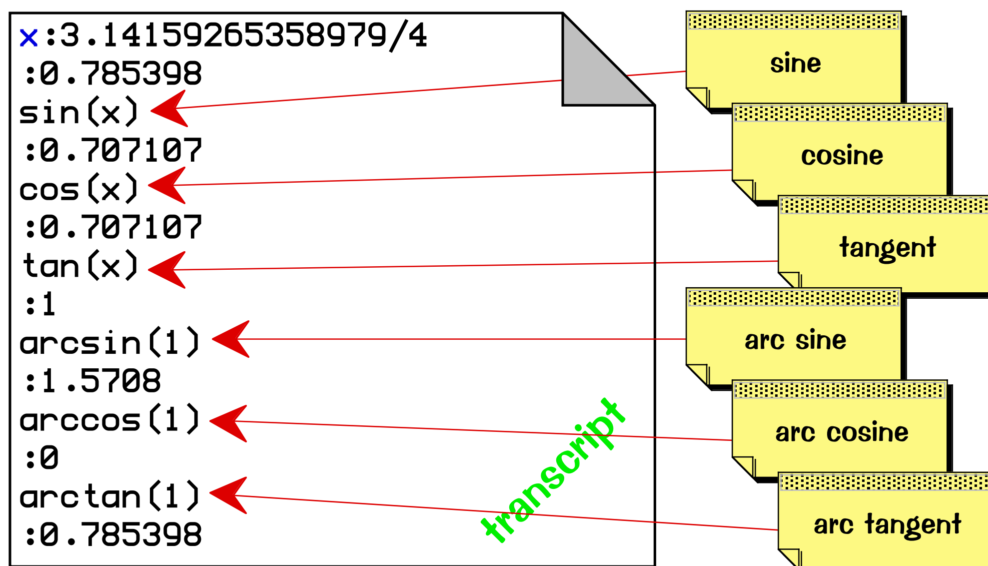
- the second argument of /, // and \\ cannot be zero
- the first argument of ^ cannot be negative

These rules are illustrated by the following transcript:

```
1/0
zero division: /
2.5//1.5
argument type conflict: //
1\\0
zero division: \\
-2^0.5
negative argument: ^
```

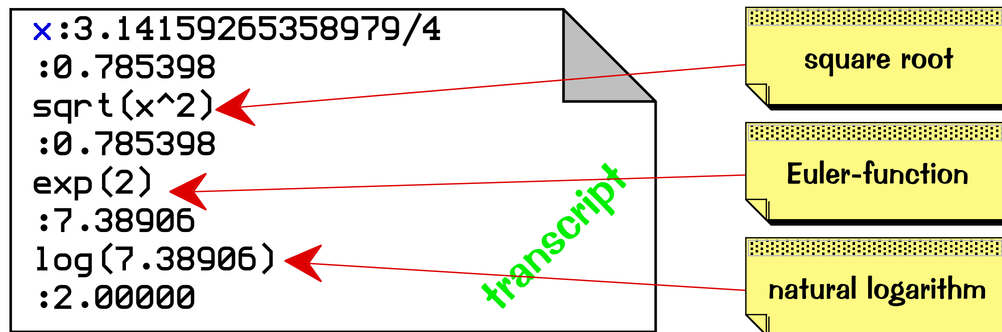
Transcendental functions

Pico is equipped with a standard set of native transcendental functions, that is the functions readily available on most pocket calculators. A first sequence consists of the three basic trigonometric functions and their inverse:



Note that arguments of `sin`, `cos` and `tan` as well as values returned by `arcsin`, `arccos` and `arctan` are expressed in π-radials. Moreover, the arguments of `arcsin` and `arccos` are restricted to the interval [-1, 1].

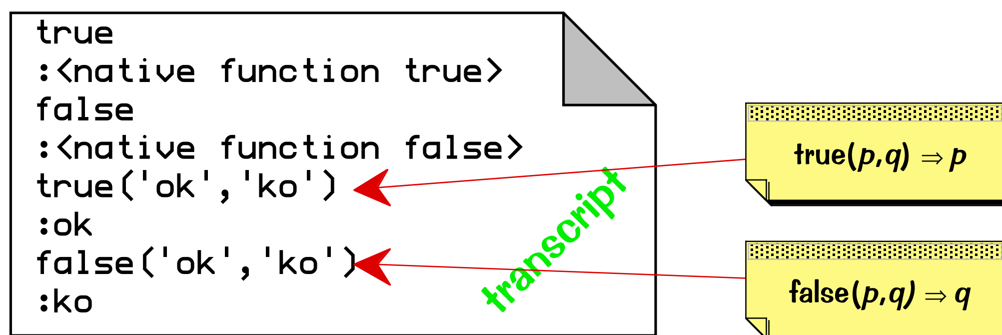
The next sequence consists of the `sqrt`, `exp` and `log` functions:



The argument of the `sqrt` function cannot be negative; the argument of the `log` function is required to be strictly positive.

Logical functions

The next family of native functions introduce *logic* into Pico. First of all, two values `true` and `false` are introduced to represent the two logical values:



Note that `true` and `false` are functions, or rather native functions as indicated by the behaviour of the **print** component. Both `true` and `false` take two arbitrary arguments; `true` however ignores the second argument and returns the value of the first argument while `false` ignores the first argument and returns the value of the second argument. At this point we do not know how to make a function ignore an argument; this will be explained in the next chapter.

Armed with `true` and `false` we can build the three basic logical predicates `and`, `or` and `not`. These predicates both take and return logical values and obey the following set of rules:

- and(p, q) is only true if both p and q are true
- or(p, q) is only false if both p and q are false
- not(p) is only true if p is false

The following three transcripts enumerate the possible combinations of applying `and`, `or` and `not` to `true` and `false`:

```
and(true,true)
:<native function true>
and(true,false)
:<native function false>
and(false,true)
:<native function false>
and(false,false)
:<native function false>
```

and(*p*,*q*) ⇒ *p*(*q*,false)

```
or(true,true)
:<native function true>
or(true,false)
:<native function true>
or(false,true)
:<native function true>
or(false,false)
:<native function false>
```

or(*p*,*q*) ⇒ *p*(true,*q*)

```
not(true)
:<native function
false>
not(false)
:<native function true>
```

not(*p*) ⇒ *p*(false,true)

These constitute so-called *truth tables* and completely define the behaviour of the three logical predicates. Are also included: their definition in terms of

the functional behaviour of the two logical values[5]. Again, the reader is invited to explore these definitions or else wait until the next chapter for a much more rigorous treatment of this matter. We will use this logic system to explore the concept of *control* in Pico programs.

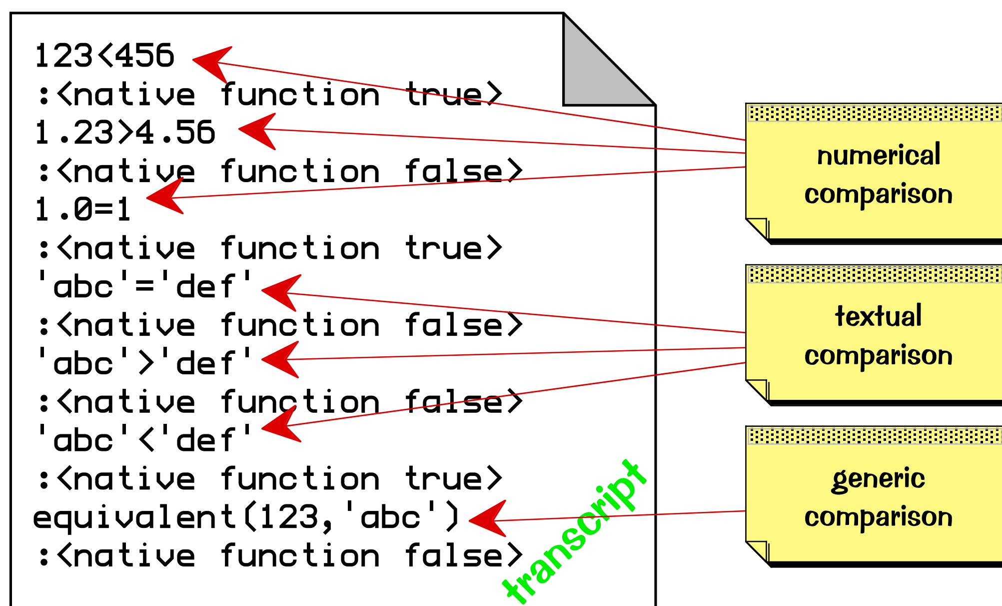
Relational functions

Three native binary operators <, > and = are available in Pico to compare numerical or textual values. The rules are the following:

- the two arguments are both either numbers, fractions or a mixture; in this case both numerical values are compared according to the standard sequencing of numbers
- the two arguments are both text; in this case both textual values will be compared according to the lexicographic ordering of the alphabet of characters

In order to compare arbitrary values, —that is numbers, fractions, text, functions and tables— a generic operator `equivalent` is provided. This operator checks for exact identity of the two arguments; for a detailed specification we again refer to the next chapter.

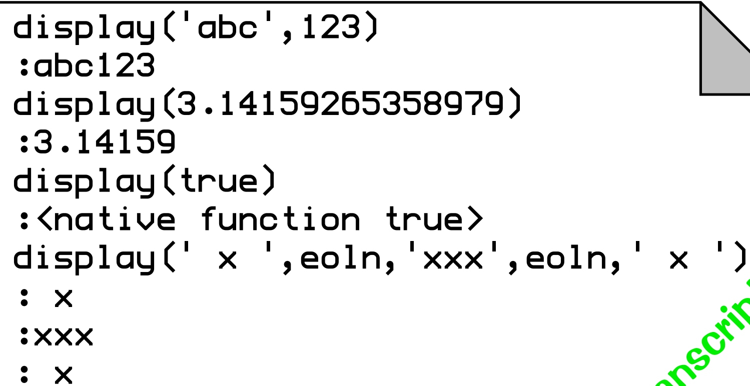
In all cases the resulting value will be either `true` or `false`, as illustrated by the following transcript:



[5] Actually, this definition of a logic system is due to Alonzo Church, an American mathematician who established a formal system supporting functions, called the λ-calculus, in the early thirties —long before computers came into being.

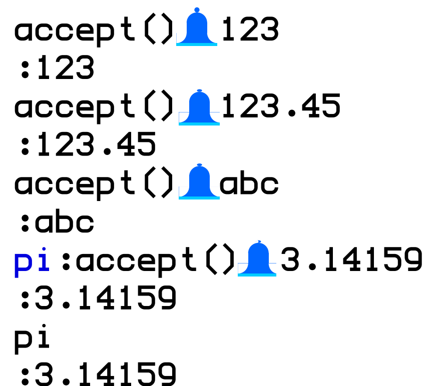
Communication functions

Pico also provides means for communication between a running program and the outside world. Typically the function `display` allows any value to be sent to the transcript —typically a computer screen— while the function `accept` retrieves a fragment of text —typically from a keyboard. The following transcript illustrates the use of the `display` function:



Note that `display` can take any number of arguments. We do not know yet how to define a function that behaves in this way; again we refer to the next chapter for more details. Also note that a native variable `eoln` is available: whenever displayed, the value bound to `eoln` generates a line break in the text printed to the transcript.

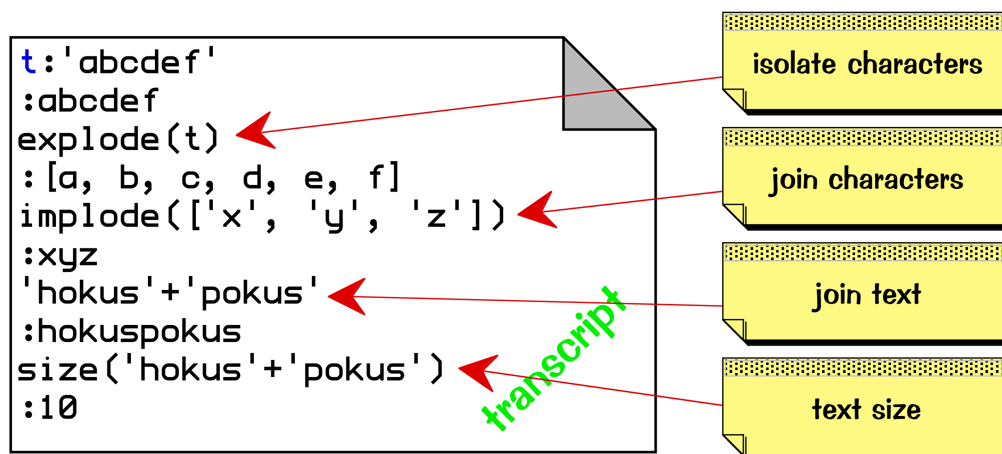
The `accept` function is illustrated in the following transcript:



The function `accept` takes no arguments; whenever applied, it produces some audio prompt —represented by the bell symbol— to warn that input is required. Evaluation is then resumed after the input is terminated —indicated by for instance an enter-key on the computer keyboard. The value of the `accept` function is the text representing the entered string of characters. Note that the result is text, even if only digits were entered. Converting text consisting of digits into a number will be discussed in one of the following paragraphs.

Text functions

Pico proposes a set of native functions to manipulate text:



The functions `explode` and `implode` convert text to and from a table with slots containing its individual characters —as text values of length one. The operator + is extended to accept two text values as arguments and serves to join these into one text value. Finally, the `size` function takes a textual argument and returns the number of characters that it consists of.

Composition function

It is often necessary to assemble multiple expressions into one composite expression, for instance when the prescription for a function definition cannot be stated in one simple expression. Pico provides the function `begin` as an answer to this need.

As shown by the following transcript, the `begin` function behaves as follows:

- at least one argument must be provided
- the arguments are evaluated from left to right
- the value of the last argument serves as final value

```
begin(display('name?'),
      name: accept(),
      eoln+'You are '+name)
:name?🔔 Napoleon
:You are Napoleon
{ display('name?');
  name: accept();
  eoln+'You are '+name }
:name?🔔 Bonaparte
:You are Bonaparte
```

transcript

The transcript also illustrates a non-essential syntax for an application of `begin`:

```
begin(a,b,…,z) ≡ {a;b;…;z}
```

The `begin` arguments are simply delimited by curly braces and separated by semicolons. Practice has shown that large Pico programs become much more readable and easier to debug whenever this alternate representation is used.

The `begin` function is extremely useful for introducing *nested* definitions. For instance:

```
f(x):{a:x;a}
```

features a variable definition of `a` that is valid during an application of `f` and has no effect otherwise. This is an effective means for hiding specifics about the elaboration of a function. In complex problems this enhances the legibility of programs.

Selection function

One of the most important native functions provided by Pico is the `if` selection function. The `if` function[6] takes three arguments, the first of which must have a `true` or `false` value[7], depending on which either the second

---

[6] The `if` is the Pico counterpart of the curly braces used in mathematics to list alternatives.

[7] Actually, in the next chapter we will see that any function with a behaviour similar to `true` and `false` will do

or the third argument will be evaluated. Pasted to the transcript is a first hint of a way to express a definition of the `if` function: it suffices to apply the value of the first argument to the two other arguments. For a formal elaboration we refer to the next chapter.

```
x:123
:123
y:456
:456
if(x>y,x,y)
:456
if(x>y,x-y,y-x)
:333
if(x\\2=0,display('even'),display('odd'))
:odd
display(if(x\\2=0,'even','odd'))
:odd
```

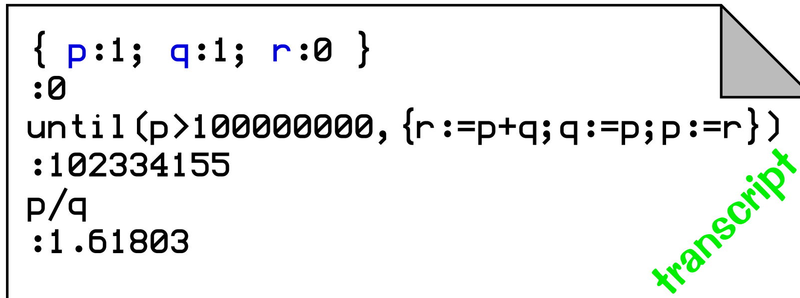

Loop functions

Pico provides three native loop functions: `while`, `until` and `for`. These are used to evaluate some expression iteratively; the only difference between these three loop functions lies with start and stop criteria for the iterative process. `while`, `until` and `for` closely reflect how loops are performed in the majority of programming languages; we will make relatively little use of them at this stage, which is why their behaviour is only superficially sketched here. In the next chapter all three will be formally defined.

```
k:1000
:1000
log2:0
:0
while((k:=k//2)>0,log2:=log2+1)
:9
```


The preceding transcript illustrates how the native `while` function is used to compute the truncated logarithm with base 2 of the number 1000. The function `while` takes two arguments; the first one should have a logical value. For as long as this value remains `true`, the second argument keeps on

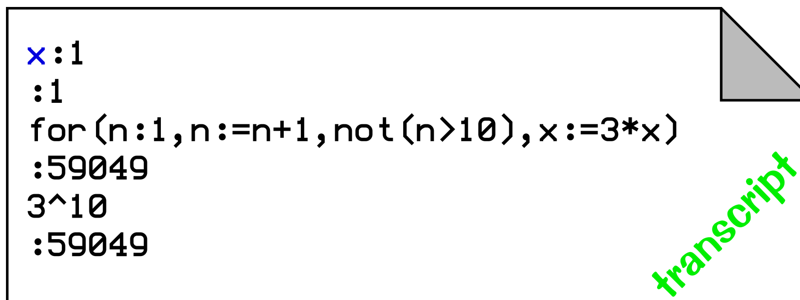
being evaluated. In this example, the expression `log2:=log2+1` is evaluated 9 times, because that is the number of steps it takes for `k:=k//2` to become zero.



The preceding transcript illustrates how the native function `until` is used to compute the first Fibonacci[8] number exceeding 100000000.

The function `until` takes two arguments; the first one should have a logical value. The second argument keeps on being evaluated until this value becomes `false`. In this example, the expression `{r:=p+q;q:=p;p:=r}` is actually evaluated 38 times for `p` to reach the value `102334155`. This transcript concludes by computing `p/q` which is an approximation for the golden ratio[9].

The following transcript illustrates how the native function `for` is used to compute the tenth power of the number 3:



The function `for` takes four arguments; the third one should have a logical value. For as long as this value remains `true`, the fourth argument keeps on being evaluated. This is similar to the `while` function; however, the other two arguments provide additional features. The first argument is evaluated once before the loop is started, while the second argument is evaluated at the

[8] In this text, Fibonacci numbers $(F_n)_n$ are defined as $F_n = F_{n-1} + F_{n-2}$ with $n>1$ and $F_1 = F_0 = 1$

[9] The golden ratio is defined as $(1+\sqrt{5})/2 = \lim(F_n / F_{n-1})$ for $n\to\infty$

conclusion of each iteration step. In this example, this mechanism is used to define a *counter*, that is a variable n counting the number of iterations and used to decide that the loop should stop after exactly 10 iterations. Note that the transcript checks the correctness of this loop against the standard computation 3^10.

Table functions

Pico provides a number of shortcuts to work with tables. First of all, there exists a native function tab that can be used to create and initialise table values with. It will take any number of arguments, evaluate these and insert the resulting values in the successive slots of a newly created table. In the transcript below, the tab function is used to set up a table holding the text values of the five vowels a, e, i, o and u. The variable t bound to this table is then used in a for loop to decide whether a character o accepted from the keyboard is indeed a vowel. Note that the native function size —introduced in the paragraph on text functions— is extended to return the number of slots in a table.

```
t:tab('a','e','i','o','u')
:[a, e, i, o, u]
z:accept() o
:o
for(i:1,i:=i+1,not(i>size(t)),
    if(t[i]=z,
       display(z,' is a vowel'),
       eoln))
:o is a vowel
:
t:=['a','e','i','o','u']
:[a, e, i, o, u]
```

transcript

In analogy to the begin function, a non-essential syntax for an application of tab has been defined:

$$\text{tab(a,b,…,z)} \equiv \text{[a,b,…,z]}$$

The tab arguments are simply delimited by square brackets and separated by comma's, again for reasons of convenience in handling larger programs.

Type functions

By now it is fairly clear that Pico values are partitioned into a number of distinct groups, called *types*. A value belongs to exactly one type and can never switch from one type to another. On the other hand, simply reading the text of a program is generally not sufficient to determine the type of the values that will be bound to the various variables and parameters used in the program. For instance:

```
if(a=0, x:=1, x:='one')
```

will assign either a number or a text value to the variable `x`, depending on the value for `a` retrieved from the current dictionary. As it is, the value of `a` could have been set using a totally unpredictable function, such as `accept`. The only way therefore to determine the types of values manipulated by a program consists of evaluating the program.

Pico provides a number of native functions to identify the types of values as they are computed in a program:

```
is_number(1)
:<native function true>
is_fraction(2.4)
:<native function true>
is_text('abc')
:<native function true>
is_table([1,2,3])
:<native function true>
is_function(+)
:<native function true>
is_void(void)
:<native function true>
```

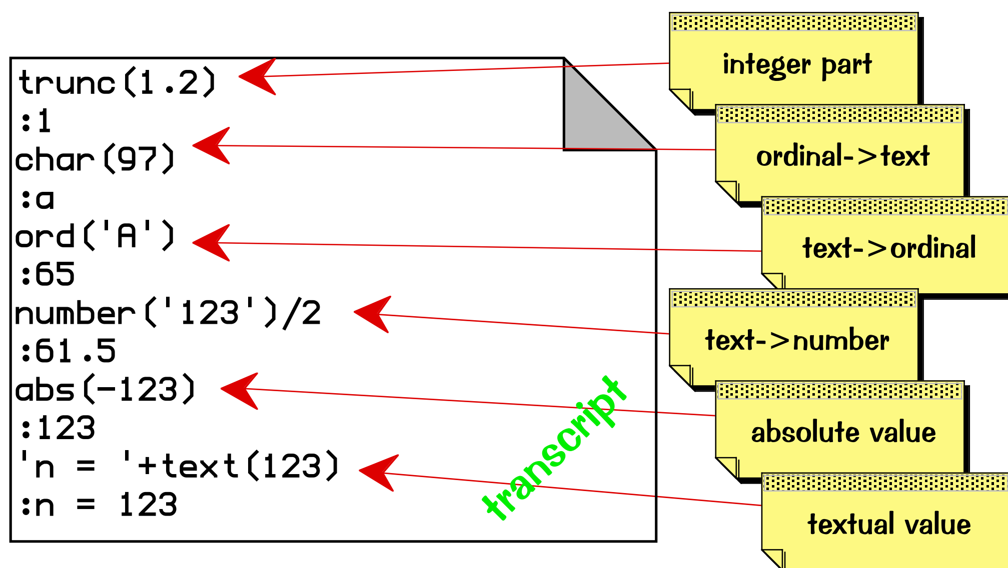
transcript

The functions `is_number`, `is_fraction`, `is_text`, `is_table` and `is_function` generate either `true` or `false` depending on whether the value of their single argument is a number, a fraction, a text, a table or a function. The function `is_void` is used to determine whether the value of the argument is void; this value, accessible through the native variable `void`, is the only value of a type distinct from the five others, and is generally used to indicate the absence of value. Examples will be given in the next chapter.

Conversion functions

The last group of native functions is used in Pico programs to perform *conversions*, that is, starting from a given value of a certain type generating an associated value of another type. These are the rules:

- the function `trunc` generates a number value corresponding with the truncation of the number or fraction value of the argument
- the function `char` generates a text value containing the character representation corresponding with the number value of the argument[10]
- the function `ord` generates a number value corresponding with the ordinal of the character held in the text value of the argument
- the function `number` will —if possible— generate a number or fraction value from the text value of the argument; if not, the value of `void` is returned
- the function `abs` generates a numerical value corresponding with the absolute value of the number or fraction value of the argument
- the function `text` generates a text value corresponding with the number, fraction or text value of the argument



Note that typically a combination of `accept` and `number` will be used to retrieve numerical values from a keyboard.

[10] In the next chapter we will cover the Pico alphabet and the ordinal value of each character

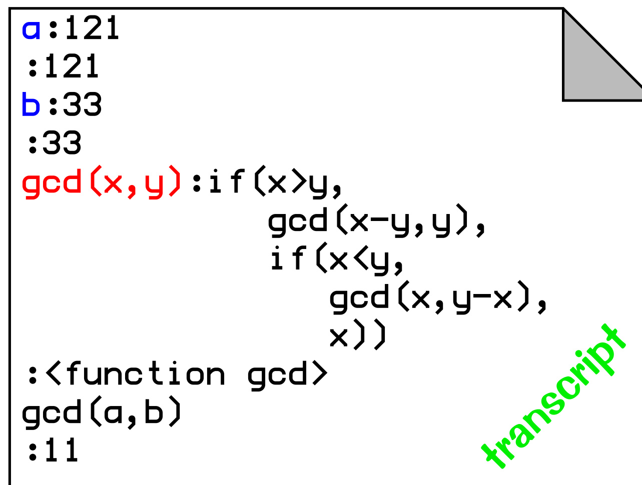
The greatest common divisor

Let us consider a task to be performed using Pico: we want to establish a general function that is capable of computing the greatest common divisor (gcd for short) of two positive integers.

In order to solve this problem, we draw upon the following recurrence relationship from discrete mathematics:

$$\gcd(p,q) = \begin{cases} \gcd(p-q,q) & \text{if } p>q>0 \\ \gcd(p,q-p) & \text{if } q>p>0 \\ p & \text{if } p=q>0 \end{cases}$$

which states that the gcd of two distinct positive integers equal the gcd of the smallest and the difference between the greatest and the smallest[11]. We can immediately transform this property into a computational process, using the combination of recursive function application and a selection:



```
a:121
:121
b:33
:33
gcd(x,y):if(x>y,
            gcd(x-y,y),
            if(x<y,
               gcd(x,y-x),
               x))
:<function gcd>
gcd(a,b)
:11
```


The prescription of the function `gcd(x,y)` specifies that either `gcd(x-y,y)`, `gcd(x,y-x)` or `x` is evaluated, depending on the relative values of `x` and `y`. This is obtained through a careful application of two nested references to the native `if` function. Since at each recursive step, the values of the arguments to `gcd` become smaller and smaller, we will eventually reach the gcd —which might be 1.

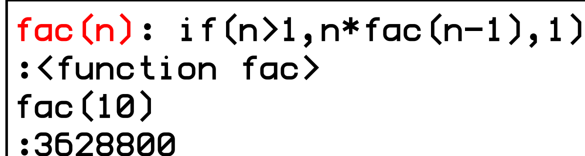
[11] A much more efficient recurrence relationship involving remainders is not considered here

Factorials

Let us try and repeat this approach in order to compute factorials:

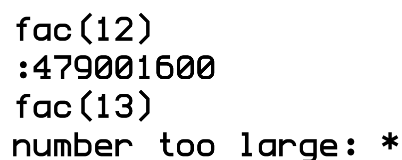
$$fac(n) = \begin{cases} n*fac(n-1) & \text{if } n>1 \\ 1 & \text{if } n\leq 1 \end{cases}$$

This recurrence relationship can again readily be transformed into a Pico function:



In the above transcript, the function application `fac(10)` will result in 9 recursive applications, ending when n reaches 1. Backtracking from this level, the necessary arithmetic is performed to compute 10! = 3628800.

Trying this out for other values of the argument, we eventually run into trouble:



We must never forget that in the physical world, somewhere in our tower of **REP**-machines there is a level at which we will encounter constraints on the values that we use in our programs. In this case we violated a rule stating that numbers cannot exceed the range [-1073741824, 1073741823][12].

---

[12] The Pico machine used to generate the transcript uses 31 binary digits to represent a number; hence $-2^{30} \leq \text{number} < 2^{30}$

Fibonacci numbers

The existence of physical limits to the magnitude of numbers should not detract us from trying to solve yet another problem: computing Fibonacci numbers (see the paragraph on loops). We again consult mathematics on the lookout for a suitable recurrence relationship:

$$fib(n) = \begin{cases} fib(n-1)+fib(n-2) & \text{if } n>1 \\ 1 & \text{if } n\leq 1 \end{cases}$$

which we can readily transform into a Pico function:

```
fib(n):if(n>1,fib(n-1)+fib(n-2),1)
:<function fib>
fib(5)
:8
```

Inspecting the function `fib` we might conclude that a lot of superfluous work is going on. For instance: `fib(5)` calls upon the computation of `fib(4)` and `fib(3)` and `fib(4)` calls upon the computation of `fib(3)` and `fib(2)` therefore `fib(3)` is computed twice! Actually, every function application of `fib` to a number greater than 1 generates two new applications. This indicates a kind of chain reaction where every recursive step doubles the number of computations.

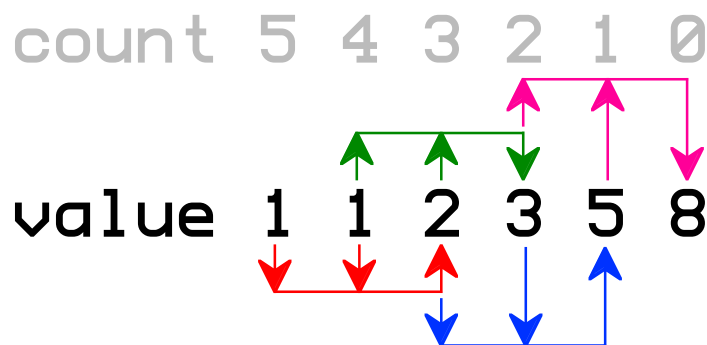
This behaviour is confirmed by an experiment:

```
fib(10)
:89
fib(20)
:10946
fib(30)
```

We actually have to pull the plug[13] on the evaluator after 10 minutes have passed by without any result.

This illustrates the difference between the approach involving a recurrence relationship and our approach involving a Pico program. While mathematics is concerned with definitions and properties, we also have to take the explicit behaviour of our computational process into account[14].

In order to construct a more appropriate program to compute Fibonacci numbers, we start by examining the computational process that goes on in our own head:



We start with the first two Fibonacci numbers and count steps while adding together two successive numbers to generate the following one. In contrast, our program performs the following action:

$$
\begin{aligned}
8 &= 5 + 3 \\
&= (3 + 2) + (2 + 1) \\
&= [(2 + 1) + (1 + 1)] + [(1 + 1) + 1] \\
&= \{[(1 + 1) + 1] + (1 + 1)\} + [(1 + 1) + 1]
\end{aligned}
$$

which is of course excessively cumbersome. Let us therefore try to adapt our program to our own way of dealing with this computation:

---

[13] Actually, we use an escape key on the keyboard of our personal computer

[14] This illustrates the difference between the concerns of mathematics and computer science or the difference of the *what* vs. the *how* of computational problems.

```
fib(p,q,r):if(r>1,
              fib(q,p+q,r-1),
              q)
:<function fib>
fib(1,1,5)
:8
fib(1,1,10)
:89
fib(1,1,20)
:10946
fib(1,1,30)
:1346269
```

transcript

The new definition of the function `fib` is slightly more complicated than before: we have to provide parameters `p` and `q` to hold two successive Fibonacci numbers and a parameter `r` to hold a counter. We can easily identify the following progression of applications of `fib`:

`fib(1,1,5)` ∅ `fib(1,2,4)` ∅ `fib(2,3,3)` ∅ `fib(3,5,2)` ∅ `fib(5,8,1)` ∅ `8`

indicating that the time consumed by our computational process is proportional to the index n of the required Fibonacci number. In our original program the time was proportional to 2^n.

Quadratic equations

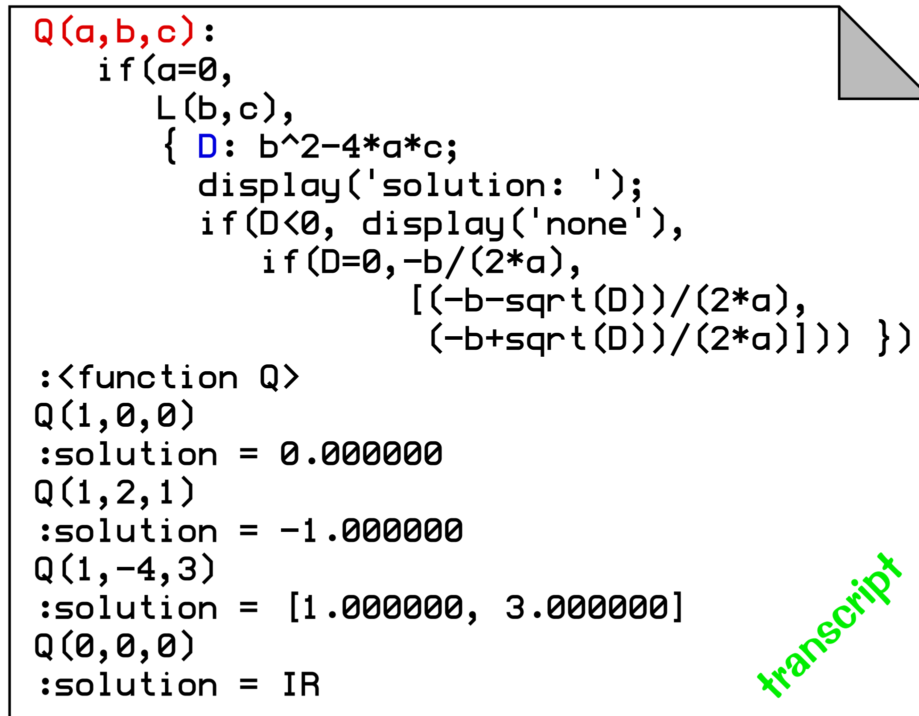
We will finish this chapter with a last example: finding the real roots of a quadratic equation $ax^2+bx+c = 0$. We will start by considering the case where a equals 0:

```
L(b,c):
  { display('solution: ');
    if(b=0,
       if(c=0,'IR','none'),
       -c/b) }
<function L>
L(0,0)
solution: IR
L(1,2)
solution: -2
L(0,1)
solution: none
```

transcript

The preceding transcript contains the definition of a function `L(b,c)` that solves this particular case.

The general case can then be treated as follows:



```
Q(a,b,c):
   if(a=0,
      L(b,c),
      { D: b^2-4*a*c;
        display('solution: ');
        if(D<0, display('none'),
           if(D=0,-b/(2*a),
                  [(-b-sqrt(D))/(2*a),
                   (-b+sqrt(D))/(2*a)])) })
:<function Q>
Q(1,0,0)
:solution = 0.000000
Q(1,2,1)
:solution = -1.000000
Q(1,-4,3)
:solution = [1.000000, 3.000000]
Q(0,0,0)
:solution = IR
```


The function `Q(a,b,c)` defers to `L` whenever `a` is zero; in the other case a discriminant `D` is computed and depending on its value the existence of a double root -b/2a or two distinct roots $(-b \pm \sqrt{D})/2a$ is reported.

Conclusion

We have made an informal introduction of the language Pico. We have covered the various values —number, fraction, text, function and table— and the various grammatical constructs —references, applications, indexations— and their definition or assignment.

This gives us a sufficient picture of Pico so as to use it as meta-level language for the specification of a machine to interpret base-level programs also written in Pico. This will be the subject of the next chapter.