Anubis Language Manual

2006/06/01

www.anubis-language.com

Contents

1 Conditions of use and Copyrights **3**

2 User manual **4**
2.1 Using the software . . . 4
2.1.1 Using an editor . . . 4
2.1.2 Invocation of the compiler . . . 4
2.1.3 Files . . . 4
2.1.4 Lauching an Anubis module . . . 5
2.1.5 Compiler messages . . . 5
2.2 Source texts . . . 6
2.2.1 Comments . . . 6
2.2.2 Paragraphs . . . 6
2.2.3 Some elements of syntax . . . 7
2.2.4 The keywords 'read' and 'public' . . . 9
2.2.5 The keyword 'global' . . . 10
2.3 Data types . . . 10
2.3.1 Introduction . . . 11
2.3.2 Enumerations . . . 11
2.3.3 Explicit typing . . . 12
2.3.4 Agglomerations . . . 13
2.3.5 Alternatives . . . 14
2.3.6 Inductive types . . . 15
2.3.7 Schemas . . . 16
2.3.8 Functional types . . . 17
2.3.9 The keyword 'type' . . . 18
2.4 Terms . . . 19
2.4.1 Applicative terms . . . 19
2.4.2 Conditionals . . . 20
2.4.3 Abbreviations and selective conditionals . . . 22
2.4.4 Local definitions . . . 23
2.4.5 Functions . . . 24
2.4.6 The keyword 'define' . . . 25
2.5 Concrete data . . . 26
2.5.1 Dynamic variables . . . 26
2.5.2 Multiple dynamic variables . . . 28
2.5.3 Monitoring variables . . . 28
2.5.4 Connections . . . 29
2.6 Programming techniques . . . 31
2.6.1 Determinism . . . 31
2.6.2 Modularity . . . 31
2.6.3 Functional programming . . . 33
2.6.4 Recursion . . . 35
2.6.5 Loops . . . 37
2.6.6 How to make an automaton . . . 38
2.6.7 Object oriented programming . . . 41
2.7 Tricks . . . 43
2.7.1 How to use 'alert'. . . . 43
2.7.2 What are the cases in my conditional ? . . . 44
2.7.3 What is the type of this term ? . . . 45

Copyright © Alain Prouté 2005–2006

1 Conditions of use and Copyrights

Conditions of use of the Anubis software

- The user receives the right to use the software, not a property right of any kind.

- The right to use Anubis Personal Edition is granted for free for non-profit and personal/family use only. **Any profit or non personal/family use of the Anubis software** is subject to a licencing fee. If this eventuality, you must contact the authors at: licencing@anubis-language.com

- The authors do not provide any warranty of any kind. It is the responsability of the user to test if the software is useful for any purpose.

Copyright

The software Anubis Personal Edition (the compiler, the virtual machine, the library, the examples, the documentation) remain indefinitely the property of their authors:

- **Alain Prouté** : Main author (compiler, virtual machine) and inventor of the Anubis language. Author of parts of the library and documentation.

- **David René** : Author of the Windows(R) port of the compiler and virtual machine. Authors of parts of the library and documentation.

- **Olivier Duvernois** : Author of parts of the library.

Nevertheless, the authors grant the right of using the software and the library to the final user according to the above conditions of use of the Anubis software.

Third party copyrights

Anubis uses for its internal operation several third party libraries. Below are the legal references of these libraries:

- "This software is based in part on the work of the Independent JPEG Group."

- "This product includes software developed by the OpenSSL Project (http://www.openssl.org/)"

- "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

- "This product includes software written by Eric Young (eay@cryptsoft.com)"

Windows(R) is a trademark of Microsoft(R) Corp.

All names of products or brands mentioned in this documentation are trademarks of their respective owners.

Copyright © Alain Prouté 2005–2006

2 User manual

In this manual, you will learn the essentials of the Anubis language and you will learn how to use the Anubis compiler. Since Anubis is a language which does not look like the most commonly used languages, you will have to adapt your way of thinking programming, especially if you are trained to the C language or similar languages. However, if you are trained to strongly descriptive languages like Lisp for example, it will be easier for you. Finally, if you did never write a computer program, you will not have much problems. The Anubis language is easy to learn and to use.

2.1 Using the software

The Anubis compiler has been created under a Unix system and is a console style program. Under a strictly graphic system like Windows(R), console programs are less easy to use. However, text editors for programmers may help you to use the compiler more comfortably. We recommend to users of Anubis under Windows(R) to use the Crimson Editor `http://www.crimsoneditor.com`.

So, the Anubis compiler is invoked (started) by a command line to be normally written into the console. This command line is described in the **Invocation of the compiler** section below. Under Windows(R), the command line should be known by the text editor and executed by a simple hit on a key (normally **F7**).

The Anubis compiler reads one or several source files, written by yourself or belonging to the library, and produces eventually some modules. These modules may be later executed.

2.1.1 Using an editor

The Anubis compiler and virtual machine are much easier to use with an editor for programmers than within a console.

The editor is used of course for the writing of your source files, but it may also be able to call the Anubis compiler and the Anubis virtual machine for you. In this case, you just have to strike a key (usually **F7** for compiling and **F5** for executing) instead of writing a command line into a console.

For Windows(R) users we recommend the Crimson Editor `http://www.crimsoneditor.com`, which is very simple to use. We provide in the distribution of Anubis for Windows(R), several files to be copied to the directories of the Crimson Editor. Precisely, the subdirectory **My Documents\my_anubis\Crimson Editor** should be joined to **Program Files\Crimson Editor**.

For Linux users, we don't recommend any editor. A number of them are using Emacs which is able of all the above (and much more).

2.1.2 Invocation of the compiler

The Anubis compiler is invoked by a command line like this one:

```
anubis file_name options
```

where `file_name` is the name of the source file to be compiled, and where `options` represents zero, one or several options. Options begin by an hyphen. They may be placed either before or after the name of the file to be compiled.

If you invoke the compiler without a file name and without options:

```
anubis
```

you get the version number of the compiler, a recall of the invocation syntax, and the list of all available options.

2.1.3 Files

The Anubis compiler reads the file whose name is given on the command line. Of course, this name may also be a path, and this path may be either absolute or relative. Here is how the Anubis compiler finds the file.

Copyright © Alain Prouté 2005–2006

- If the path is absolute, it indicates a file unambigously, and this file is read by the compiler.
- If the path is relative, the compiler looks for that file at three places successively, until it finds the file:
  - First relatively to the directory from which the compiler has been invoked,
  - if the file is not found, it looks relatively to the directory **my_anubis/library**,
  - if the file is still not found, it looks relatively to the directory **anubis/library**.

For example, to compile the file **hello.anubis** which is in the directory **anubis/library/examples**, it is enough to write the following from within any directory:

```
anubis examples/hello.anubis
```

Since the compiler will (probably) not find the file in the subdirectory **examples** of the current directory (which maybe does not exist), neither in the subdirectory **examples** of **my_anubis/library** (which maybe also does not exist), it will find the file in **anubis/library/examples**.

2.1.4 Lauching an Anubis module

Since the 1.6 package, the specialised shell, **anbshell.exe** is discontinued in the Windows(R) version of Anubis. Because of the recent improvements of **anbexec.exe**, **anbshell.exe** is no more useful. From now on, the behavior of **anbexec** is unified between Windows(R) and Linux.

To execute an **Anubis** module `.adm`, it is enough to enter the following in a console (shell):

```
anbexec <name of module> <argument 1> ...
```

or even more simply:

```
<name of module> <argument 1> ...
```

because the compiler constructs a file `<name of module>.bat` (under Windows(R)) or `<name of module>` (under Linux) which is placed into the directory **my_anubis/shells**. This directory is recorded in the value of the environment variable `PATH`.

It is not necessary to add the extension `.adm` after the name of the module.

Nevertheless, it is much more handy to configure an editor of programs in order to launch these commands by striking a single key.

2.1.5 Compiler messages

When an error is detected, the Anubis compiler sends a message to the console. That no message appears on the console means that your text is correct. In this case only modules may be created by the compiler.

The Anubis compiler performs a fine analysis of your texts, and has a set of about 100 error messages. Each message indicates first the name of the source file within which the error has been found, and the line and column numbers of the error. In the case of a syntax error, the error must be searched for before the indicated position.

The messages of the compiler are rather detailed, and some of them contain expressions which have been reformated by the compiler. The compiler does not make big formating efforts, and these expressions are sometimes not so easy to read, especially if they are very long.

The Anubis language has a notion of type parameter for representing arbitrary types. During the analysis of the paragraphs, some types are unknown and are progressively determined by unification. When it is the case that, at the end of the checking of a paragraph, some types are still unknown, the compiler sends a message in order to show where these unknown types are involved. To that end, it prints the whole paragraph with all the typing details which have been determined. Within this paragraph, unknown types are represented by type

Copyright © Alain Prouté 2005–2006

unknowns, which are of the form $5453, for example, in other words, a character $ followed by an integer. You must find these unknowns and update your source text by typing explicitly appropriate terms.

Author's note: The Anubis 1 compiler is a complex program (about 550 pages in the C language) and is furthermore a first writting. It suffers a little from the fact that since its creation it has been enriched by concepts which where not included in the very first design, and it is true that such addons are sometimes not completely clean. For sure, it is not perfect, even if since the year 2000 it gives satisfactory results. Nevertheless, in order to ensure an easy maintainance of this program, it is full of tests. If a problem appears, a message will be sent to the console, indicating the precise position of the problem in the very compiler source. Of course, such a message is of no use except for the author. If you get one, it would be very kind of you that you communicate the text of this message to anubis.language@free.fr.

2.2 Source texts

The content of a file written in the Anubis language is called a source text. Such a file is written by yourself or is found in the Anubis library.

A source text is a (eventually empty) sequence of paragraphs. Outside the paragraphs one may freely write comments.

2.2.1 Comments

The Anubis compiler detects the beginning of a paragraph when it encounters a keyword in the leftmost column. On the other hand, it detects the end of a paragraph simply because it knows the syntax of paragraphs. What you write before the first paragraph, between the paragraphs, or after the end of the last paragraph is ignored by the Anubis compiler. This allows to write comments very freely, in other words without having to use recognition signs for these comments. We recommend that you put a margin of three blank characters everywhere, not putting anything in the leftmost column, except when there is a keyword beginning a paragraph.

It is also possible to put comments within the paragraphs, but of course, in this case, they must be indicated by a recognition sign allowing the compiler not to confuse them with the actual text of the paragraphs. These recognition signs are the same as in the C++ language, in other words, a comment may be written between the signs `/*` and `*/` or between the sign `//` and the end of line.

Below is an example of a source text within which off paragraph comments are printed in green, and in paragraph comments in red.

```
    'length' is a function for computing the length of a list:

 define Int32     // returns the length of the list 'l'
    length
      (
         List($T) l    /* the list whose length is to be computed */
      ) =
    if l is
      {
         [ ] then 0,     // empty lists are of length 0
         [h . t] then 1+length(t)   /* otherwise the length is 1 plus
                                       the length of the tail
                                       of the list*/
      } .

    Easy ! Is'nt it ?
```

Notice that `/*` and `*/` are convenient for multiline comments. They may also be nested.

2.2.2 Paragraphs

A source text in the Anubis language is a (possibly empty) sequence of paragraphs. A paragraph begins by one of the following keywords or key locutions:

Copyright © Alain Prouté 2005–2006

- **read**
- **type**
- **public type**
- **define**
- **public define**
- **global define**

provided that it is written in the leftmost column.

The keyword **read** instructs the compiler stop reading the current file for reading another one, and then return to the reading of the first file.

The keyword **type** introduces a type definition, and the keyword **define** introduces a definition (or declaration) of a datum (which may be a function).

public and **global** modify the scope of paragraphs.

How to use these keywords is explained in the subsequent sections.

Notice that the fact that the keyword beginning a paragraph must be placed in the leftmost column, is very convenient when you are developping your program. If you want to inhibit a paragraph temporarily, without erasing it, just add a white space in front of the keyword.

2.2.3 Some elements of syntax

We provide here some elements of the syntax of the Anubis language, the most important ones.

A symbol is a non-empty concatenation of characters among the following:

- lowercase letters from `a` to `z` (no accent),
- uppercase letters from `A` to `Z` (no accent),
- digits from `0` to `9`,
- the underscore character: `_`.

A symbol cannot begin by a digit (only numbers may begin by a digit). Symbols which are beginning by an uppercase letter are reserved as names for types. Other symbols (i.e. beginning by the underscore or by a lowercase letter) are used as names for data (including functions).

The Anubis compiler is case sensitive, which means that it considers uppercase and lowercase letters as distinct. For example, `a` and `A` are considered as distinct letters.

However, a small set of such concatenations are keyword. They cannot be used as symbols. They are the following:

- alert
- delegate
- else
- if
- is
- lock
- protect
- terminal
- then

Copyright © Alain Prouté 2005–2006

- with

The Anubis language also has initial keywords, which are used to begin paragraphs. They are considered as keywords only at the beginning of a paragraph. Hence, they may still be used as symbols within the paragraphs. They are:

- define
- global
- public
- type
- read

The expressions of the Anubis language represent either types or data (in this last case, these expressions are called terms). The distinction between those expressions which represent types and those which represent data is made on a syntactical basis (not on a semantical basis). For example, for symbols, this distinction is visible on the first character, which is uppercase for types.

The syntax of Anubis would be ambiguous without precedence and association rules. For example, if you write:

```
1 + 2 * 3
```

the compiler understands `1 + (2 * 3)` (so that the result is `7`). This is because the binary sign `*` has precedence over the binary sign `+`. If you want to multiply `1 + 2` by `3` (which will make `9`), you must write:

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

In Anubis, there are functions (which are just a particular sort of data), and these function may be applied to arguments (which are also data). For example, if the function is represented by the symbol `f`, and if it takes (say) two arguments, represented by the symbols `a` and `b`, the result of applying the function to these arguments is denoted:

```
f(a,b)
```

Anubis uses a particular syntax which may be used for example for representing lists. First of all notice that `[ ]` (i.e. an opening bracket followed by any number of blank characters, including none, and a closing bracket) is a regular symbol (this is called an exceptional syntax). Next, there is a particular binary sign which is made of three parts which are `[`, `.` and `]`. This sign is applied to its two operands (say `a` and `b`) as follows:

```
[a . b]
```

This may be used for representing the list whose head is `a` and whose tail is `b`. Using this syntax, the list of the four elements `a`, `b`, `c` and `d` should be represented as follows (where `[ ]` represents the empty list):

```
[a . [b . [c . [d . [ ]]]]]
```

which is obviously too complicated. For this reason, the Anubis compiler allows the following notation instead of the above one:

```
[a, b, c, d]
```

Of course, this principle applies to any number of elements. Notice that this is purely syntactical, and is valid independently of the many definitions you may give to the symbol `[ ]` and to the binary sign `[ .  ]`.

This will be enough for reading the subsequent sections. We will say more on syntax later.

Copyright © Alain Prouté 2005–2006

2.2.4 The keywords 'read' and 'public'

If you write down a definition of a type or of a datum, it is for sure that you have the intention of using it somewhere in your source texts. In order to use a definition at a certain place, it is necessary that this definition is visible from that place. The keyword `public`, added in front of a definition, allows this definition to be potentially visible from other files, and the keyword `read`, used in another file, allows to see this definition.

For example, your file **main.anubis** may contain the following line:

```
read tools.anubis
```

because it needs a certain type `T` defined in **tools.anubis**. In this last file, we will find the definition of type `T` which begins like this:

```
public type T:
```

The keyword `public` used as above makes the definition of type `T` potentially visible from within all other files. Furthermore, the line `read tools.anubis` written in **main.anubis** allows the use of all public definition of **tools.anubis** from within **main.anubis**.

The visibility provided by the keyword `read` is not transitive. This means that a public definition is not visible through two `read`. Consider for example three files **main.anubis**, **features.anubis** and **tools.anubis**, and assume the contain (among other things):

main.anubis:

```
read features.anubis
```

features.anubis:

```
read tools.anubis
```

tools.anubis:

```
public define T:
```

Then, the type `T` will be visible from within **features.anubis**, but will not be visible from within **main.anubis**. In other words, the visibility provided par `read` is not transmitted.

As a consequence, you must write at the beginning of your source files all the '`read`' which are required for accessing directly public definitions you need.

This method allows to avoid public name collisions when you join several projects int a single one, especially if these projects contains maybe modified copies of source files of the same origin. If `read` were transitive, the resulting collisions would oblige to rename many public types and data.

The Anubis compiler remembers the paths of all files it reads during a single compilation. It will never reread a file it has already read. It just gives access to its content. Hence, there is no risk to use the keyword **read** even if its use introduces circularities (which is however not recommended).

The file path you write after **read** may be either relative (it does not begin by /, or something like **C:** under Windows) or absolute (it does begin by /, or **C:**). When it is absolute, it represents a unique, well determined file, and the compiler just looks for this file. When it is relative, the compiler looks for the file successively in three different places:

- (1) relatively to the directory of the file currently being read,
- (2) relatively to the directory **my_anubis/library**,
- (3) relatively to the directory **anubis/library**.

Copyright © Alain Prouté 2005–2006

Of course, if at the end of this search the file is not found, the compiler stops with an error message.

Since version 1.6, and in order to avoid unplesant surprise for the user, a new initial key locution has been introduced. It allows to move a library file from one place to another one in a way which is quite transparent for the user. Thus, for example, the file **library/tools/sdbms.anubis** has been replaced by **library/data_base/sdbms.anubis**. At its original place, in **library/tools**, we find a new file **sdbms.anubis** containing the following line only:

```
replaced by data_base/sdbms.anubis     (since version 1.6)
```

Normally, you don't have to use the key location `replaced by` yourself. Its use is reserved to the team preparing the library packaging.

If the compiler is led to open such a file (called a relay file), it generates an information message saying where the file has been moved. It is recommended to make corrections of your `read`s as soon as possible so as to avoid this message. In any case, the compilation is performed normally even if you don't do it.

2.2.5 The keyword 'global'

The keyword **global** may be placed at the beginning of a paragraph in front of the keyword **define**. The effect of **global** is the creation of a module, i.e. a file which may be later executed.

The use of **global** puts some restrictions on the paragraph. It must begin as follows:

```
global define One
   my_module
     (
       List(String)  args
     ) =
   ...
```

where `my_module` is the name of the module you want to create. The module itself will be a file named **my_module.adm** that the compiler puts on the directory **my_anubis/modules**.

The above definition is actually the definition of a function taking only one argument of type `List(String)` and returning a value of type `One`. It must be like this in the presence of the keyword **global**. The only things you can change are the name of the module (`my_module` above) and the name of the arguments list (`args` above).

The argument of type `List(String)` (a list of character strings) represents the arguments which will be put on the command line at the time the module will be executed. We will see later how one may recover the elements of a list. The return type `One` contains only one datum whose name is `unique`. This datum must be returned by the function.

Of course, the execution of the module consists in applying the function defined that way to the arguments which will be put on the command line.

2.3 Data types

The data that you will manipulate are not undifferentiated. On the contrary they are classified according to their type. Each operation can be applied only to data of a certain type, and not to data of any type. This constraint provides the safety of programming. The types force some form of coherence to be maintained, avoiding the application of an operation to data for which the operation makes no sens.

It is easy to imagine that the finer the typing (in other words the more the data are differentiated by their types), the safer the system, and the lower the risk of error. From this point of view, Anubis is fairly more advanced than C, C++, Java, Basic, etc... and even than Lisp, Ada or CAML. If you know one of these languages, you will quickly have evidence for this fact.

Copyright © Alain Prouté 2005–2006

2.3.1 Introduction

Anubis allows you to create (define) an infinite variety of types. To that end Anubis provides several mecanisms enabling the construction of types:

- the agglomerations,
- the alternatives,
- the induction,
- the constructor of functional types.

Do not hesitate to define a new type each time you want to introduce a new sort of data in your program, even if one of the previously defined types may do the job. Defining a new type, even if its definition is identical to the definition of an already defined type (of course, it has a distinct name), allows the compiler to understand that you attach a different meaning to the new data. That way you reduce yet a little more the risk of errors.

2.3.2 Enumerations

The simplest way for defining a new type is just enumerating the data it contains. Of course, this allows only to define finite types and even types which are not very big. Nevertheless, as we shall see through some examples, they may be very important.

Let us begin by a classical example: the type of booleans. Notice that you don't have to define this type, because it is already defined in the Anubis system. It is well known that this type contains only two elements which are usually called 'true' and 'false'. We will call them `true` and `false`, and the type of booleans itself will be called `Bool`. Below is the paragraph which establishes this definition:

```
type Bool:
   false,
   true.
```

The meaning of this definition is that there is a type whose name is `Bool`, and that there are two and only two (distinct) data of this type, whose names are `true` and `false`.

Notice the following facts:

- the keyword **type** is written in the leftmost column,
- the name of the type begins by an uppercase letter,
- the name of the type is followed by the sign **:** (colon),
- the names of the elements do not begin by an uppercase letter,
- the names of the elements are separated by commas,
- the paragraph is terminated by a dot.

It could have been the case that we put `true` before `false`. This has no importance at all.

In general, an enumerated type is defined by a paragraph as this one:

type *Name*:
   $element_1$,
   $element_2$,
   ...
   $element_n$.

You cannot give the same name to two elements of an enumerated type. This would create an ambiguity impossible to solve. On the contrary, you may put only one element or no element at all. For example, the Anubis system uses frequently the type `One` which is defined as follows:

Copyright © Alain Prouté 2005–2006

```
type One:
   unique.
```

which contains only one datum named `unique`. An enumerated type with only one element is called a singleton. This is the case for the type `One` (hence its name). Of course, if you have a datum whose type is a singleton, this datum does not carry any information, since it is alone of its type. We will have several occasions to use the type `One`.

The Anubis system also contains a type `Empty`, which is defined as follows:

```
type Empty:.
```

in other words, with no element at all. Actually, this type is not very useful, but it is correct.

2.3.3 Explicit typing

We will make some experiments with enumerations. This will allow us to begin to examine the behavior of the Anubis compiler when it encounters errors.

Let's begin by defining an enumerated type. For example, one may define a variant of the type `Bool` for a ternary logic (with three truth values, instead of two). Such a type may be defined as follows:

```
type Ternary:
   false,
   maybe,
   true.
```

So, this type is called `Ternary` and has three elements whose names are `false`, `maybe` and `true`. Notice that the fact that the type `Bool` already contains elements named `false` and `true` does not create any problem. In Anubis, symbols may be overdefined (we may also say overloaded), which means that they may represent several different things. If thoses things are of different types, the compiler will be able to chose the right interpretation. If however an ambiguity persists, the compiler will complain about it (and ask for its resolution).

In order to test this, open a new file, copy the definition of the type `Ternary` above into it, and add the following paragraph:

```
define Ternary f = false.
```

In other words, give the name `f` to the element `false` of type `Ternary`. This definition is clearly non ambiguous, because of the presence of the indication `Ternary`. Anyway, the compiler will not complain. Make this test by compiling the file.

On the contrary, if you modify the last paragraph as follows:

```
define One f = forget(false).
```

the compiler will complain, because `forget` (which is defined in **predefined.anubis**), accepts an argument of any type (and returns the unique value of type `One`). Since there are at least two definitions of the symbol `false` (one of type `Bool` and one of type `Ternary`), it is not possible to know which one is to be chosen.

You can solve this ambiguity by explicitly typing the symbol `false`, in one of the following ways:

```
define One f = forget((Bool)false).
define One f = forget((Ternary)false).
```

One may also type explicitely the function instead of its argument. In this case, one must write this for example:

```
define One f = ((Bool -> One)(forget))(false).
```

Copyright © Alain Prouté 2005–2006

Indeed, the pair of parentheses around `(Bool ->One)(forget)` is mandatory because the precedence level of explicit typing is lower than the precedence level of applicative terms. Furthermore, the pair of parentheses around the function itself is mandatory for obscure syntactical reasons. If one writes:

```
((Bool -> One)forget)(false)
```

the compiler produces a syntax error.

Notice that explicit typing is in no way a transtyping (the fact of forcing a change of type). The transtyping does not exist in Anubis. It is worth noticing that this is made possible by the typing system, which is not the case of most other languages.

2.3.4 Agglomerations

One of the most usual principles (in programming languages) for creating new data is to put together several data (which may be of different types). For example, one may want to consider pairs made of a boolean and an integer. One may write down the following definition:

```
define (Bool,Int32) p = (true,4).
```

So, the symbol `p` gets the value `(true,4)`, which is of type `(Bool,Int32)`.

The type `(Bool,Int32)` is called (in Anubis) an agglomeration type (in mathematics, it would be called a cartesian product), and the pair `(true,4)` itself is called an agglomeration.

One may agglomerate any number of data, actually at least two, because if `T` is a type, `(T)` is the same type, and the syntax `( )` is not accepted by the compiler for representing a type. Mathematically, an agglomeration of zero types must be a singleton. The type `One` already plays this role, and its unique element is denoted `unique`, not `( )`.

The example above is an anonymous agglomeration type. Indeed, it received no name. Most often it is preferable to name agglomeration types. An example which is defined in **predefined.anubis** is the following:

```
type RGB:
   rgb(Int8 red,
       Int8 green,
       Int8 blue).
```

The formal meaning of this definition is the following. We have a new type whose name is `RGB`, and it contains all possible triplets of 8 bits integers. Of course, beyond this formal definition, there is an intention. The intention is to represent colors, and the three 8 bits numbers represent the intensities of red, green and blue in this color. Of course, the compiler does not understand intentions. It understands only the formal aspect. Nevertheless, the fact of defining this type as above make it understand part of our intentions, because a datum of type `RGB` is not a datum of the anonymous agglomeration type `(Int8,Int8,Int8)`, since these two types are distinct. No confusion nor any transtyping is possible.

Notice that in this definition we have introduced four symbols in addition to the name of the type, namely the symbols: `rgb`, `red`, `green` and `blue`.

`rgb` is called the constructor of the type `RGB`, because it enables to construct data of this type. For example, one may write down the following definition:

```
define RGB black = rgb(0,0,0).
```

which turns the symbol `black` into a datum of type `RGB`.

On the contrary, the other symbols are destructors. They may be applied to a datum of type `RGB`, and will return a datum of type `Int8`. For example, the term:

Copyright © Alain Prouté 2005–2006

```
green(black)
```

is of type `Int8` (and actually has the value 0, after our definition of `black`). We will see later that `red`, `green` and `blue` are actually three functions from the type `RGB` to the type `Int8`. In mathematics, these functions would be called canonical projections.

Notice that regardless of the size of the agglomerations (i.e. the place they take into the memory of the computer), there is no need to allocate memory to create them, and also no need to free that memory. The Anubis compiler manages the memory through a system called garbage-collector, and you never have to worry about that.

The types which are involved in an agglomeration type are called the components of this agglomeration type, and the data which are agglomerated are called the components of the agglomeration. For example, the three components of the datum `black` are `0`, `0` and `0` all of type `Int8`.

2.3.5 Alternatives

We have already seen enumerated types and agglomeration types. We will now see the general concept which encompasses all the previous, that of type with alternatives. The alternatives are the first element of the realisation in Anubis of the concept of sum in Category Theory.

In order to define the data of a new type, there are good reasons not to restrict ourself to only one way of agglomerating data. For example, imagine that we need a type of messages. At first glance, one may think that the primitive type `String` may do the job. However, remember that it is preferable to give to the compiler as much informations as possible on our intentions. This is how we can make less errors. So, one may define a type of messages as follows:

```
type Message:
   msg(String text).
```

This is a named (non anonymous) agglomeration type with only one component type. The constructor is called `msg` and the destructor is called `text`. Actually, the type `Message` is just some kind of clone of the type `String`. In mathematics, we would say that they are isomorphic, which means that it is possible to convert a datum of type `String` into a datum of type `Message`, and conversely, without loosing information. The two conversion functions are of course `msg` of type `String ->Message` end `text` of type `Message ->String`.

However, it may be the case that our messages are of two kinds, say positive messages which confirm that some operation succeeded (and which we may print in green), and negative messages, which are error messages (which may be printed in red). It is very easy to modify the type `Message` in order to take these subtleties into account:

```
type Message:
   ok     (String text),
   error  (String text).
```

Now, we have two constructors whose respective names are `ok` and `error`, and only one destructor whose name is `text`, which can be applied to any message (positive or negative), and which returns the text of the message (of type `String`).

`ok(String text)` and `error(String text)` are called the alternatives of the type `Message`. A datum of type `Message` belongs either to the first alternative (positive message) or to he second alternative (negative message), but for sure not to both. In mathematics, we would say that the type `Message` is the disjoint union os the types `String` and `String`.

We are not obliged to put the same components and of the same types in the alternatives, neither the same number of components. We may consider for example that our error messages involve two integers (one for the line and one for the column) indicating the position of the error in some source text. In this case, we would define the type `Message` as follows:

Copyright © Alain Prouté 2005–2006

```
type Message:
   ok    (String text),
   error (String text, Int32 line, Int32 column).
```

In this case, we still have two constructors named `ok` and `error`, but still one destructor named `text`. Indeed, the components named `line` and `column` have no meaning for positive messages. This is why the compiler does not generate these destructors, because it generates only concepts which are meaningful in all circumstances. There is no notion of exception in Anubis. We will see later in the section explaining conditionals, how and in which circumstances the values of the components named `line` and `column` may be recovered.

Notice that enumerated types are nothing else than types with alternatives, whose alternatives have no component at all. Of course, we may mix alternatives with no component and alternatives with components in a single type definition. For example, a function whose purpose is to read the content of a file (supposed to be of type `String`), could return a result of the following type:

```
type ReadFileResult:
   file_not_found,
   error_while_reading,
   ok(String content).
```

In this type, the first two alternatives (which correspond to error situations) have no component, while the last one has a component of type `String`, which will be the content of the file read. Here again, there is no destructor named `content`, and the content of the file can be accessed only from within an appropriate conditional, as we shall see later. It may be the case that this example enlightens the reason why there is no notion of exception in Anubis, in particular for programmers trained to languages with exceptions (like Java). Another way of understanding this is to consider that exceptions are treated through the way we type the result of operations.

2.3.6 Inductive types

With the methods we have seen up to now, we were able to define only finite types (except if we use already infinite primitive types like `String`). Now, we will see how to define infinite types using only finite types, or even no type at all. Here are the inductive types (also called recursive types).

The simplest possible example is that of Peano numbers. This type may be defined as follows:

```
type Peano:
   zero,
   succ(Peano pred).
```

Here the novelty is the presence of the name of the type itself (here `Peano`) among the components in its own definition. In this definition the intention is the following. The problem is to represent natural integers 0, 1, 2, 3, ... What this definition says is that a Peano number is either `zero`, or the successor (abbreviated here into `succ`) of a Peno number. It is furthermore in this case the successor of its predecessor (abbreviated here as `pred`).

In this type there are a priori an infinity of data. Indeed, on may consider at least the following, to which we give at the same time a more handy name:

```
define Peano _0 = zero.
define Peano _1 = succ(_0).
define Peano _2 = succ(_1).
define Peano _3 = succ(_2).
```

and we could continue like this indefinitely. Of course, this definition is rather irrealistic for computations, despite the fact that it works perfectly well as you can convince yourself by compiling the file **anubis/library/examples/peano.anubis**, and by testing the examples it contains.

Copyright © Alain Prouté 2005–2006

A more realistic example is that of lists. For the time being we consider only lists of character strings. The type may be defined as follows:

```
type ListString:
   empty,
   non_empty(String head, ListString tail).
```

The type is inductive because it is the second component in its own second alternative. What this definition says is that a list of character strings is either the empty list `empty`, or a non empty list, and in this case, it is made of a head (`head`), which is of type `String`, and a tail (`tail`) which is of type `ListString`.

Of course, the compiler does not generate destructors named `head` or `tail`, because these destructors would have no meaning when applied to the empty list. Recovering the head and the tail of a list may be achieved only from within an appropriate conditional as we shall see later. We also saw that there is in Anubis a handy notation for lists, which avoids the use of too many symbols.

2.3.7 Schemas

It is often the case we have to define types some components of which could be of any type. In this case, we can use a type parameter as the type of this component. A type parameter is a notation which is supposed to represent an arbitrary type. Syntactically, a type parameter is just the concatenation of the character `$` (dollar) and a symbol beginning by an uppercase letter. For example, `$T`, `$U` and `$Data` are valid parameter names. A type definition which is using parameters is called a schema of definition of type (or a definition of a schema of type). The notion of schema exists in mathemaics (for example, we have schemas of axioms), and it is known in C++ under the name of template.

A particularly important example is the schema `Maybe`. It is defined as follows in **predefined.anubis**:

```
type Maybe($T):
   failure,
   success($T).
```

Remark that the parameter `$T` is declared between a pair of parentheses just after the name of the schema (here `Maybe`)and before the character **:** (colon).

The intention in this type definition is to represent hypothetical data of type `$T`. If the datum does not exist, it is represented by `failure`. If it exists, it is represented by `success(d)`, where `d` is that datum of type `$T`.

For example, the division of integers (of type `Int32`) defined in **predefined.anubis** does not return a datum of type `Int32`, but a datum of type `Maybe(Int32)`. Indeed, if the division is a division by zero, the result is `failure`, and otherwise, it is `success(q)`, where `q` is the wanted quotient.

One may use several parameters in a schema of type definition. For example, one may also find in **predefined.anubis** the following schema of type, which is a variant of `Maybe`:

```
type Result($E, $T):
   error($E),
   ok($T).
```

Here, the intention is that `$E` is a type of errors and that `$T` is a type of actual result. There are many examples of use of this schema in the files of the library.

Here is another example, still taken from **predefined.anubis**, that of lists, which also makes use of exceptional syntaxes:

```
type List($T):
   [ ],
   [$T . List($T)].
```

Copyright © Alain Prouté 2005–2006

It just says that a list (of any type of data `$T`) is either the empty list (`[ ]`) or a non empty list `[a . b]`, where `a` is of type `$T` and `b` of type `List($T)`.

Type parameters introduce some kind of ambiguity, which is called parametric ambiguity. For example, the symbol `failure` alone has in general no precisely defined type. We only know that its type is `Maybe(T)` where the type `T` is still unknown. The Anubis compiler wants to solve all kinds of ambiguities. To that end it uses the method known as unification for the determination of types to be put in place of the parameters. In most cases, it is not necessary to give him particular informations, but it may be the case that some ambiguities remain. It is then enough to type explicitly one or several well chosen terms (which is not always obvious) in order to solve these ambiguities.

2.3.8 Functional types

The Anubis language also has functional types, in other words, data types whose elements are functions. Actually, in Anubis, functions are just data like any other, and of course like all data they need to have a type. Hence the need for functional types.

A function is some kind of method computing a datum (say of type `U`) from a certain number of data (say `n` data of respective types T_1,...,T_n). The type of such a function is denoted like this:

$$(T_1,\dots,T_n) \to U$$

The types T_1,...,T_n are called the source types of this functional type, and the type `U` is called the target type of this functional type. If there is only one source type (say `T`), this functional type may be written:

```
T -> U
```

(without the pair of parentheses).

Of course, since functional types have the same rights as other types, it is allowed to use them as components in types with alternatives, or as source or target types in functional types.

Here is a realistic example within which a functional type is the target type of another functional type. Imagine that we need to construct a tool taking as its first argument a datum (say of type `Language`) symbolically representing a natural language, and as its second argument a datum (say of type `Text`) symbolically representing some text. This tool is supposed to return a character string which is the text represented by its second argument written in the natural language represented by its first argument. Of course, one may construct this tool as a function of type:

```
(Language,Text) -> String
```

But, it may be the case that we have to change much more often the text than the language. In this case, we should better define our tool in its curryfied version, in other words as a function of type:

```
Language -> (Text -> String)
```

Indeed, if this tool is called for example `f`, and if `l` is a language, then `f(l)` is a function of type:

```
Text -> String
```

that we may apply successively to several texts which will all be realized into the language represented by `l`. That way we get rid of the language for the rest of the programming.

Note for those who know about object oriented programming: It is easy to make object oriented programming in Anubis. An object is essentially an agglomeration of methods, and methods themselves are functions. Hence,

Copyright © Alain Prouté 2005–2006

it is enough to define a type whose component types are functional types. Using only one alternative is enough for simulating the notion of class (as known in object oriented programming). A datum of this type will be a perfectly respectable object. It is not necessary to include state variables for the object as components of the above type. It is preferable to define the methods in a context where these variables are accessible, so that these functions will have access to the variables. We will come back to this discussion on object oriented programming later in other sections.

2.3.9 The keyword 'type'

The keyword **type** is used as we already saw to introduce a definition of type. In this section, we summarize the rules for using this keyword, and give some extra informations.

So, a type definition (or a definition of a schema of type) has the following form (what is indicated between brackets [ ] is not mandatory):

```
[public] type [(<parameters>)]:
   <alternative>,
   ...
   <alternative> <terminator>
```

<parameters> is a non empty sequence of type parameters separated by commas, for example $T,$U. The number of alternatives is arbitrary (including zero). *<terminator>* is either a dot (indicating the end of the definition) or a comma followed by three suspension dots indicating that the definition is continued in another paragraph (of the same file). For example, one may define a type in two paragraphs like this:

```
type T:
   a,
   b,...     it's not finished

   here comments and other paragraphs

type T:
   c,
   d,
   e.       end of definition
```

It is required to do like this for defining cross recursive types, because one cannot use the name of a type before it has been declared. One may also use this feature for defining opaque types in a module. One puts only the first line of the definition in the public part of the module, and the very definition is put in the private part.

<alternative> may have one of the following forms:

- a symbol (singleton alternative),
- a symbol followed by declarations of components separated by commas, and within a pair of parentheses,
- an exceptional syntax.

In the case of an alternative with components:

```
<symbol>(<component>,...,<component>)
```

each *<component>* is either only a type (anonymous component) or a type followed by a symbol (named component). The exceptional syntaxes are the following:

- `[ ]` (exceptional symbol)
- `[<component>. <component>]`

Copyright © Alain Prouté 2005–2006

- <component>+ <component>
- <component>* <component>
- <component>^<component>
- <component>|<component>
- <component>& <component>
- <component>-><component>
- <component>= <component>
- <component>=><component>
- <component><<<component>
- <component>>><component>
- <component>- <component>
- <component>/ <component>
- <component>(mod <component>)
- <component>/= <component>
- <component>=<<component>
- ~<component>

None of these syntaxes has a predefined meaning. These are only syntactical variations for the convenience of writting.

Notice finally that the compiler makes the destructors (called implicit) automatically when it is possible, which means that it writes itself (not in your source file of course, but only into its memory) the paragraphs which define these destructors. For the definition of a type `T`, a destructor named `d` of type `T ->U` is automatically defined if all the alternatives of `T` have a component of type `U` whose name is `d` (the syntax being exceptional or not). The role of this destructor is of course to extract that component from any datum of type `T`.

2.4 Terms

Terms are the expressions of the language which are used for representing data. Since each datum has a type, each terme has also a type, which is the type of the datum it represents. The datum represented by a term is also called the value of this term. In contrast with some languages (like for example the C language) within which there are expressions without any value (whose only purpose is to produce an effect), in Anubis all terms have a value (do they produce an effect or do they not). When this value has no interest, it is in general of type `One`.

Each term has only one type (taking into account the context within which it is found). Even if a symbol may be overdefined (may receive several definitions of different types), only one of these definitions may be the right one at a given place in the text. If it were not the case, the symbol would be ambiguous. The same applies to all terms. Even if a term taken as a standalone entity has several interpretations, one and only one of them should be compatible with the the surrounding terms. The Anubis compiler asks for all ambiguities to be solved. If at the end of the analysis of a paragraph, there are terms for which several interpretations are still possible, the compiler produces an error message.

2.4.1 Applicative terms

As we already saw, functions may be applied to arguments. Actually, this is part of the justification of functions. A term made of a function followed by arguments between parentheses is called an applicative terms. For example, if the function `f` is of type `(Int32,String) ->Bool`, the applicative term:

Copyright © Alain Prouté 2005–2006

```
f(4,"gabu")
```

is of type `Bool` et represents the result of applying the function `f` to the arguments `4` and `"gabu"`.

In general, if the term `f` represents a function taking `n` arguments of respective types T_1,...,T_n, and returning a result of type `U`, and if a_1,...,a_n are termes of respective types T_1,...T_n, one may form the applicative term:

$$f(a_1,\dots,a_n)$$

and this term is of type `U`.

Important remark: When the term $f(a_1,\dots,a_n)$ is computed, the arguments are computed in an arbitrary order. So, there is no warranty that a_1 is computed before a_2, etc... As a consequence, it is recommended to avoid any side effect within the arguments of an applicative term. In order to master evaluation orders, see the sections on conditionals and local definitions.

2.4.2 Conditionals

In Anubis, the conditionals (terms which begin by the keyword `if`), are structurally linked to the types with alternatives. The reason for this is simple.

In order to use a datum belonging to a type with alternatives, it is certainly necessary to distinguish several cases according to the alternative to which the datum belongs. It is the role of conditionals to provide the syntax which will enable to take this requirement into account. All conditionals contain a term (placed just after the keyword `if`) which is called the test of the conditional. So, it begins like this:

```
if t
```

where `t` is the test term, whose type must be a type with alternatives (regardless of the number of its alternatives). After the test, one writes the keyword `is` and, enclosed into a pair of braces, the cases of the conditional. So, the conditional has the following form:

```
if t is
   {
     case_1,
     case_2,
      ...
     case_n
   }
```

if the type of the test has `n` alternatives.

Now, we must have exactly one case for each alternative of the type of the test. This is logical because one must be able to handle all the situations. The Anubis compiler will also require that the first case handles the first alternative, the second case handles the second alternative, etc...

Each case is itself syntactically made of two parts: the head of the case, and the body of the case. They are separated by the keyword `then`. So, our conditional looks like this:

```
if t is
   {
     head_1 then body_1,
     head_2 then body_2,
      ...
     head_n then body_n
   }
```

Copyright © Alain Prouté 2005–2006

Right now, we remark that the bodies of the cases are terms, that they must be all of the same type, and that this type will be the type of the conditional itself. Otherwise, they are arbitrary terms.

Now, let's see the syntax for the heads of cases. Each head of case mimics the syntax of the corresponding alternative. We will have a look at the following three situations: that of a singleton alternative (i.e. without component), that of an alternative with components, and that of an alternative whose syntax is exceptional. Hence, imagine that we have defined the following type:

```
type T:
   a,
   b(Int32 x, String s),
   [T y . String u].
```

which of course does not correspond to any intention except that of having an example gathering our three syntaxes. Now, let `t` be a term of type `T`. One may write the following conditional:

```
if t is
   {
     a       then false,
     b(x,s)  then true,
     [y . u] then true
   }
```

which could be of type `Bool` (or maybe of type `Ternary` !).

One may remark that the heads of cases have exactly the same syntax as the corresponding alternatives, except that the types of the components have been droped. It is logical to omit the types of the components, because the type of the test being known, the types of the components are also known. Nevertheless, it is not forbidden to indicate the types explicitly, and this may in some circumstances make the text easier to read. Furthermore, the names of the symbols which appear as arguments in the heads of the cases may be chosen freely. So, one may rewrite this conditional like this:

```
if t is
   {
     a               then false,
     b(Int32 i, s)   then true,
     [a . String b]  then true
   }
```

The symbols which are printed in blue above, are called resurgent symbols. They are declarations. The heads of the cases declare symbols for representing the components of the datum represented by the test. You may remark that the syntax of conditionals is such that these declarations can be made only in a part of the text where it is certain that these components exist, and this is the main reason why Anubis has a so high degree of programming safety.

Of course, if one declares symbols, are they resurgent, one has eventually the intention to make use of these symbols, what we did not do in the example above. The resurgent symbols declared in the head of a case can be used only in the body of the same case (of course, you already guessed that). Below is a fairly classical example of a paragraph containing a conditional:

Copyright © Alain Prouté 2005–2006

```
define Int32
   length
     (
        List($T)    l
     ) =
   if l is
     {
       [ ]      then 0,
       [h . t]   then 1 + length(t)
     } .
```

This is the function which computes the length of a list. You may remark that the resurgent symbols `h` and `t` are accessible only when it is certains that the list represented by `l` is not empty. So it is impossible in Anubis to take the head or the tail of an empty list, which is in glaring constrast with most other programming languages.

2.4.3 Abbreviations and selective conditionals

In some circumstances the syntax of conditionals may be simplified (abbreviated).

- Conditionals with only one case. If the type of the test has only one alternative (it is an agglomeration type), the pair of braces in the conditional may be omitted. For example, if you want to permute the components in a color (of type `RGB`), you may define the following function:

```
define RGB
   permute
     (
       RGB     c
     ) =
   if c is rgb(r,g,b) then rgb(g,b,r).
```

- Test of type `Bool`. If the type of the test is `Bool` the conditional should normally be written like this:

```
if t is
   {
     false  then body1,
     true   then body2
   }
```

It may be abbreviated into:

```
if t then body2 else body1
```

Notice that the two bodies are not in the same order.

- Test of type `One`. If the type of the test is `One` the conditional should normally be written like this.

```
if t is
   {
     unique then body
   }
```

It may be abbreviated into:

```
t; body
```

Copyright © Alain Prouté 2005–2006

Anubis also recognizes selective conditionals which are not abbreviations of general conditionals as described above, but another kind of conditionals. It is sometimes the case that we have to consider a conditional the type of the test of which has many alternatives (for example more than 50), and that we are interested in giving a particular treatement to data belonging to one alternative and a default treatment to data belonging to all other alternatives. In this case we may use a selective conditional. It has the following form:

```
if t is head then body else term
```

Normally, the compiler will recognize the alternative which is concerned. Anyway, if this is ambiguous, the compiler will complain. Remember that you are allowed to give the same name to two alternatives in the same type provided that something differs in the components.

2.4.4 Local definitions

The purpose of a definition is to give a name to a datum. Of course, in order to set a definition one must not only choose a name, but must also be able to write down a term representing the datum to which one wants to give that name. For example, the paragraph:

```
define String ln = "The Anubis language".
```

gives the name `ln` (a symbol) to the datum `"The Anubis language"` (a character string).

The above definition is not local because it has the form of a paragraph, and consequently is valid in the rest of the source file within which it is written, and would be valid in the whole program if we add the keyword **public** in front of the keyword **define**. The part of the source text from within which a definition is visible (hence usable) is called the scope of this definition.

A local definition is a definition whose scope is limited to just one term (hence does not pass beyond the limits of a single paragraph). So, its use is internal to the paragraph within which it is found. A local definition is introduced by the keyword **with**. If `x` is the name one wants to give to `a`, and if `t` is a term, one may write:

```
with x = a, t
```

Notice that `with x = a, t` is a term, and that its type is the type of `t`. The term `t` is the scope of this local definition. In other words, the symbol `x` defined after the keyword **with** is usable only within the term `t`.

For example, imagine that we need a list of character strings like this one:

```
define List(String)
   chapters
     =
   ["Chapter 1",
    "Chapter 2",
    "Chapter 3",
    "Chapter 4"].
```

We will preferably write this:

```
define List(String)
   chapters
     =
   with s = "Chapter ",
   [s+"1", s+"2", s+"3", s+"4"].
```

That way, we write down the word Chapter only once, which, beyond the fact that we get less tired, is the best way of not writting it each time in the same way. It is clear that this definition of the symbol `s` has no reason to persist after the writting of the list `[s+"1", s+"2", s+"3", s+"4"]`, hence that a local definition is perfectly justified. A definition of `s` in the form of a paragraph would just encomber the source text.

Copyright © Alain Prouté 2005–2006

The use of a local definition may make the text easier to read, not only because it allows the separation of a concept, but also because the fact of giving a name to something enlightens the reader on the intention. When a term is going to be rather complex, it is preferable to begin by giving names to the data represented by one or several subterms, and to use these names instead of the subterms.

One may need to give names to several data. One may write something like:

```
with x = a, with y = b, with z = c, t
```

However, in this case, only the first occurence of the keyword **with** is mandatory, and we may simplify the above as follows:

```
with x = a, y = b, z = c, t
```

Notice that the term `b` may use `x`, that the term `c` may use `x` and `y`, and of course that the term `t` may use `x`, `y` and `z`.

When the term

```
with x = a, t
```

is evaluated, `a` is always evaluated before `t`. So, local definitions enable the mastering of evaluation orders, which is necessary when side effects are involved.

2.4.5 Functions

A function may be seen as the description of a computation using data that we have not currently at hand, but that we will get later.

So, we are obliged to use symbols instead of the missing data, and we must first declare these symbols, and use them in the description of our computation. This is why all definitions of functions have a declaration part, within which the names and types of the missing data are declared, and a body of definition which is a term within which the declared symbols may be used.

In Anubis, functions may be defined in two ways: either at the top level, in other words in the form of a paragraph, or with the help of the sign:

```
|->
```

(an arrow with a vertical bar at the left end), which is the constructor of functions currently used in mathematics.

Let's see an example. Consider a function which takes a character chain as its unique argument and produces the concatenation of the prefix `"p_"` in front of this character string. At the top level, i.e. using a paragraph, the function may be defined as follows:

```
define String
   put_prefix
     (
       String s
     ) =
   "p_"+s.
```

Then, we can for example write the term `put_prefix("gabu")` whose value will be `"p_gabu"`.

To get this result, we may also avoid to defined the function, and construct it directly within the text just for a local use. Indeed, the term:

Copyright © Alain Prouté 2005–2006

```
((String s) |-> "p_"+s)("gabu")
```

also has the value "p_gabu", since it represents the application of the function (String s) |->"p_"+s to the character chain "gabu".

The general syntax for using this arrow is as follows.

$$(T_1\ x_1,\ \ldots,\ T_n\ x_n)\ |\!-\!\!>\ \textit{body}$$

where $(T_1\ x_1,\ \ldots,\ T_n\ x_n)$ is the declaration of the missing data (each T_i is a type and each x_i is a symbol), and where the term *body* is the body of the definition of the function (within which the symbols x_i may be used). If *body* is of type U, the function represented by the above term is of type:

$$(T_1,\ldots,T_n)\ \text{->}\ U$$

You will have remarked that the two arrows |-> (used to construct functions) and -> (used to construct functional types) are distinct.

2.4.6 The keyword 'define'

The keyword **define** is used, as we already saw, to introduce definitions of data. Actually, it may also be used for declaring data.

So, a definition of datum has the following form (what is indicated between brackets [ ] is not mandatory):

```
[public | global] define <type>
    <name>
       [<arguments>]
    =
    <term>.
```

where [public |global] means that one of the two keywords public or global may be used.

<*type*> represents:

- the type of the datum to be defined if <*arguments*> is not present,
- the target type of the function defined if <*arguments*> is present.

<*name*> is the name of the datum defined by the paragraph. It is a symbol beginning either by a lower case letter or the underscore character.

<*arguments*> is a list of declarations of formal arguments for a function definition. At least one argument must be present. Declarations of arguments must be of the form T x where T is a type and x is a symbol (naming arguments is mandatory).

Finally, <*term*> is a term of type <*type*> which is the body of the definition. Don't forget the (mandatory) dot ending the definition.

Anubis has a number of unary and binary signs (exceptional syntaxes) which may also receive definitions (we do not define only symbols). For example, the sign + may be defined as follows:

```
define T
   U x + V y
     =
   <term>.
```

Then, if a is of type U and if b is of type V, the term a+b is a valid term of type T. The same principle applies to the following binary signs:

Copyright © Alain Prouté 2005–2006

```
*   ^   |   &   <<   >>   -   /   <   =<   /=
```

Notice that the signs `>` (greater than) and `>=` (greater than or equal to) cannot be defined directly. They are automatically defined when `<` (less than) or `=<` (less than or equal to) are defined. For example, you can give a meaning to `=<` using the following definition:

```
define Bool
   T x =< T y
     =
    <term>.
```

and immediately after write the term `a >= b` (where `a` and `b` are of type `T`), that the compiler will read as `b =<a`.

Note: Anubis also recognizes the binary sign `=>` but it is called implication.

Be the syntax exceptional or not, a definition always ends by:

```
     =
   <term>.
```

One may transform it into a declaration by replacing all this by a dot. For example, you may declare the sign `+` like this:

```
define T
   U x + V y.
```

You can declare and redeclare the same sign or symbol (same name, same type) as many times as you want, but you can define it only one time with a given type, except if you make private definitions (without the keyword **public**) in distinct files.

2.5 Concrete data

All the data we saw up to now are abstract data. This roughly means that they are independant of space and time. In particular they have no state, which could change in time. For example, character strings, numbers, agglomerations of abstract data, functions involving only abstract data etc... are abstract data.

However, since our computers also need to interact with the real world, we have to consider concrete data. For examples, network connections, files, windows on our graphic screen, etc... are concrete data.

One way of testing if some kind (type) of data is abstract or concrete is to wonder about the notion of equality for such data. For example, if you write the number 23, somewhere in your program, and the number 23 again at another place, no doubt that these two numbers are equal. On the contrary, if you open a network connection with a given IP address and port number, and if later you do it again with the same address and the same port number, no doubt that you have a new connection, which is not equal to the previous one.

Equality is not the sole criterion allowing the recognition of concrete data. You may for example wonder if a datum may be opened. One opens network connections, files and graphic windows, but not numbers, character strings and functions.

The need to consider concrete data is the source of non determinism in programming. If we had only abstract data to consider all our programming could be purely deterministic. The use of non deterministic programming for the manipulation of abstract data is most certainly the main source of errors in programming. It should be avoided, and performance criteria are generally bad reasons of using it.

2.5.1 Dynamic variables

A dynamic variable is just some kind of box into which you can put a datum. The content of a dynamic variable may change during execution of the program. So, a dynamic variable is at each moment in some state (hence,

Copyright © Alain Prouté 2005–2006

its name of variable). A dynamic variable is by itself a concrete datum, because it has states, but also because two dynamic variables containing the same datum may be nevertheless distinct.

Any number (not necessarily known at compile time) of dynamic variables may be opened (one preferably says created) during execution of your program. This is for this reason that these variables are called dynamic.

In order to create a dynamic variable, use the following (non deterministic) term:

```
var(a)
```

where `a` is a term. `var(a)` returns the newly created dynamic variable, and this one contains the datum represented by `a`, which is called its initial value. In Anubis, it is impossible to forget to initialize a variable, because you must provide its initial value to be able to create it.

It is often the case that the initial value of the variable must be explicitly typed, in order to avoid ambiguities. For example, if `a` is of type `T`, one creates the dynamic variable by writting `var((T)a)`. If a dynamic variable is created with an initial value of type `T`, this dynamic variable is itself of type `Var(T)`.

The role of such a variable being of holding a variable content, we need tools for reading this content or modify it. If `v` is a dynamic variable, its content is represented by the term:

```
*v
```

(which is of course non deterministic). Notice that if `v` is of type `Var(T)`, the type of `*v` is `T`. In order to modify the cotent of the variable `v`, for example, to put the value `b` into `v`, one writes:

```
v <- b
```

(again a non deterministic term). This term is of type `One`.

One may for example write down a command which creates object oriented counters, but let us begin by the type of these counters:

```
type OOCounter:
   counter(One -> Int32     read,
           One -> One       increment,
           One -> One       decrement).
```

Data of type `OOCounter` typically are objects, since they are made of three methods (functions), which allow respectively the reading, the incrementation and the decrementation of the counter, and of a state variable (the counter proper), which is well hidden, since it does not appear in the definition of the type. Now, here is the command which allows the cration of such a counter:

```
define OOCounter
   create_counter
     (
       Int32 initial_value
     ) =
   with v = var(initial_value),
   counter((One u) |-> *v,
           (One u) |-> v <- *v+1,
           (One u) |-> v <- *v-1).
```

The above method is ideal for creating objects, since only the methods are visible, whilst the state variables are absolutely inaccessible directly, even if we know the definition of the function creating the object. Hence, the encapsulation of what need to be hidden is perfect, in contrast with what happens in most object oriented

Copyright © Alain Prouté 2005–2006

programming languages. Notice that it is the full functionality of the system (i.e. the behavior of the arrow |->) which brings this advantage, not the object oriented methodology itself.

Of course, there is no need to destroy this object. The garbage-collector will do the job at the right moment, i.e. at the moment the object is no more accessible to the program.

2.5.2 Multiple dynamic variables

A multiple dynamic variable is a dynamic variable which may contain several data (instead of only one as for the simple dynamic variables we saw previously). Actually, multiple dynamic variables in Anubis are equivalent to arrays that one may find in other languages. Nevertheless, we prefer to call them variables because of their capacity of changing their content.

In order to create a multiple dynamic variable, use the following term:

```
mvar(n,a)
```

where `n` (of type `Int32`) is the number of places you want to have in your multiple variable, and where `a` is the initial value for all these places. If the type of `a` is `T`, the type of the multiple dynamic variable created above is `MVar(T)`. Note: if `1` is less that `1`, the system creates a multiple dynamic variable with only one place.

Of course, as for simple dynamic variables, there are tools for reading the content of places or modifying it. Places are numbered by integers of type `Int32`. If `n` is the number of places in the multiple dynamic variable `v`, the valid place numbers are the integers between `0` and `n-1`. In order to read the content of the place number `i` in the multiple dynamic variable `v`, use the following term:

```
*v(i)
```

(which is for sure non deterministic). Of course, if `v` is of type `MVar(T)`, `*v(i)` is of type `T`. If `i` is strictly less than `0`, everything works as if `i` was `0`. If it is strictly greater than `n-1`, everything works as if `i` was `n-1`.

In order to put the value `a` in the place number `i` in the multiple dynamic variable `v`, use the following term:

```
v(i) <- a
```

(also non deterministic) which is of type `One`. Notice that nothing happens if `i` is out of bounds, but this is not considered as an error.

Remark that the notation `v(i)` is not a term by itself. Only `*v(i)` and `v(i) <- a` are terms.

For a representative example of the use of multiple dynamic variables, see the file **anubis/library/tools/hashtable.anubis**.

2.5.3 Monitoring variables

Anubis has a system for monitoring dynamic variables. It is possible to attach to a dynamic variable one or several pieces of program (called monitors) which will be executed each time the value of the variable changes. That way, changes of values of variables are easily propagated in your program. This is not multitasking but it is very practical. On the other hand, Anubis is also really multitasking as we shall see later with the keyword **delegate**.

Monitors are functions of type `One ->One` for simple dynamic variables, and of type `Int32 ->One` for multiple dynamic variables. When you attach a monitor to a variable, you receive a monitoring ticket. This ticket is a datum of type `MonitoringTicket(T)` if the variable is of type `Var(T)` or `MVar(T)`. The role of the ticket is to allow you to stop the monitoring. At the time you abondon the ticket the monitor is dettached from the variable. More precisely, he monitor is dettached from the variable when the garbage-collector collects the ticket.

So, imagine that we have a simple dynamic variable `v`, say of type `Var(T)`. You may register a monitor `m` of type `One ->One` at the variable `v`, by executing:

Copyright © Alain Prouté 2005–2006

```
register_monitor(v,m)
```

This term returns a datum of type `MonitoringTicket(T)`. You should consider the type `MonitoringTicket(T)` as absolutely opaque. The only thing you have to do is to ensure that you keep the ticket the time needed. While the ticket survives, and each time the variable `v` is reaffected, the following term is executed:

```
m(unique)
```

In the case of a multiple dynamic variable, this is the same, except that the type of the monitor is `In32 ->One`. When the monitor is executed, it rceives as its argument the number of the slot which has been reaffected in the multiple dynamic variable.

In general, you should be careful not to create loops in monitoring. For example, if you attach the monitor `m` to the variable `v`, it is preferable (and logical) that the execution of `m(unique)` does not modify the content of `v`, otherwise we may enter an infinite loop of modifications, which may eventually lead to a problem of saturation of the memory. The Anubis compiler version 1 is not able to control such loops. Hence, this is under your responsability.

Another question whose solution may not be obvious is how to keep the ticket. In general, the monitoring is the consequence of the presence of objets in your program, and stopping the monitoring of a variable `v` should correspond to the destruction of an object `o`. Hence, it is logical to keep the ticket inside the object. Since the objet contains methods, which are functions, it is easy to keep the ticket within such a fonction.

To that end, assume that one of these functions is of type `U ->V` and has been defined as follows in the command which creates the objects:

```
(U u) |-> c
```

where `c` is a term of type `V`. The monitoring ticket (let's call it `t`) being created before the object `o` itself is created, one may include the ticket into the object at the time the object is created, by rewritting this function as follows:

```
(U u) |-> forget(t); c
```

We see that the function is unchanged, since `forget(t)` does not produce any effect, and since its value is forgotten by `forget`. Nevertheless, as long as the function survives (i.e. as long as the object `o` itself survives), the ticket also survives, since the function has a reference to it.

If several objects need to monitor the variable `v`, using the same ticket `t`, it is enough to hide `t` using the above method in each object.

2.5.4 Connections

By connection we actually mean file (on your hard disk) and TCP/IP connections. SSL connections are of another sort. Actually, all this is not very coherent, and will be unified (in the object oriented way) in Anubis 2.

The first idea at the creation of Anubis 1 was that almost any kind of data (provided their type is abstract) should be able to pass through a connection. In practice, what is explained below works only for bytes (type `Int8`). This makes no problem because you can find in the file **anubis/library/tools/basis.anubis** the tools which allow the transmission of data of any abstract type (except functions).

Connections are a priori of three sorts:

- those one may read from and write into, which are of type `RWAddr(Int8)`,
- those one may only read from, which are of type `RAddr(Int8)`,

Copyright © Alain Prouté 2005–2006

- those one may only write into, which are of type `WAddr(Int8)`.

Opening a connection

Connections are concrete data (objects). Thus, they must be created (in this case, we say preferably opened). To open a file, use one of the following syntaxes:

- `(Maybe(RAddr(Int8)))file("p",read)`
- `(Maybe(RWAddr(Int8)))file("p",new)`
- `(Maybe(RWAddr(Int8)))file("p",append)`

where, as you can see, the term is explicitly typed. Of course, you use the first syntax for getting a read only file, the second one for opening an empty file, and the last one for appending data at the end of the file.

`file` returns a datum of type `Maybe(?)` because it is not sure that the file may be opened. Of course, the result is `fialure` if the file cannot be opened for any reason. Otherwise, the result is `success(f)`, where `f` is the connection representing the file just opened.

There is no need to close the connection. THe garbage-collector will do that at the right moment.

For TCP/IP connection, it is quite the same thing. In this case, use the following:

- `(Result(NetworkConnectError,RWAddr(Int8)))connect(a,p)`

Here, the informations on the nature of the error are more detailed. See the file **predefined.anubis** for the precise definition of the type `NetworkConnectError`. In this command, `a` and `p` are of type `Int32`, and represent the IP address and the port number. There are several conversion tools for IP addresses in the file **anubis/library/tools/basis.anubis**. Again, there is no need to close the connection, which is closed automatically by the garbage-collector.

Reading from and writting into a connection

We need tools for reading bytes from connections and writting bytes into connections. The syntax is the same as for dynamic variables. If `c` is a connection, one may read a byte from this connection with:

```
*c
```

and one may write the byte `a` into the connection `c` with:

```
c <- a
```

However, the types of these terms are not the same as for dynamic variables. This is because dynamic variables are reliables while connections are not reliable. Reading from a connection or writting into a connection may fail, but it cannot fail in the case of a dynamic variable.

Thus, the term `*c` above is of type `Maybe(Int8)`, and he term `c <- a` above is of type `Maybe(One)` (this type could have been `Bool`, but it is more logical to use `Maybe(One)`). Of course, for these two terms, the value is `failure` if a problem is encountered, which normally means for a TCP/IP connection, that the connection has been closed (not by the garbage-collector, but by the user at the remote end of the connection), and for a file that there is a problem of disk.

Weakening the type of a connection

It is sometimes useful to weaken the type of a connection in order to satisfy typing constraints. If a connection is of type `RWAddr(In8)`, you can weaken the type to `RAddr(Int8)` or to `WAddr(Int8)`. This change of type is acheived by the primitive `weaken`. For example, if the connection `c` is of type `RWAddr(Int8)`, `weaken(c)` will be interpreted as being of type `RAddr(Int8)` or of type `WAddr(Int8)` depending on the surrounding terms. If the compiler cannot resolve the ambiguity, you will have to type explicitly.

Copyright © Alain Prouté 2005–2006

2.6 Programming techniques

In the subsequent sections we discuss several general programming techniques, and some general concepts. We explain how these techniques and concepts may be put at work in Anubis.

2.6.1 Determinism

The determinist/non determinist dichotomy is probably the most important notion of programming.

The terms you write in your programs always return a value (the datum that they represent), but may also produce an effect or depend on the current content of some variables (or connections). If the execution of a term does not produce any effect, and if the datum it represents does not depend on the current content of any variable, we say that this term is deterministic.

Below are some examples:

- The term $1 + 2$ is deterministic. It does not produce any effect, and its value is always 3 regardless of the values contained in the variables of your program.
- The term `*v` (where `v` is a dynamic variable) is non deterministic since what it represents depends on the content of `v`. The same is true for connections. Of course, the term `v <- a` is non deterministic because it produces an effect (changing the current value of `v`).
- The term `print("gabuzo")` is non deterministic, because it produces an effect on the console.
- The term `(String x) |->print(x)`, which constructs a function is deterministic, because it does not execute the function `print`, whilst the term `((String x) |->print(x))("gabuzo")` is non deterministic because it prints on the console.

Here is a test for checking if a term `t` is deterministic. Ask if the two terms:

`(t,t)` and `with x = t, (x,x)`

are equivalent. If they are, in other words, if you are sure that you program will behave the same way (regardless of performance questions) if you replace one by the other, the term `t` is for sure deterministic. Do that test with each example above.

Now that we have define the notion of determinism, it is time to explain why it is so important. The reason is the following: **a deterministic term is much easier to understand than a non deterministic term**, because its meaning does not depend on time nor on its environment (current values of variables, etc...). Consequently, programming deterministically as much as possible is the best way to avoid errors and to keep the program easy to maintain. Furthermore, the more deterministic the program, the more the compiler may optimize it. So, you interest is to write deterministic as much as possible.

Of course, in a program, we need non deterministic terms, because we need to interact with the environment (for example read from network connections or produce effects). Nevertheless, you should limit this programming to what is strictly necessary and to localize it into some paragraphs about which you will know that all the non deterministic stuff lies within them. If you write a program spposed to be a tool to be used by other programs (a *library module*), make sure that all the non deterministic part may be easily mastered by the user of the module.

2.6.2 Modularity

If you plan to write a big program, it is important to conceive it in a modular way. The modularity is essential for having a clear and easily maintainable program. It is required if there are several persons working on the same project.

In order to allow the creation of modular programs, Anubis provides the keywords **public** and **read** that we already explained in a previous section. They don't force you to program in a modular way, and in order to do it you need to respect several methodological rules. Nothing is dictated. In this section we just give some advices. Follow them if you think they are useful.

In a module (a piece of program with some kind of autonomy), one must clearly have two parts:

Copyright © Alain Prouté 2005–2006

- The public part, also called boundary or interface.
- The private part, also called interior or realization.

The destiny of a module is to be used by other modules. So, it provides to other modules what may be called tools. There are always two points of view on a given tool: the point of view of the maker of the tool, and the point of view of the user of the tool. The user does not care about how the tool is constructed, but only about how to use it. For this reason, each tool has directions of use.

As far as we are concerned, tools are data types or data. For a data type, the directions of use are the definition of the type, even if in some cases the name of the type is enough. In this last case, we say that the type is opaque. For a datum, what is important is its type, but also a number of properties which characterize the datum among other data of the same type (maybe not in a completely non ambiguous way). Hence, for a datum, we need a declaration of its type and a comment allowing to distinguish it from other data of the same type. For example, for a function, we need to explain what it is computing to some extend.

As a consequence, the boundary of a module must be made of:

- type definitions (not necessarily complete for opaque types), maybe with comments on the interpretation of these types,
- declarations of data, and for each one a comment for characterizing it.

Experience (or just common sense) shows that if we want to create a new module, we should first write down the boundary of the module, in other words, specify very clearly the directions of use of the module. In order to make this work, the creator of the module must identify himself with a user. In general, this part represents the bigest part of the work, because most decisions are taken in this step. It is also the most interesting. Once it's done, we need also of course to make the interior. Generally, this is less interesting, and may be delegated to someone else. If the boundary is well specified, the writting of the interior is fast and easy.

Of course, for the same boundary there are several different ways of making the interior. I general, modules are optimized by a remaking of the interior. The directions of use are unchanged. Only the performances and/or the security are changed.

In practice, the boundary of a module must be a well written, short, atractive, and well commented text, and also physically separated from the interior of the module, because the user of the module is of course pressed for time and has other things to do. For presenting a module contained (boundary and interior) in a single file, put the boundary first, and indicate clearly the end of the boundary (see the examples in the Anubis library). Of course, there are several way of useing the keywords **read** so that the user has only one **read** to put in its own file.

All this is even not enough to make a good module. A good module is a module which is easy to use, and which provides the widest possible service. These constraints are of course opposite, and you must find the right balance between them. In practice, apply the following principles as much as possible:

- Choose easilly readable public names, having a clear meaning at first glance (avoid contractions), even if this may make longer symbols.
- Show only what is needed by the user. In particular, use opaque types whenever possible.
- Use a type parameter instead of a precise type whenever possible.
- Treat only one subject per module, but if it is sometimes preferable to group in a single module small connected subjects, in order to avoid the proliferation of modules. An big number of small modules may be more discouraging than a module with a long boundary.
- Organize the boundary of the module like a book, with chapters and sections, and a table of contents at the beginning.

These principles are more or less well followed in the Anubis library itself. It may nevertheless be used as a model.

Copyright © Alain Prouté 2005–2006

2.6.3 Functional programming

The arrow |-> which is used to make functions, is a very powerful tool, the possibilities of which you may perhaps not imagine. In this section, we give several gradually sophisticated examples if its use.

Recall first that this arrow is used to construct functions. With it you may construct as many functions as you want during the execution of your program. With most languages this is not possible, and in general we have only those functions which have been textually written in the source files. Here is a first example, where we use only one arrow, but where we construct at run time as many different functions as we want:

```
define List(Int32 -> Int32)
   make_adders
     (
       List(Int32)   l
     ) =
   if l is
     {
       [ ]     then [ ],
       [h . t] then
         [((Int32 x) |-> h + x) . make_adders(t)]
     } .
```

When we compute make_adders([4,7,12,37]), we get a list of four functions of type In32 ->Int32. They are all different because the first one adds 4, the second one adds 7, etc...

Remember that functions in the Anubis language are data with the same rights as other data. There is no need to make a distinction between 'functions' on the one hand, and 'data' on the other hand. In Anubis, there are only data, and some of them are functions (which may be seen through their type). As a consequence, functions may take functions as arguments and return values which are also functions (or lists of functions as in the example above).

But there is something even more important (and probably more subtle) which is called full functionality. It is the fact that functions defined with the help of the arrow |-> remember the context within which they were constructed. For example, the term:

```
with y = (Int32)2,
     (Int32 x) |-> x+y
```

represents the function of type Int32 ->Int32 which adds 2 to its argument. Furthermore, even if this function is used in a context within which another symbol y has been defined with a different value, it remains the same function. For example, the term:

```
with y = (Int32)2,
     f = (Int32 x) |-> x+y,
     y = (Int32)5,
     f
```

still represents the function which adds 2, not the function which adds 5. Actually, the function defined by the arrow remembers the values of the symbols at the time of its creation. It must be like this, otherwise we may be faced to an error especially difficult to debug, known as capture of variable. Full functionality is a very powerful tool, in particular for object oriented programming.

Let's return to te classical example of the function map (which is defined in **predefined.anubis**):

```
define List($U)
   map
     (
       $T -> $U   f,
       List($T)   l
     ) =
   if l is
     {
       [ ] then [ ],
       [h . t] then [f(h) . map(f,t)]
     } .
```

This function takes a function (called `f` above) as one of its arguments. Notice that such a functional argument is usually called a 'callback' function is programming. `map` is very often used in conjunction with the arrow. For example, assume that we have a list `l` of character strings and that we want to concatenate the prefix `"p_"` in front of all these strings. This is easily achieved by the following term:

```
map((String s) |-> "p_"+s, l)
```

The reader who knows Lisp will have recognized the very classical construction `(mapcar (lambda ...))`, where the role of `mapcar` is played by `map`, and the role of `lambda` by the arrow `|->`.

Now, let us permute the roles of functions and arguments, and define another function `map` as follows:

```
define List($U)
   map
     (
       List($T -> $U)   l,
       $T    x
     ) =
   if l is
     {
       [ ] then [ ],
       [h . t] then [h(x) . map(t,x)]
     } .
```

In this case, instead of applying a function to all the elements of a list, we apply all the elements of a list of functions to the same argument. For example, the following term:

```
map(make_adders([1,2,3,4]),7)
```

represents `[8,9,10,11]`.

Here is another very functional example using exceptional syntaxes. A function of type `$T ->$T` i.e. from one type to itself, may be called an endofunction. An endofunction may be iterated. This means that it may be applied a first time, and reapplied to the result (since it is of the same type), and so on. We define first a function which iterates `n` times an endofunction `f` on a datum `x`:

Copyright © Alain Prouté 2005–2006

```
define $T
   iterate
     (
       $T -> $T   f,
       Int32   n,
       $T x
     ) =
   if n < 1
   then x
   else iterate(f,n-1,f(x)).
```

Now, here is the function which computes the nth iteration of an arbitrary endofunction `f`. This iteration will be denoted `f^n` (`f` to the power `n`):

```
define $T -> $T
       ($T -> $T)   f   ^   Int32   n
    =
   ($T x) |-> iterate(f,n,x).
```

Now, if we compute:

```
(((Int32 x) |-> x*2)^4)(3)
```

we get `48`, in other words `3*2*2*2*2`. Notice that it is not `2` which is elevated to the power `4`, but the function which multiplies by `2`.

2.6.4 Recursion

Principles

In order to convince you (if needed) of the importance of recursion in programming, a small historical recall will be enough. In the first half of the XXth century, Alan M. Turing invented the machines which are called after him. They are rudimentary computers inspired by the operation of typewriters. A Turing machine is an automaton with a finite number of states and above all an paper tape from which the machine may read and on which it may write. Since this paper tape is infinite, the memory of the Turing machine is itself infinite. Turing proved that his machine was able to perform all known computations, and also that it is possible to design a univeral Turing machine, able to emulate the behavior of any other Turing machine.

Other researchers at the same time invented either machines or formal systems, like for example rewriting systems, or systems based on formal grammars or even models of neural networks. The interesting point is that it was always possible to prove that all these systems were able to emulate the behavior of all the others, and thus that the notion of computability that they define are all equivalent. So, it was tempting to define the notion of computability in the absolute as anyone of the notions of computability defined by theses systems. The fact of considering that there is no wider notion of calculability tyhan this one is known as the Church-Turing thesis. This hypothesis has never been challenged up to now, and it most probably it will never be.

One of these formal systems is the theory of recursive functions. Even if this is not very difficult to explain (despite the precautions that we must always take in all questions of logic), this documentation is not the place for this. There are plenty of books on this question. We will just say that a recursive function is a function which may call itself during the computation. Beside this, the thoery allows only structural operations like duplication, erasing, exchange, etc... As a consequence, recursion is really the fundamental principle of computing, in any case, the principle without which the possibilities of computations would be very narrow. We will see in the next section that loops (that we can find in most programming languages) are just a special case of recursion, but not the general case.

Recursive programming

In a programming language like Anubis, which does not know the notion of loop, recursive programming is required. In Anubis, without recursion, you cannot do much. However, thinking recursively is not a very well

Copyright © Alain Prouté 2005–2006

spread attitude of mind, even among programmers, who generally think about loops. It must be recognized that the thought process needed to understand a loop is quite different from the thought process to understand a recursive function. In order to make this more precise, here is an example that we will realize in the C language (with a loop) and in Anubis (with a recursive function). For each program we provide a detailed analysis.

The example is intentionally very simple. We consider a function taking two arguments a and b and computing a^b (a to the power b). In order to avoid any question which could be out of the subject, we don't care about performances, we do not test the validity of input values, and consequently we just assume that b is positive or zero. Recall that a^0 is 1. Here is the version in the C language:

```
int power(int a, int b)
{
  int result = 1;
  while(b > 0)
    {
      result *= a;
      b--;
    }
  return result;
}
```

One may analyse this program as follows. Of course, we have in mind that a^b is the product of b copies of a. So, we are going to multiply a by itself as many times as needed. This will produce intermediary results which must be stored in a variable (called `result` above). The initial value of this variable is of course important. We have chosen 1 because 1 is the product of zero copies of a. Hence, it is the correct result when b is zero. This is why the loop begins by `while(b >0)`. If this condition is not satisfied, (i.e. if b is zero), the loop is not entered, and the result is the right one. Now, if b is not 0, we enter the loop. In order to be sure that we will not make a mistake about the number of passages in the loop, we need to give to b an operational interpretation. In the present case, and at the time the loop is entered, b is the number of factors a which are not yet part of the result. In order to reduce this number, we multiply the result by a, and decrement b. Then we continue by a jump to the beginning of the loop. When b becomes zero, which means, after our interpretation that no factor is missing in the result, and that the computation is finished, we avoid the loop and return the result.

Now, here is the Anubis version:

```
define Int32
   Int32 a ^ Int32 b
     =
   if b > 0
   then a*a^(b-1)
   else 1.
```

One may analyse this program as follows. This analysis is a reasoning by induction on b.

- First case, b is at least 1. By induction hypothesis, we know how to compute a^{b-1}. So, we just call the function (here ^) to make this computation. Then we just have to multiply by a, sinc $a^b = a*a^{b-1}$.
- Second case, b is zero. The result is a^0, in other words 1.

The morality os this is that justifying the recursive program is much easier than justifying the loop, and thus less prone to errors. Anyway, what makes the loop more difficult to justify is only the fact that loop (in the C language and in similar languages) require a non deterministic programming style, because a loop does not return a value, and is consequently useful only if it produces some effect. Here, the effects are produced by `result *= a;` and `b--;`. In a deterministic program, we don't have to care about current values of variables, which makes the understanding much more simple. There is the same difference between a deterministic program and a non deterministic program as between a photograph and a video.

One must also understand, and it is maybe the most important point, that in most cases, the very definition of what we want to program is recursive. This is due to the fact that the data types themself are recursive.

Copyright © Alain Prouté 2005–2006

It is nevertheless not the case here, except that `int` and `Int32` are bad substitutes for the type of tru natural integers, which is actually recursive (Peano axioms). The fact of writing loop for computing what a recursive definition says to us, is by itself an implementation work. This is low level work, and should be leaved to the compiler as much as possible. The only valid reason not to proceed like this is a need for performances. The advantage of recursive programming is that instead of implementing, we cotent ourself in making a word by word translation of the natural definition.

2.6.5 Loops

In Anubis, there is no notion of loop. The notion which replaces it is that of terminal recursion. This is special case of the recusion we saw in the previous section.

Recall that a function is recursive when it calls itself. It is the case for the following function, which computes the length of a list, and that we already encountered several times:

```
define Int32
   length
     (
      List($T)   l
     ) =
   if l is
     {
       [ ]      then 0,
       [h . t]  then 1 + length(t)
     } .
```

where the name of the function and the recursive call are shown in bold.

Remark that when the list `l` is not empty, we first compute the length of the tail `t` of `l` (through a recursive call of the function `length`), and that it is only after this call that we add `1` to the result returned by this call in order to get the length of `l`.

This call is not terminal by definition, because we still have computations to do (namely to add `1`) after the return from the call. On the contrary, in the following example (this function computes the sum of `n` and the length of the list `l`), the recursive call is terminal:

```
define Int32
   length2
     (
       Int32     n,
       List($T)    l
     ) =
   if l is
     {
       [ ]      then n,
       [h . t]  then length2(n+1,t)
     } .
```

Indeed, when the list `l` is not empty, we compute `n+1` first and then we call **length2** with `n+1` and `t` as arguments. The call is terminal because the value returned by the function itself is the same as the value returned by the recursive call, without any extra computation.

Important remark: As a consequence of the above, when a call of some function `g` by some function `f` is terminal, the target types of `f` and `g` are the same.

When a function `f` calls a function `g`, the system must remember from where in the function `f` the function `g` was called, because after the end of the execution of `g`, it is necessary to return at the right place in `f` in order to continue the computation. So, it is necessary to record a return address. This return address is stacked into

Copyright © Alain Prouté 2005–2006

a stack, i.e. an area of computer memory which may contain addresses, but also other things. This process consumes some memory which is liberated only after the return of the call.

If the call of `g` is terminal, it is not necessary to stack a return address, because we have nothing more to compute in function `f`. We may as well return directly to where we must return after the end of the execution of `f`. That way, we make the economy of stacking a return address. The Anubis compiler generates, in case of a terminal call, a code which takes into account this possibility of simplifying the returns.

So, the function **length2** defined above, never stacks return addresses (nor anything else). As a consequence, the call is without return and is just a jump to the beginning of the code of the function. So, this is really a loop, and this is how one may make loops in Anubis (and in several other languages).

Recursion as a computation principle is more powerful than loops. Indeed, one may prove (mathematically) that all loops may be translated into a recursive program, while there are recursive programs which cannot be translated into loops (except if one introduces a stack).

In general you don't have to check if your calls are terminal or not. Don't try to make them terminal systematically, because it is not always the case that it gives better performances, and for sure it may obfuscate your program. Indeed, in order to transform a non terminal recursion into a terminal one, you may have to introduce extra arguments in your recursive functions (for example in order to simulate a stack). The role of these arguments being purely technical, they don't help for the understanding of the text. Furthermore, if you have to use a stack, use the system stack which will be more efficient than a simulated stack.

Nevertheless, there are circumstances in which calls must be necessarily terminal. It is the case for infinite loops, i.e. loops from which we should never escape. For example, a server listening to the network and accepting connections, is in an infinite loop, from which it escapes only when the server itself is shutdown. In this case of course, calls must be terminal, because otherwise some memory would be irremediably consumed each time the loop is reentered. See the function £run_server in **predefined.anubis** for an exemple of such a loop, within which all calls to £run_server are of course terminal.

In the next section, we apply this principle of terminal recursion for making an automaton.

2.6.6 How to make an automaton

An automaton is a machine which has states and which reacts to events. Each time an event arrives, the automaton reacts, which produces a change of its state. Of course, there are in the scientific literature more precise definitions of automatons (often more restrictive), but this general description will be enough.

The automaton is in general started in a state which is called initial. The automaton may have several initial states, which means that it may be stated in several different ways. Furthermore, the automaton may have been designed for working indefinitely (like a beverage dispenser for example) of for stopping after some time in a state which is called final. It may have several final states, which simply means that it may produce several sorts of results. An automaton which never stops produces effects (otherwise it is not very useful). On the contrary, an automaton which stops may produce no effect, but delivers a result (like a function which returns a value).

In the theory of automatons, one uses the word deterministic. However, the meaning is different from the meaning we introduced in a previous section. A deterministic automaton is an automaton who behaves always the same way when receiving the same event in the same state. On the contrary, the state to which a non deterministic automaton will change is not always the same when receiving the same event in the same state.

Of course, this tutorial is not the place for making a course on automatons. We will only consider an example of an automaton which stops. This automaton is a lexical analyser. The job of a lexical analyser is to check if some sequence of characters satisfy some criterion. The criterion we heave chosen is that the sequence represents a floating point number in decimal notation. In other words, that:

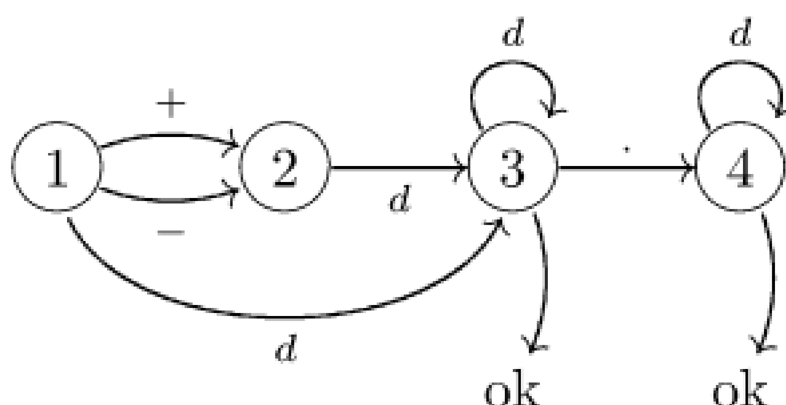
- It begins by `+` or `-` (non mandatory).
- It continues by a non empty sequence of decimal digits (integral part).
- It terminates by a non mandatory dot followed by digits including zero (decimal part).

For example, the following sequences satisfy the criterion:

Copyright © Alain Prouté 2005–2006

- `234`
- `+234`
- `-345.87`
- `12.543`

Generally, an automaton is represented by its graph. In the case of interest, the graph is the following:



The automaton has four states numbered from 1 to 4. State number 1 is the initial state. From this state start three arrows lanelled `+`, `-` and `d`. These arrows mean that if the automaton is in state 1, and if the next event (character just read from the sequence) is `+` or `-`, it changes for state 2, but if it is `d` (representing any decimal digit), it changes for state 3. The remaining of the graph is read the same way. It should be clear that it corresponds to the specifications given above, but is more precise. If, being in some state, the automaton reads a character for which no arrow may be followed, this means that the test failed. The test succeeds when one of the arrows with target `ok` is used. These arrows are used when other arrows starting from the same state cannot be used. Remark that the arrows labelled `d` starting from states 1 and 2 ensure that the integral part contains at least one digit.

After the graph is drawn, it is very easy to make the automaton in Anubis. First of all, we need a source from which characters will be read. This source may be of various kinds: network connection, file, character string, etc... In order to be independant of the nature of the source, we define the type `Source` in a perfectly behavioristic way:

```
type Source:
   source(One -> Maybe(Int8) next).
```

In other words, a source is just a function able to return the next character waiting in the source. No matter how this character is obtained. Of course, the function `next` returns `failure` if no more character may be read (end of file, connection closed, end of character string, etc...) and returns `success(c)` in the other case, where `c` is the character just read. Now, here is our automaton, which is made of function per state. All these functions call each other terminally. Since they are muttually recursive, we nned to declare them before we can define them:

```
define Bool state_1(Source s).
define Bool state_2(Source s).
define Bool state_3(Source s).
define Bool state_4(Source s).
```

Remark that they all have the same target type. This is required by the fact that all calls are terminal. Now, here are the definitions (we use a function `is_digit` for testing if a character is a digit):

Copyright © Alain Prouté 2005–2006

```
define Bool state_1(Source s) =
   if next(s)(unique) is
     {
       failure then false,
       success(c) then
         if c = '+' then state_2(s) else
         if c = '-' then state_2(s) else
         if is_digit(c) then state_3(s) else
         false
     } .


define Bool state_2(Source s) =
   if next(s)(unique) is
     {
       failure then false,
       success(c) then
         if is_digit(c) then state_3(s) else
         false
     } .


define Bool state_3(Source s) =
   if next(s)(unique) is
     {
       failure then true,
       success(c) then
         if c = '.' then state_4(s) else
         if is_digit(c) then state_3(s) else
         true
     } .


define Bool state_4(Source s) =
   if next(s)(unique) is
     {
       failure then true,
       success(c) then
         if is_digit(c) then state_4(s) else
         true
     } .
```

Remark that if it is impossible to read a character (case `failure`), we return `false` in states 1 and 2, and `true` in states 3 and 4. This is consistent with what the graph dictates.

Now, in order to test if a given source `s` passes the test, it is enough to compute `state_1(s)`, since the initial state is state number 1.

Of course, in order to test this program we need a source. Below are functions making sources from different kind of data:

Copyright © Alain Prouté 2005–2006

```
define Source
   make_source
     (
       RWAddr(Int8) connection
     ) =
   source((One u) |-> *connection).

define Source
   make_source
     (
       String s
     ) =
   with v = var((Int32)0),
   source((One u) |-> if nth(*v,s) is
     {
       failure then failure,
       success(c) then
         v <- *v+1;
         success(c)
     } ).
```

Remark that our sources are objects (in the sense of object oriented programming), because they have states (stored for example in the variable v above), and because their components are functions (methods).

2.6.7 Object oriented programming

Anubis provides the essential mecanisms allowing the application of the object oriented methodology. This methodology is perfectly justified in some cases (we already used it above), but for sure not in all cases.

Principles

Objects are by their very nature concrete data, in the sense we already explained in a previous section. An object is obviously something which is at each moment in some particular state, and this state may change in time. The role of the methods which are attached to the object is essentially reading that state and modifying it. One of the fundational principles of object oriented methodology is that of identity. It says that event if two similar objects are in the same state, they are nevertheless distinct. This is a clear confirmation that objects are concrete data. In computing we also have data which are abstract by their very nature, so that it is not good to consider everything as an object. Anyway, considering abstract data (of a mathematical nature) as objects introduced non determinism in computing at a place where it has nothing to do. This obfuscates the programming and increases the risk of errors. Of course, we what we have at hand are truely concrete data, the identity principle must apply.

Another main principle of object oriented methodology is that of classification. It is realized in object oriented languages by the notion of class. Actually, this notion looks like that of type, and if we speek about instances of classes instead of elements, as we do for types that we assimilate to sets, this is again because of the concrete nature of objects. The identity principle itself requires this notion of instances, since two objects absolutely similar may be non equal. Don't forget that an object (concrete datum) must be created. This creation is called an instantiation. It should be clear that we have two sorts of ypes with rather different properties: the types of concrete data and the types of abstract data. For example, the type `Bool` has clearly only two elements, whilst this is nonsense to say that `Var(Bool)` has only two elements even if data of this type may have only two distinct states. Speaking about elements for a type of concrete data seems to be just meaningless. Concrete data types are better seen as some king of process for making instances, and such instances may be created in any number. So, it could be beter to call concrete data types classes even in Anubis.

The third main principle of object oriented methodology is that of polymorphism. By definition, polymorphism means that an operation may have several forms dependng on classes. In practice, we realize polymorphism through a mecanism called polysemie, which enables to give the same name to sevral data (for example operations). The frontier between polymorphism and polysemie is blured and is mainly a matter of interpretation.

Copyright © Alain Prouté 2005–2006

Two operations with the same name, applying to data of different types (polysemie) may be seen as a single operation taking several forms depending on the type of the data they apply to (polymorphism).

The fourth main principle of object oriented methodology is that of inheritance. The idea is that from some (say general) calss, one may construct a new class whose objects are more specialized. They have all the methods of the first class, but the objects of the first class do not have all the methods of the second class (calld a sub-class of the first one). This notion of inheritance is as old as the world itself, and the mathematicians for example, are using it since centuries. For example, a ring is nothing else than a group with extra methods (in this case, we preferably say axioms), and of course, all the results obtained for groups apply as well to the group part of rings. This is a perfect example of inheritance.

Furthermore, object oriented methodology has a number of recurrent themes which are abstraction, encapsulation, reusability. Abstraction consists in making pieces of programs which are as much as possible independant of the rest of the program. For example, a lexical analyser must be programmed in some way completely independant of the nature of the source from which characters are read. From this point of vue, the fact that the language is fully functional plays a very important role. Indeed, full functionality allows to bring the methods to the fore, and hide the details of the realization. Encapsulation is essentially the fact of not allowing access to state variables except through the methods attached to the object. This ensures that the variables are manipulated in the right way. Finally, reusability is the fact of being able to reuse a piece of program a number of times (including for uses that were not thought about during the conception), without having to modify anything. Here again, mathematicians are the champions of reusability. Consider for example the number of uses which have been made of the theorem of Pythagoras since so many centuries.

Anubis and the objects

Let us now see how these principles apply in Anubis. The principle of identity applies to concrete data, but of course not to abstract data. For example, if you write:

```
"buzo" = "buzo"
```

you get `true` for sure, but if you write:

```
var("buzo") = var("buzo")
```

you get `false`. This is of course due to the fact that `var` creates a new dynamic variable each time it is executed. The same happens if you write:

```
("ga",var("buzo")) = ("ga",var("buzo"))
```

because, even if the first component of this agglomeration is abstract, the fact that the second one is concrete renders the aggglomeration itself concrete.

You have in Anubis all the tools needed for classifying the objects, since the typing system is very acute. From this point of view, Anubis is better than most languages, and this is essentially due to the concept of alternative.

As we already say, polymorphism is just an epiphenomenon of polysemie. Since Anubis is able of a sophisticated polysemie (in particular thanks to the type parameters), there is no problem for generating polymorphism, including parametric polymorphism.

In Anubis, there is no notion of inheritance, but it may be easily simulated, since any type may become a component in another type.

Abstraction and encapsulation are very easily applied in Anubis, especially through the use of full functionality (properties of the arrow |->), and by using type parameters. We already saw examples of this in this documentation.

The reusability of your programs is made possible by the use of the keyword `public`, which allows modular programming. In order to get reusable modules, you should separate them into two parts: a public part, in

Copyright © Alain Prouté 2005–2006

which all the tools making the interface are declared, and where you put all the comments which are useful for using the module, and a second private part within which you implement these tools. See the files in the Anubis library for examples of this presentation.

Of course, Anubis differs notably from other languages and in particular from object oriented languages. You will not find all your favorite constructions in Anubis. Nevertheless, with a small effort of adaptation, you will be able to replace them by other almost equivalent constructions, with the benefit of the high level of safety of Anubis, that you cannot find in most other languages.

2.7 Tricks

Like any other language, Anubis has his tricks. Most of them are not comming from a deliberate will of the conceptor. Most often they have been discovered by practicising.

2.7.1 How to use 'alert'.

Recall that the keyword `alert` may be used in replacement of any term (of any type). This is of course very handy when you don't know what to write down. You just write `alert` and the compiler will not complain.

Of course, the counterpart is that when this term is executed, the virtual machine executing it stops (but not the other virtual machines), and you get a message like this one on the console:

```
--> Alert in 'main.anubis', line 345, column 6 executed by machine number 56.
```

Normally, your source texts should not make use of `alert`. Nevertheless, there are at least two good ways of using `alert`.

You can use `alert` as the body of a case in a conditional, if you are sure that this case will never happen. Be very carefull because the Anubis compiler (version 1) is not able to check that his case will never happen. You do this under your responsability. Also, if your program changes, it may not be so obvious that the case will still never happen. So, this possibility is to be used with extrem care.

More interesting is the use of `alert` as a development tool. If you are at some intermediate stage in the writing of some functionality in your project, you may want to test the already written part before going ahead. For example, you have a conditional and a lot of work for each case. After the first case is written down, just put `alert` for the other cases, compile and try to execute. This allows you to test the first case. This also reminds you that some work is still to be done because during your test you will maybe execute an `alert`. This appears to be very handy for developping.

As another example, it may be the case that during the writting of a paragraph, you decide to define an auxiliary function, because otherwise the paragraph would become too big or two complicated. In this case you declare the auxiliary function just before the paragraph you are currently writting down. For example, you may write this:

```
define Int32
   my_auxiliary_function
     (
       String s
     ).
```

This will be ok for compiling, but not for executing because the compiler will complain the the function `my_auxiliary_function` has not been defined. In order to make a test of your program without having to define this function, write the following instead of the above:

```
define Int32
   my_auxiliary_function
     (
       String s
     ) =
   alert.
```

Copyright © Alain Prouté 2005–2006

You should also search for occurences of `alert` in your source text when you think your work is finished. If you find some, your work is probably not finished. To that end, you can use the option `-alert` when compiling. The compiler will produce a warning for each `alert` encountered in the source text.

2.7.2 What are the cases in my conditional ?

As you already remarked, conditionals in Anubis are used very often. Whenever you want to do something with a datum whose type is a type with alternatives (even if there is only one alternative), you use a conditional in order to access the components of this datum according to the alternative it belongs to. So, for example, you write something like this:

```
if x is
  {
  }
```

At that point you have to remember about the alternatives of the type of `x`, in order to be able to write down all cases.

The compiler may help you, because as soon as the type of `x` has been determined, he knows how the heads of all cases must be written. The above syntax is correct even if no case has been written. If you compile this, the compiler will produce a message from which you can cut the part to be pasted into your program.

For example, in order to read the whole content of a file, you write something like this (the function `read_from_file` is defined in predefined.anubis):

```
if read_from_file("my_file") is
  {
  }
```

You don't know the cases of the conditional, and you don't want to open predefined.anubis and search for the return type of `read_from_file`. This is not needed. Just compile your text as written above (with the pair of braces, so that there is no syntax error). The compiler will produce the following message:

```
Error in 'essai.anubis', line 8, column 6:
  the conditional has 0 cases,     whilst the type of the test has 3 alternatives.
  Use the following template for this conditional:

    cannot_find_file then
    read_error(ByteArray _0) then
    ok(ByteArray _0) then
```

You just have to cut the last 3 lines with your mouse, and paste them into your source text between the two braces, so that you get:

```
if read_from_file("my_file") is
  {
    cannot_find_file then
    read_error(ByteArray _0) then
    ok(ByteArray _0) then
  }
```

At that point you just have to write down the bodies of the 3 cases.

Notice that the compiler provides also the types of the resurgent symbols. You can leave them in place. The above syntaxe for the heads of the cases is correct. You can also erase the types, and rename the resurgent symbols. If the components have names (which is not the case in the above example), the compiler provides these names instead of the 'anonymous' symbols `_0`, `_1`, ...

Copyright © Alain Prouté 2005–2006

2.7.3 What is the type of this term ?

When you are reading a source text written by someone else, or a source text written by yourself but that you forgot more or less, you may have difficulties to determine the type of a term.

We assume that the source text is compiling without errors. Just add the explicit typing `(One)` in front of the term the type of which you want to know, and recompile.

If there is still no error, the term is of type `One`.

If there is an error, the compiler shows all the possible types for the term. There may be several types because at the time the error is found, not all unifications of types may have been done. If you want to know more, replace the explicit typing `(One)` by `(T)` where `T` is one of the types given by the compiler, and recompile.

Several problems may occur.

- The type `T` you choosed contains unknows, i.e. types of the form `$5654`, a dollar sign followed by an integer. You cannot write unknowns in the source text. This would generate a syntax error. Hence you must replace these unknowns by actual types.

- Due to the relative rigidity of the syntax of Anubis, a particular syntax error may occur. Assume that you have an applicative term, say `f(a)`, and that you want to know the type of `f`. If you write `(One)` in front of `f(a)`, you will determine the type of `f(a)`, because explicit typing has a very low precedence level. This will give only the target type of `f`. Hence, you will write something like:

  ```
  ((One)f)(a)
  ```

  Unfortunately, this produces a syntax error. In order to remedy to this situation, it is enough to put a pair of parentheses around `f`, like this:

  ```
  ((One)(f))(a)
  ```

  This syntax will be accepted and the compiler will show the types which are possible for `f`.

When the type of the term has been determined, it may be a good idea to leave in the source text the explicit typing of this term.

Copyright © Alain Prouté 2005–2006