epsilon

A GNU Manual
for version 0.2.1CVS, 24 January 2004

Luca Saiu

Table of Contents

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Part I - Introduction to epsilon

1 Introduction

This book is the comprehensive documentation of the epsilon functional programming language, library and tools.

In this book "epsilon" is always written in *lower case*, and even indicated as a lower case "e" in acronyms. This convention and the name of the language itself come from the idea that epsilon is intended to be a simple language with uniform syntax, easy to learn — which however does not prevent it from being powerful and expressive.

epsilon is thought to be a **production** language, useful for writing real applications. It has an elegant type system, compositional semantics, referential transparency; it will be also easy to write compilers and interpreters in epsilon. But all these features are intended to help writing applications, and were not implemented only for the sake of creating a beautiful conceptual model. Also the default library has its weight, and it will be worked on as much as possible.

epsilon is a young project, and many things still remain to be completed or rewritten. Help is welcome.

1.1 Suggestions, bug-reports and comments

This book is not finished yet. For any comment, suggestion or correction you can send us a message using the public mailing list `bug-epsilon@gnu.org`. Documentation problems are not unlike bugs, and any input from users aiming to improve the quality of this book is precious.

Please understand that Luca Saiu's English is not native, and as such it is surely far from being perfect; any reporting of misspelling, or generally of any mistake, is welcome. You can use `bug-epsilon@gnu.org` also for this.

If you need help for using GNU epsilon you can write to `help-epsilon@gnu.org`.

1.2 Audience

In this book we do not assume any previous knowledge of functional programming, nor of programming altogether; nonetheless some previous programming experience will ease reading.

The functional programming tutorial explains everything is needed to start, and the following chapters introduce concepts as they are needed. There should be no forward-references, so this book can be mostly read sequentially, from beginning to end. Some sections, however, are primarily meant as reference documentation, and you can safely just skim them in a first reading, and rewiew them with more attention at the time you actually need them. We are referring, first of all, to the chapters about Library and Internals.

1.3 History

The epsilon language was born at the end of 2001 as a small programming project (hence its name: the Greek letter ε is used in mathematics to indicate small constants[1]), a way to experiment with the implementation of functional machines. That first implementation was fully written in C (with flex and Bison), including the compiler. The code could only be executed via a virtual machine written in C for that purpose, the LVM. The LVM managed memory with a reference counter, later replaced by the Boehm-Demers garbage collector.

In the winter of 2002, while playing with the language and adding new features the author Luca Saiu became more and more impressed by the power and expressivity of the functional

[1] A late thought: ε is also used to express *errors* in numerical analysis. Fun.

paradigm and decided to make epsilon a "real" language: many important features were added at that time, including type inference, polymorphism, modules and abstract types.

Additions from the spring and summer of 2002 are garbage collector support, concrete types and exceptions. In Autumn 2002 the new abstract machine, the *epsilon Abstract Machine* or *eAM*, was started. The eAM works generating fast C code from the epsilon Abstract Machine Language (*eAML*), an intermediate code representation.

In Autumn 2002 Matteo Golfarini joined the project. In this period the eAM, `epsilonlex` and the purely functional I/O system were developed.

On 27th December 2002 epsilon was officially approved as part of the GNU Project[2]. Richard Stallman asked to enable epsilon to generate also Scheme as target code, so that epsilon can be used as an extension language for applications supporting GNU Guile[3]. Scheme generation from eAML is still at an experimental stage, but works.

The most recent important additions are the peephole optimizer and the eAM garbage collector. The collector works, is fast and reliable, but is not yet incremental and could be made parallel with relatively little work.

The eAM was essentially completed in Autumn 2003. The `epsilonlex` scanner-generator worked and was usable, and the `epsilonyacc` parser-generator was planned as the next step. Some new features introduced in late 2003 are *C-libraries* (a clean an easy way to extend the eAM with compiled, dynamically-loaeded C code), support for graphics and a library to handle S-expressions; `epsilonlex` was rewritten from scratch twice. The third implementation is much cleaner and faster than the previous ones. It still lacks the frontend, but the backend works very well. `epsilonyacc` was initially written for *SLR* grammars; it worked, but the author was not satisfied about the implementation, so he started a new rewrite from scratch. This new version supporting canonical *LR(1)* grammars is much better than the first one, and is next to be finished.

The author now plans to push the language towards the direction of Lisp, allowing runtime generation and execution of epsilon code, but retaining type-safety and the functional style. This will be the feature making the epsilon language really unique.

On 20th January 2004 around 11pm `epsilonyacc` bootstrapped[4] for the first time, followed by `epsilonlex` on 24th January, at 2:30am.

The language was influenced by ML, Haskell and Lisp, and in a minor way by the author's favourite imperative languages: Ada, Java, Python, C++, Smalltalk.

---

[2] See `http://www.gnu.org` for information about the GNU Project.

[3] See `http://www.gnu.org/software/guile` for information about Guile.

[4] When we say "bootstrap" dealing with a tool like a scanner generator or a parser generator we mean generating the scanner or parser for the tool itself.

2 Functional programming tutorial

This chapter introduces functional programming, not assuming any previous programming experience. If you already know functional programming you can just skim it.

2.1 What functional programming is

The functional paradigm is a very *high-level* programming style. "High-level" means that you program in an abstract way, "far" from the machine details and "near" your human way of thinking.

With functional programming you can safely *ignore* low-level details such as allocating and freeing memory; there is no need of using pointers or references; no need to know the internal representation of data structures. And don't even worry if you don't understand the above concepts. You will simply have *no* need of any such complication for using a functional language like epsilon.

Simplifying a bit, a program in a functional language is an *expression*, i.e. a piece of code which *computes* some value, and writes it back to you. In fact you can also use a functional language as a desk calculator. You can simply write `2 + 3 - 1`, and get `4` as result.

Of course you can also do *much* complex things: you can write a program playing chess, or drawing graphics. You will even be able to write programs which generate other programs and execute them. Reading this book you will learn, among the other things, why and when this is useful.

2.2 Mathematical foundations

For understanding the principles of functional programming you need to understand some *very basic* mathematical concepts. No advanced algoebra or analysis is needed, and this presentation will be informal.

2.2.1 Sets

A basic concept involved in most functional languages, including epsilon, is the idea of *set*. A set is a collection of homogeneous objects, such as number, words, or even real-world objects like people, houses, books. You can represent any object you can imagine in some way; the one thing you must remember is that a set is *homogeneous*: you choose some related objects to represent, and you can think of a set containing all of them.

A common example of a set is *the set of natural numbers*, written as $\mathcal{N}$. It contains *all* integer numbers starting from zero: {0, 1, 2,...}

$\mathcal{N}$ contains an infinite number of elements.

The set of *integer* numbers contains all natural numbers and also negative numbers: {..., -3, -2, -1, 0, 1, 2, 3, ...}

You can find a representation for any object you can think; for example, say you are interested in representing your collection of books (for brevity let's assume you only have three):

{*Tom Sawyer*, *Macbeth*, *Ulysses*}

This latest set is *finite*.

In a programming language, when a given object a belongs to a set A, you say that *a has type A*. "Has type" is commonly written as a colon (`:`). For example, you can write "`-27 : integer`", or "`Ulysses : book`". By convention, sets have plural names but types have *singular* names: you write `0 : natural`, and not `0 : naturals`.

2.2.2 Functions

A function is a relation from some elements in a set A and some elements of a set B[1], with one constraint: for any element a belonging to A, the function must associate it with *at most one* element b of B.

If the function f is between the set A and the set B you can say that f *maps A into B* or that f *is a function from A to B*, and write "`f : A` $\rightarrow$ `B`". It is not a coincidence that we used the "`:`" operator; the function f is itself an element of a set, i.e. has a type: the set is *the set of functions mapping A into B*.

Functions can be *applied*, i.e. they can be given an object of type A (called *argument* or *parameter*[2]); when functions are applied they *compute* some value of type B as result, and they finally *return* it.

For example, the successor[3] *succ* is a function with maps the integer set into the integer set itself. You can write "`succ : integer` $\rightarrow$ `integer`", and indeed *succ* belongs to the set of functions from *integer* to *integer*. Let's see an example of *application*: "`succ 10`" gives as result `11` (you can write "`succ` : $10 \mapsto 11$", or "`succ` $10 = 11$"). For applying a function, just write its argument after it. That's all.

If a function is undefined on one or more elements, we say it is partial, and if a is an element which f is undefined on we write "$f(a) = \perp$", or "$f : a \mapsto \perp$"; read "$\perp$" as "bottom". Partial functions are very common and useful.

Another example: let g be the function which associates a book with its author (for example, it maps *Ulysses* into *James Joyce*). g is from *book* to *author*, so we can write "`g : book` $\rightarrow$ `author`". Note that the constraint above compels us to associate one book to *at most one* author; we cannot use g if we want to describe the relationship between a book and its authors when they are more than one.

How could we solve this problem? Quite easy: let's use another function instead of g, say h, mapping the set of books into *the set of sets* of authors. For example, h maps *The Capital* into the set {*Marx*, *Engels*}: the element is mapped into only *one* element (even if this single element is a set containing two elements), so the constraint is respected. For the type, we can write "`h : book` $\rightarrow$ `set of authors`".

2.3 Lambda-notation

Let's get back to the *succ* example. How could we **define** *succ*?

A common way of defining functions is the *lambda-notation*[4]: we can define the successor as $\lambda\ n.n + 1$. This means "if we call n the argument of the function, then the value which the function computes is n + 1".

One more example: let's define the *reverse_number* function[5]; very simple: the definition of *reverse_number* is $\lambda\ x.1/x$. Note also that *reverse_number* is a partial function, since it is undefined on 0.

How can we define a function taking more than one argument? In lambda-notation you can simply write the arguments sequentially, between the λ and the .; for example, (say this function

[1] A and B can also be the same set.

[2] Formally there would be some difference between an argument and a parameter, but the difference is not important in this context. We will use these names interchangeably.

[3] The successor of a number n is n + 1; for example the successor of 34 is 35.

[4] Since the letter λ does not belong to most standard keysets, epsilon uses the character \ in its place. This convention was inspired by the language Haskell.

[5] *reverse_number* maps 2 into 1/2, 3 into 1/3, etc.

is called *plus*) $\lambda\ xy.x+y$. We can say that *plus* is a function with two arguments, or that *plus* is a function of **arity** 2.

There is another way to see the question: we could define *plus* as $\lambda\ x.\lambda\ y.x+y$; with this definition *plus* is a function which takes a parameter named x and returns another function which takes one parameter named y and returns $x+y$. This way of defining functions is called *currying*[6], and we can say that with this new definition *plus* is *curried*.

Note that with currying we can only use functions with **one** argument, without losing generality: $\lambda\ x.\lambda\ y.x+y$ is a function taking only *one* argument, x; the object returned by the function is another function, also taking only *one* argument, y.

Another advantage of currying is the possibility of *partial application*: for example we can apply *plus* to only one argument: $(\lambda\ x.\lambda\ y.x+y)7$ returns the function $\lambda\ y.7+y$ (it's nothing so strange, just a function which takes an argument and returns it incremented by 7). Of course we can also pass two arguments:

$(\lambda\ x.\lambda\ y.x+y)7\ 3$

returns 10, as expected. Just pay attention to the type:

`plus : integer` $\rightarrow$ `(integer` $\rightarrow$ `integer)`

plus is a function which takes an integer and returns a function which takes another integer and finally returns a third integer.

Curried functions are extensively used in epsilon.

2.4 A first introduction to recursion

Let's define a more complex function, the factiorial[7] function *fact*.

A way to see the factorial is this: if the argument (we call it n) is 0 then the result is 1, else the result is n times the factorial of $(n-1)$. You should convince yourself that this definition is correct: for example

$5! = 5\cdot 4! = 5\cdot 4\cdot 3! = 5\cdot 4\cdot 3\cdot 2! = 5\cdot 4\cdot 3\cdot 2\cdot 1! = 5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 0! = 5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 1$

Ok, let's write it in a more formal way: the definition of *fact* is

```
λ n . if n = 0 then
        1
      else
        n · (fact (n - 1))
```

Here is the main point: while defining *fact* we used *fact* itself. The technique we used, **defining a function using itself**, is called *recursion*.

Recursion is a very powerful tool; you can define many important and useful *recursive* functions, from simple ones such as *fact* to very complex ones.

Another simple example: the identity[8] function *id*, restricted to map integers to integers.

Let's define *id* as a recursive function. The idea is this: call the argument n; if n is zero then the result is zero, else the result is one plus $(id\ (n-1))$.

More formally, *id* is

[6] The name is in honour of Haskell Curry, the mathematician who invented this technique.

[7] The factorial of n, written $n!$, is the product of all natural numbers from 1 to n included. For example $4! = 1\cdot 2\cdot 3\cdot 4 = 24$. By definition $0! = 1$.

[8] The identity function maps an object into itself. An obvious definition is $\lambda\ n.n$

```
λ n . if n = 0 then
        0
      else
        1 + (id (n - 1))
```

You have surely noticed some similarity with *fact*. In fact this pattern is quite common in recursive definitions, even if it is surely not the only one.

Before going on with recursion we are going to make a little digression introdcing a fundamental data structure (a data structure is an object made of other objects), the *list*.

2.5 Introduction to lists

A *list* is a sequence of objects of the same type. In lists order *does* matter: for example `[1;2;3]` is different from `[2;1;3]`.

Formal definition: a list can be the *empty list* (written `[]`), or the *cons*[9] of an object a and a list L (written `a::L`).

Intuitively speaking, *consing* means adding *one* element before a list: to obtain the list `[17;-2;32]`, for example, you can cons `17` and `[-2;32]`, writing `17::[-2;32]`.

Beware of the types: you can't, for example, cons a book and a list of integers; you can only cons a book and a list of books (and obtain another list of books), or an integer and a list of integers (and obtain another list of integers). The empty list `[]` poses no type problems: you can see it as a list of integers, of books, or of any type you need in a given moment[10].

An example: you can cons 1 to the empty list:

`1::[]`

then you may cons 2 to the list you built:

`2::1::[]`

The list you obtained, `2::1::[]`, can also be written as `[2;1]`, and the previous one `1::[]` can also be written as `[1]`; they are commodity abbreviations.

A final note: re-read the definition "a **list** can be the *empty list*, or the *cons* of an object a and a **list** L". You may have noticed that we defined lists using lists: it's a *recursive* definition.

2.6 More on lists: `empty`, `head` and `tail`

Other than *cons* there are three operators for working on lists: they are called *empty*, *head* and *tail*[11]. We are now going to describe them in some detail.

`empty : (list of` τ_1`)` $\rightarrow$ `boolean`

You can read τ_1 as "any type"[12]. This is the first time you see the `boolean` type; it is a very simple yet important type: the set of *booleans* is the set containing only the two values *true* and *false*. `empty`, given a list L as parameter, returns *true* only if L is the empty list `[]`; if L is not empty then `empty` returns *false*. Two examples: `empty []` returns `true`; `empty [1;2]` returns `false`.

---

[9] The name *cons* derives from the Lisp language, and is now a universally accepted way of denoting the "*cons*truction" operation inserting an element before a list.

[10] This feature is called *polymorphism*. Polymorphism will be fully discussed later in this book.

[11] In the Lisp language (hence by tradition) they are called `null`, `car` and `cdr`, repectively.

[12] Here's one more case of polymorphism.

`head : (list of` τ_1`)` $\rightarrow \tau_1$

`head`, given a list L as parameter, returns the first element of L. It is an error to apply `head` to the empty list[13]. For example, `head [-2;450;0;3]` returns `-2`.

Notice that *head* is a partial function: *head* :`[]` $\mapsto \perp$.

`tail : (list of` τ_1`)` $\rightarrow$ `list of` τ_1

`tail`, given a list L as parameter, returns the the whole list *without* the first element. It is an error to apply `tail` to the empty list[14]. For example `tail [1;2;3]` returns `[2;3]`; `tail [17]` returns `[]` (don't forget that `[17]` is the same as `17::[]`).

tail is also partial: *tail* :`[]` $\mapsto \perp$.

Always pay attention to types: `head` takes a list of objects of some type τ_1 and returns an object of the *same* type τ_1; `tail` takes a list of objects of some type τ_1 and returns another list of objects of the *same* type τ_1. This is very intuitive: for example, if you have a list of *numbers* and extract its first element with `head`, you expect to find a *number*, and not something different.

2.7 More on recursion

Now we are going to illustrate some important concepts about recursion, analyzing few noteworthy functions in all their important aspects.

In this section the concept of *reduction* will be used for the first time. We say that an expression E "reduces to" an expression F, and we write

$E \Rightarrow F$

when computing E leads to compute F, as a single computation step[15]. For example

$3 + (7 - 2) \Rightarrow 3 + 5 \Rightarrow 8$.

Reductions are a useful mean to express computations.

2.7.1 The function *last*

We can easily compute the first element of a list using *head*, but say you want to know which is the *last* element of a list; for example, if we pass `["a"; "b"; "c"]` to the function (we call it *last*), we expect to be given back as result `"c"`.

Always start thinking of the type: *last* takes as its parameter a list of objects, each having some type τ_1, and returns a single object of the *same* type τ_1; the returned object belongs to the list, so it must have the same type as the elements of the list.

`last : (list of` τ_1`)` $\rightarrow \tau_1$

This is a definition of *last*:

```
\ x . if empty (tail x) then
        head x
      else
        last (tail x)
```

We call x the parameter.

There are two cases:

---

[13] Applying `head` to `[]` *raises an exception*. Exceptions are a general and powerful way to deal with errors, and will be dealt with later in this book.

[14] `tail []` also raises an exception.

[15] The idea of "computation step" is indeed quite subjective. In this book we say "a single computation step" to mean the minimum computation unit which is interesting in the context. The discussion about reductions will be informal.

1. x is a one-element list, i.e. its tail is the empty list:

   In this case the first element of x is also the last element of the list: we return `head x`.

2. x is *not* a one-element list, i.e. its tail is a non-empty list:

   The last element of x is the last element of its *tail*: we return `last (tail x)`.

Pay attention to the second case: it the list is not one-element (for example say it has three elements), we return the *last* of its tail, which in the example is a two-element list: the *last* of the two-element list is the *last* of its tail, which is one-element. The important fact to note is that **the arguments of successive recursive calls are more and more simple**: this is very typical when recurring over data structures[16], and a different behaviour is nearly always an indicator of errors.

Let's track down how this function works for, say, `[45; 43; -4; 35]`, step by step:

- evaluate *`last`* `[45; 43; -4; 35]`:

  is `[45; 43; -4; 35]` a one-element list? No (we are in case 2), return *`last`* `[43; -4; 35]`.

- evaluate *`last`* `[43; -4; 35]`:

  is `[43; -4; 35]` a one-element list? No (we are in case 2), return *`last`* `[-4; 35]`.

- evaluate *`last`* `[-4; 35]`:

  is `[-4; 35]` a one-element list? No (we are in case 2), return *`last`* `[35]`.

- evaluate *`last`* `[35]`:

  is `[35]` a one-element list? Yes (we are finally in case 1), return `head [35]`, i.e. `35`.

"Coming back" to the first function call we have:

`last [45; 43; -4; 35]` $\Rightarrow$

`last [43; -4; 35]` $\Rightarrow$

`last [-4; 35]` $\Rightarrow$

`last [35]` $\Rightarrow$

`35`, which is the value we were expecting.

Notice that in the definition of *last* it is assumed that the argument is not `[]`: if x is `[]` then the evaluation of `head (tail x)` fails. *last* is a partial function, being it undefined on `[]`: $last : [] \mapsto \perp$.

2.7.2 The function *interval*

Say you want to compute the list of integers from a given value to another given value; for example, if we pass `1` and `5` to the function, we expect to be given back as result `[1; 2; 3; 4; 5]`; if the first parameter is greater than the second one, then we expect the empty list `[]`; if we use the same value x for both parameters we expect `[x]`. We are going to call this function *interval*.

interval takes two integer parameters and returns a list of integers; we write it curried, so we may think of it as a function taking an integer parameter and returning another function, which takes another integer parameter and returns a list of integers:

`interval : integer` $\rightarrow$ `(integer` $\rightarrow$ `list of integer)`

interval is defined as

[16] For example lists as in this case, opposing to, say, naturals. This is not conceptually exact, but the complexity of a natural number seen as a data structure is not entirely evident.

```
\ a . \ b . if a > b then
                [] 
              else
                a :: (interval (a + 1) b)
```

As you could image *interval* is a (curried) function taking two parameters; we call them a and b, respectively.

As often happens, there are two cases:

1. Case 1: a is greater than b.

   We simply return the empty list, as the definition says.

2. Case 2: a is less than or equal to b.

   The list that we are going to return will surely contain a as the first element; the rest of the list will be the interval from $(a + 1)$ to b; so we cons a to `(interval (a + 1) b)`.

Let's examine an example of application, say `interval 5 7`, step by step:

- is 5 greater than 7? No, so we are in the second case, and we return `5 :: (interval (5 + 1) 7)`.
- Now we need to evaluate `(interval (5 + 1) 7)`, i.e. `interval 6 7`: is 6 greater than 7? No, so we are again in the second case: we return `6 :: (interval (6 + 1) 7)`.
- Evaluate `interval (6 + 1) 7`, i.e. `interval 7 7`: is 7 greater than 7? No (it's equal to it, not greater than it), so we are in the second case: we return `7 :: interval (7 + 1) 7`
- Evaluate `interval (7 + 1) 7`, i.e. `interval 8 7`: is 8 greater than 7? Yes, so we are in the first case: return `[]`.

Now we have evaluated everyhing we needed: "coming back" to the first function call we have:

`interval 5 7` ⇒

`5 :: (interval 6 7)` ⇒

`5 :: (6 :: (interval 7 7))` ⇒

`5 :: (6 :: (7 :: (interval 8 7)))` ⇒

`5 :: (6 :: (7 :: []))`, i.e. `5 :: 6 :: 7 :: []`,

which can be abbreviated into `[5; 6; 7]`, the value we were expecting to be given.

2.7.3 Tail-recursion

Notice the difference between the evaluation of *interval* and the evaluation of *last*: each call to *interval*, except for the last one, is expanded to an *expression* containing another call to itself among other things (here the expression is the cons of an object and another call to *interval*). In *last* the situation is simpler: each call, except the last one, is simply expanded to another call, with *no* other expressions involved which "surround" the recursive call:

`interval 1 3` ⇒ `1 :: (interval 2 3)` ⇒ ...

`last [345; -50; 4555]` ⇒ `last [-50; 4555]` ⇒ ...

Said in another way, you have no need to "keep track" of temporary results when evaluating a call to *last*, but you have when evaluating a call to *interval*: for computing `interval 1 3` you need to compute `interval 2 3`, then cons `1` to it; to compute `interval 2 3` you have to compute `interval 3 3` and cons `2` to it, and so on. With *interval* you have first to compute temporary values, then to "attach" them with expressions. With *last* you can simply forget all the temporary values before the last one; once you have computed them, you will not need them anymore:

`last [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16]` $\Rightarrow$

`last [2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16]` $\Rightarrow$

`last [3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16]` $\Rightarrow$

...

$\Rightarrow$ `last [16]`.

For all these reasons it's easy to see that *last* is somehow simpler than *interval* (apart from the number of parameters: this is not important in this context). *last* is said to be a *tail-recursive* function, while *interval* is not[17].

We are going to explain more formally what means for a function to be tail-recursive, later in this book. For this time just an intuitive understanding is enough.

Evaluating a call to a tail-recursive function with a computer is **noticeably more efficient** than evaluating a call to a plain recursive function: when using a functional language the programmer should strive to use tail-recursion whenever possible.

2.7.4 Non-termination: the function *dontstop*

Let's examine a recursive function called *dontstop*, defined in this way:

```
\ x . dontstop x
```

epsilon would detect (*infer* is the most appropriate term) its type as

`dontstop :` $\tau_1 \rightarrow \tau_2$,

altough the reason for this may not appear evident yet. The argument x can have any type, hence the generic type τ_1. But what is the type of the object returned by the function? The answer is that *the function never returns*, so in a sense the returned object has an unknown type; and there is no reason to suppose that the returned object has the same type as x, hence the new type τ_2. This may seem counterintuitive now, but there are deep reasons[18] (which will be explained later in this book) not to write, say,

`dontstop :` $\tau_1 \rightarrow$ ***`nothing`***.

The behaviour of *dontstop* is quite simple to understand: let's see it when applied to `[[10; 3]; [2]]` (an object with type `list of list of integer`, which is ok: the parameter x can have *any* type, as we have just said):

`dontstop [[10; 3]; [2]]` $\Rightarrow$

`dontstop [[10; 3]; [2]]` $\Rightarrow$

`dontstop [[10; 3]; [2]]` $\Rightarrow$

...,

never stopping. A call to *dontstop* expands to another call to *dontstop*, **with exactly the same argument**. The fact that the complexity of the argument does not lessens in successive calls usually indicates that evaluation does not terminate, as in fact happens in this case.

Any expression whose computation is non-terminating is said to *diverge*. If E diverges you can write $E\uparrow$. By constrast, if the evaluation of E terminates at some point then E is said to *converge*, and you can write $E\downarrow$.

dontstop is a *partial* function, being undefined for at least some values of its argument (in this case for *all* the possible values), and the meaning of *dontstop* can be expressed as $\lambda x.\bot$;

---

[17] When defined in this form. *interval* can also be defined in a tail-recursive form, but the definition becomes somewhat more complicated.

[18] These reasons are bound to the *type system* of epsilon, i.e. the set of rules which govern the typing of expressions. epsilon supports the Hindley-Milner type system, also used by the languages ML and Haskell. The enforcing of such rules prevents many programming errors and can also make programs run faster. Details will be explained later in this book.

note, however, that here the symbol "$\perp$" stands for something different from the meaning of `last []`, also expressed by "$\perp$". Here "bottom" stands for *non-termination*, in the other case it stood for *error*. The actual meaning of an expression including $\perp$ will be specified case by case, when ambiguity can occour.

As a final comment notice that *dontstop* is a tail-recursive function.

Of course *dontstop* is illustrative as an example, but such a function should never be needed in actual programs.

2.8 Differences from imperative programming

This section is aimed to the readers with some experience in imperative programming. If you are learning to program from this book you can safely skip this part.

2.8.1 No side effects

The most obvious difference between functional languages and imperative languages is that in functional languages there are *no side effects*. For example in an imperative language you can write something like `a := 67`, or, to increment a variable, `a := a + 1`. How can we increment a variable in a functional language? The answer is simply that *there is no way*; what is achieved with side effects in imperative languages must be dealt with in some other way, in most cases using recursion.

This has some advantages: when you create a variable you give it a value, and you are sure *that the value will never change*; in imperative languages, instead, it's a common mistake to think a variable has some value, while instead it has been updated, possibly by some procedure you are not thinking of: this is impossible in a functional language. It's equally impossible to have uninitialized variables: *you name it, you create it*; there will never be a uninitialized-variable or null-reference error.

2.8.2 Recursion instead of loops

The absence of loops in functional programmers captures the attention of many programmers even more than the absence of side effects; however the absence of side effects obviously implies the absence of loops: iterating means executing some command many times, but *what command are we about to execute?* There is no state to change since there are no side effects. You just keep calling recursive functions and computing values until you reach the one you are interested in, and finally return it.

Recursion is not inherently more difficult to use than loops, but requires a little adapting to "think recursively". Don't be afraid if you find yourself stuck to think "iteratively" at first: just try to express the same idea as a recursive function. In many cases, when you have finished, you will be surprised from the clearness of recursive code with respect to iterative code.

2.8.3 No pointers or references

In a functional language there is no need for pointers or references. Data structures are beautifully expressed without pointers or references[19] using abstract and concrete types, which are **very** intuitive to define and to use, and not error-prone at all.

[19] References (as in Java, or, to a lesser extent, in C++) do not to cause the same problems which pointers cause. However they are nonetheless error-prone: you can forget initializations by mistake, and in general references suffer from all the vulnerabilities which are bound to side-effects.

Concrete and abstract types will be fully dealt with, later in this book.

Semantics[20] also gives some more justifications[21] to our claims.

2.8.4 First-class functions

In functional languages a function is an *first-class* object, i.e. an object like any other: you can compute a function at run time, you can make a function return another function, you can define a function with no special syntax, you can write an unnamed lambda-expression in the middle of a bigger expression: for example

```
2 + ((λ x . x + x) 3)
```

gives back `8` as result, as expected.

For an example of a function returning a second function, think of any curried function (look at its type if you don't understand at first).

This use of functions is natural and simple, but is forbidden in nearly all imperative languages.

2.8.5 Higher-order

In nearly all functional languages functions can have other functions as parameters (we speak about *second order* in this case), and the parameters of those functions can be yet other functions (*third order*), and so on. If there is no limit to the order of functions then the language is said to be ω-*order*[22]. epsilon is an ω-order language.

You will learn later in this book how higher-order functions are useful to write simple and compact programs; in many cases you can even use higher-order functions as substitutes for recursion.

Higher-order functions don't exist or their use is seriously restricted in imperative languages.

2.8.6 Referential transparency

In functional languages a noteworthy property holds, named *referential tranparency*: in practical terms it says that if an expression E has value v, you can replace every occurrence of E with v in a program, without changing its meaning. This makes programming more clear and less error-prone and allows the compiler to make some optimizations which would be impossible in an imperative language.

2.8.7 Type safety

Many functional languages, epsilon included, are *strongly typed*, i.e. they recognize as invalid all the programs which contain type errors, such as multiplying an integer and a boolean, **at compile time**[23]. Many type errors are subtle, and in general having the compiler detecting them is a great help, saving time and frustration.

Most imperative languages allow unsafe use of types, and this may lead to detect errors very late, even after program release.

---

[20] Semantics is a branch of Computer Science dealing with the formal meaning of computer programs. It's not required you know Semantics for using epsilon.

[21] Semantics says pointers and references are related with memory, also named *store*. All side effects are also operations on stores. In a functional language there is no concept of store and all operations are made *only* on *environments*; imperative programs, by constrast, use *both* environments *and* stores. This is a deep reason why functional programming can be simpler than imperative programming.

[22] In mathematics the letter ω is used to indicate the cardinality, i.e. the number of elements, of the set of the natural numbers $\mathcal{N}$.

[23] i.e. **before** execution.

2.8.8 Type inference

Some functional languages[24], including epsilon, *infer* types; it's the **compiler** to tell the type of an expression to the programmer, and the programmer is saved from the pain of declaring, say, the type of every function parameter. The output from the compiler is a mean to verify that the meaning of the program is really what the programmer intended.

Implementations of imperative languages do not usually provide type inference.

2.8.9 Purely functional I/O

Purely functional languages, including epsilon and Haskell, completely avoid the dangers of side effects forbidding the user to mix input/output with normal computations, as a way to preserve referential transparency[25]. For example, the epsilon compiler refuses to accept code like `2 + input_integer + 3`.

Anyway not all functional languages have these restrictions. ML, for example, has some imperative features including side effects and I/O in imperative style.

Languages like epsilon and Haskell are said to be *purely functional*.

2.9 Before going on

This introduction to functional programming, even if brief, may seem too abstract at a first glance, but as the name *functional language* suggests the basic mathematical aspects are of fundamental importance: while programming in epsilon you will be defining recursive functions with complex types all the time.

If you have not understood at least the basics of types, lists and recursion you should read again this chapter, paying particular attention.

Of course any suggestion for improving this documentation is welcome, but we deem *this* chapter particularly important. You can use the public mailing-list `bug-epsilon@gnu.org` to talk with us about these matters. No subscription is needed.

However, don't be afraid if you still have some doubts; most of the same concepts which were outlined here will be presented in practical terms in the next chapter.

[24] ML and Haskell are important exemples.

[25] Other referentially transparent solutions exist in the functional world, such as *linear types*; however we decided to implement the I/O system following the lesson of Haskell, which in our opinion employs the most clean and usable way to make safe I/O.

3 epsilon tutorial

In this chapter you are going to learn the basics of the epsilon language by using the tools yourself.

3.1 Before starting: a notice

We are now assuming that the epsilon meta-interpreter is already implemented and working. This is not yet true, but you can use the temporary quick-and-dirty REPL[1] in the meantime.

You can invoke the temporary REPL simply typing `epsilon`.

In the same way, in this chapter we purposefully ignore the bytecode interpreter eVM, which is likely to disappear in the future, when the interpreter is ready.

3.2 The interpreter and the compiler

An epsilon program can be run using one of two distinct tools, which are useful in different situations:

the interactive interpreter

The interpreter[2] can be used as an interactive program: you type a small piece of epsilon code, the code is promptly executed and the result is shown back to you on the screen; then you can start again: write some more code, execute it, look at the result, and so on.

When used in the way explained above, the interpreter is said to act as a *REPL*, i.e. "Read-Eval-Print Loop": it reads a piece of code, evaluates (executes) it, prints back the result and starts again.

This way of working is very comfortable when you are developing a new program: it makes easy to write some code, to test it soon, and to fix it soon if some error is found. There is a drawback, however: the code runs slowly and uses much memory.

the compiler

The compiler is a non-interactive program: you write your epsilon code in one or more files using an editor such as GNU Emacs, and you invoke the compiler which translates the files into a form which is directly executable on your computer. Then you can execute the translated program.

Using the compiler you use an "Edit-Compile-Run Loop"[3] approach. Note that it is not a program to "loop", it's **you**; it's you who must manually edit the files, save them, compile them, wait for the translation to finish, and execute the translated program.

The advantage of this approach is the high speed and efficiency of the translated code. Its drawbacks are the slowness of the translation, and the general clumsiness

[1] The *REPL*, (Read-Eval-Print Loop), is a simple C program which takes an epsilon expression, compiles it into bytecode, and runs it on the *eVM* (epsilon Virtual Machine). The REPL as it is now has some serious flaws, and can not execute all the code which the compiler can run. You can always use the compiler instead of the REPL, at the cost of some additional complications. In the rest of this chapter we assume you are using the interpreter, which when it's ready will behave much like the current REPL.

[2] The interpreter is also called *meta-interpreter*, or *meta-circular interpreter*. This means that the interpreter was written in the same language it implements: in this case the epsilon interpreter itself was written in epsilon.

[3] Often called "Edit-Compile-Debug Loop", since it's very uncommon to write a nontrivial program which works without errors the first time.

of the approach. The compiler is the right tool to use when you have finished writing and testing a program which works well, and want it to run fast.

A program run with the interpreter (*interpreted*) or translated by the compiler (*compiled*) behaves **identically**: only speed and memory use are different. You have not to worry about compatibility, since the interpreter and the compiler support the exact same language.

The interpreter is also better suited to learn the language and experiment. In the rest of this chapter we are going to assume you use the epsilon interpeter.

3.3 Using the epsilon interpreter: a first session

Try starting the interactive interpreter: at the command prompt of your system, type

```
# epsilon
```

The interpreter will show a banner similar to

```
--------------------------------------------------------------------------------
                                       i  ll
                 eeeeee                    l                  version 0.2.1CVS
                ee      p pppp   ssss  ii   l   oooo  n nnnn
                 eeee    p    p s       i   l  o    o  nn   n
                ee       p    p  sses   i   l  o    o  n    n
                ee       p    p      s  i   l  o    o  n    n
                 eeeeee  ppppp   sses  iii lll  oooo   n    n
                         p
                         p                 http://www.gnu.org/software/epsilon
                        ppp
--------------------------------------------------------------------------------
GNU epsilon 0.2.1CVS, Copyright (C) 2002, 2003 Luca Saiu
GNU epsilon comes with ABSOLUTELY NO WARRANTY; for details type ':no-warranty'.

This is free software, and you are welcome to redistribute it under certain
conditions; type ':license' for details.

Welcome to the epsilon meta-interpreter.
Type :? for help.

1 >
```

The prompt '>' followed by a blinking cursor means that the interpreter is ready to accept your code; now try typing

```
2 + 2;
```

(remember the trailing semicolon), and pushing ⟨Enter⟩. The interpreter will answer

```
- : integer
4
1 >
```

The first line means that the expression you entered, indicated by '-', has integer type. The second line shows the computed value, which, as you were expecting, is 4. The third line is a new prompt; the interpreter is ready to accept more code.

For exiting the interpreter, type

```
:quit
```

(note the leading colon, and the absence of a trailing semicolon) and push ⟨Enter⟩ at the prompt. Another way to exit the interpreter is by pressing ⟨Ctrl⟩-⟨D⟩.

3.3.1 Interpreter syntax and epsilon syntax

Pay attention to the syntax: `:quit`, `:help` and `:license` are commands directed to the interpreter itself, in the sense that they don't deal with your epsilon program. Interpreter commands need a leading ':' and no trailing ';'. Expressions such as '2 + 2;', instead, are part of the epsilon syntax. They need no leading ':' and they do need a trailing ';'.

3.4 Queries and definitions

Now start the interpreter again, typing `epsilon` at the command prompt of your system.

Try making some computations with integers; parentheses are used to group subexpressions to be computed before, as in arithmetic. The 'times' symbol is written as an asterisk ('*'), the 'divided' symbol is written as a slash ('/').

```
1> (2 + 6) * 2 / 4;
- : integer
4
```

You can try other more complex expressions if you like.

The expression above was a *query*: you asked the interpreter to evaluate an expression for you, and you were interested in the result. Queries are a common way, among the rest, to test a function you wrote, supplying a value and verifying the result is what you are expecting. Let's now show how to define a function, starting from a very simple one.

Say you want to define a function adding 3 to its only (integer) argument: you learnt in the previous chapter that such a function is written as $\lambda\, x.x + 3$. Since the letter λ is usually not present on keyboards, epsilon uses the backslash ('\') character instead of it. So try entering

```
\ x . x + 3;
```

What you get as an answer is

```
- : integer -> integer
<function>
```

which maybe is not what you were expecting. Let's examine the answer of the interpreter: the first line says that the expression you entered has type *integer* $\rightarrow$ *integer*, which is right; the second one says that the value of your expression is a function; often doesn't make much sense to write them: they are usually complex and not very useful as *output* from the interpreter (they are useful as *input* for it). Hence the interpreter just writes '`<function>`' when the result of your computation is a function. And indeed it is, in this case.

The problem is that you wrote a function, but *it still was a query*. For a definition there is need for a different syntax. Of course this syntax exists, and it is very simple: just write, for this same example,

```
define f = \ x . x + 3;
```

The interpreter answers saying just

```
f : integer -> integer
```

Now you have given your function the name f (you could have used any different name, of course). You can now use your function f in queries and in other definitions: try

```
f 10;
```

The result is 13, as you expected.

We have shown examples of function definitions, and indeed that is the most common case, but you can make definitions for objects of *any* type, not necessarily functions. The following example shows several non-function definitions:

```
define twenty = (\ x . x * 2) 10;
define forty = twenty + twenty;
define this_is_a_string = "Hello, world!";
define pi = 3.14159265358979323846264338327;
define empty_list = [];
```

3.5 Booleans

As we already said in Chapter 2 [Functional programming tutorial], page 11, a *boolean* value (also called *truth value*) is either `true` or `false`. Booleans are useful in a wide range of contexts. One of the most simple is in a query comparing two objects: *"is 1 less than 2?"*

```
1 < 2;
- : boolean
true
```

A slightly more complex query (note that '`>=`' stands for '$\geq$', and '`<=`' stands for '$\leq$'):

```
(f 1) >= (f 2);
- : boolean
false
```

You can always think of reductions if this helps you:

`(f 1) >= (f 2)` $\Rightarrow$

`((\ x . x + 3) 1) >= ((\ x . x + 3) 2)` $\Rightarrow$

`(1 + 3) >= (2 + 3)` $\Rightarrow$

`4 >= 5` $\Rightarrow$

false

You can also directly use the constants `true` and `false`: try writing the trivial query

```
true;
```

The interpreter will answer

```
- : boolean
true
```

3.5.1 Logical connectives

You can use the usual *logical connectives* $\neg$, $\wedge$, $\vee$ and $\oplus$ with boolean expressions. You can read them as "not", "and", "or" and "xor", respectively, and they are written in epsilon using these same names: `not`, `and`, `or`, `xor`. "$\vee$" is also called "inclusive or", and "$\oplus$" is also called "exclusive or".

Let us explain the meaning of boolean connectives, where e, e_1 and e_2 are epsilon expressions with boolean type. The result has always boolean type, too.

The meaning of logical connectives is:

- $\neg e \Rightarrow$ *true* if and only if $e \Rightarrow$ *false*
- $e_1 \wedge e_2 \Rightarrow$ *true* if and only if $e_1 \Rightarrow$ *true* **and** $e_2 \Rightarrow$ *true*
- $e_1 \vee e_2 \Rightarrow$ *true* if and only if $e_1 \Rightarrow$ *true* **or** $e_2 \Rightarrow$ *true*, or both
- $e_1 \oplus e_2 \Rightarrow$ *true* if and only if one of the following two statements holds:
  - $e_1 \Rightarrow$ *true* and $e_2 \Rightarrow$ *false*
  - $e_1 \Rightarrow$ *false* and $e_2 \Rightarrow$ *true*

  (or, said in a different way, if and only if e_1 and e_2 reduce to **different** truth values).

As we said above, boolean connectives applied to boolean objects yield other boolean objects, so they can be combined to form boolean expressions of any complexity. Try the following query with the interpreter:

`(true and (not not false)) xor ((1 < 2) or false);`

The result is *true*. Let us show why:

`(true and (not not false)) xor ((1 < 2) or false)` $\Rightarrow$

`(true and (not true)) xor (true or false)` $\Rightarrow$

`(true and false) xor true` $\Rightarrow$

`false xor true` $\Rightarrow$

`true`,

which is to say that reductions apply to boolean expression as to any other type of expression.

Here are some more sample queries; try computing them in your mind or with paper and pencil using reductions before using the interpreter:

```
true and (true or false);

(1 < 2) xor false;

not ((20 < 22) and true);
```

3.6 Conditionals: `if..then..else`

In the examples of Chapter 2 [Functional programming tutorial], page 11 we used the *conditional operator* `if..then..else` several times, without explaining the details.

The syntax is

`if` *guard* `then` $expression_1$ `else` $expression_2$

where *guard*, $expression_1$ and $expression_2$ are epsilon expressions. There are two constraints:

- *guard* must have boolean type; said in a more formal way
  - *guard* : `boolean`
- $expression_1$ and $expression_2$ must have the same type, which can be *any* type. Formally,
  - $expression_1 : \tau_1$
  - $expression_2 : \tau_1$

The intuitive meaning is: evaluate *guard*; then, if it reduces to *true* then reduce the whole `if..then..else` expression to $expression_1$, else if it reduces to *false* then reduce the whole `if..then..else` expression to $expression_2$. Said more formally:

- if *guard* $\Rightarrow$ *true* then

  (`if` *guard* `then` $expression_1$ `else` $expression_2$) $\Rightarrow expression_1$
- if *guard* $\Rightarrow$ *false* then

  (`if` *guard* `then` $expression_1$ `else` $expression_2$) $\Rightarrow expression_2$
- if *guard*$\uparrow$ then[4]

  (`if` *guard* `then` $expression_1$ `else` $expression_2$)$\uparrow$

Some brief comments about the type constraints:

The first constraint is obvious[5]: for deciding between two options you need a `boolean`: any other type (`integer`, `string`, `list`, etc.) would not be the right thing.

To understand the second constraint, try entering the query

```
if 1 < 2 then 1.0 else "abc";
```

This will lead to an error, since the second constraint was violated: `1.0` and `"abc"` have different types (`float` and `string`, respectively). This is reasonable: in an actual program it

[4] If *guard*$\uparrow$ then we will never be able to choose between the `then` branch and the `else` branch: we will keep evaluating the guard for ever, without ever evaluating either branch.

[5] Even if it is absent in some languages such as Lisp and C; however nowadays it is widely known that such absence of type constraints can lead to many programming errors.

would be very difficult[6] to do something reasonable if the two *branches* (the "`then` branch" and the "`else` branch") have different types; and thare are also other reasons: which type would you give to the expression `\ x . if x then 1.0 else "abc"`? You would not be able to decide between `boolean` → `float` and `boolean` → `string`.

An example: the function *monus* is somewhat famous: it takes two numbers x and y, and returns x - y if it is not negative, else returns 0. Let's see a definition:

```
define monus = \ x . \ y .
  if x - y >= 0 then
    x - y
  else
    0;
```

Try calling *monus*:

```
monus 10 12;
- : integer
0

monus 12 (5 + 5);
- : integer
2
```

A final note for imperative programmers: if you know imperative programming, you might ask whether an `if..then` operator, without `else`, exists. The answer is a strong **no**: in a functional language an expression must always be reduceable to something: it is not acceptable to say *"if a is less than b then 10"*; and if a is not less than b, what are we going to return? An explicit `else` branch is always needed.

3.7 Temporary variable bindings: `let..be..in`

In many cases it is useful to have an "abbreviation" for a given subexpression, which is used more than once. For example, say you want to compute

$2^5 + 3^5 + 4^5 + 5^5$.

A way to compute it with a query is:

```
2 * 2 * 2 * 2 * 2 + 3 * 3 * 3 * 3 * 3 + 4 * 4 * 4 * 4 * 4 + 5 * 5 * 5 * 5 * 5;
```

The above expression is perfectly good for the interpreter, but not very readable by humans. The `let` construct allows you to write, instead,

```
let f be
  \ x . x * x * x * x * x
in
  (f 2) + (f 3) + (f 4) + (f 5);
```

The meaning is quite intuitive: the name f is temporarily used for (*bound to* is the correct term) a function which takes a number x and returns x^5; this name occours four times in the following code (said the *body* of the `let` expression), as a placeholder for the function $(\lambda\ x\ .\ x \cdot x \cdot x \cdot x \cdot x)$. Out of the `let` expression, this association of a value to the name f (this *binding* of f) is not visible.

If you *expand* the body replacing every occurrence of f with the value which is bound to it, you obtain an equivalent expression: in fact the whole query above has exactly the same meaning of

[6] Some dynamically-typed languages such as Lisp *do* permit having different types in the `then` and `else` branches; however this lack of compile-time checking makes programming errors very frequent. It's rare to actually need such a feature, and when it is really needed it can be easily simulated in epsilon via *concrete types*. Concrete types will be explained later in this book.

```
((\ x . x * x * x * x * x) 2) +
((\ x . x * x * x * x * x) 3) +
((\ x . x * x * x * x * x) 4) +
((\ x . x * x * x * x * x) 5);
```

This expansion shows how much `let` can make programs more readable.

As a side note, writing `a * a * ... * a` (with a occourring b times) is not the most clever way to compute a^b. The right way is using the *power operator* '`**`', which allows to simply write `a ** b`. We did not do so above just because using '`**`' was not convenient for us to illustrate the `let` construct: we needed some more "visual clutter".

To do: syntax (single binding), intuitive semantics, more examples, multiple binding

3.7.1 Digression: free occurrences and bound occurrences

To do: move this part to the beginning of this part, with a more precise explaination substitutions and reductions.

A *free occurrence* of a variable is an occurrence of the variable which does not refer to an inner λ or `let`. An occurrence which is not free is called *bound*.

For example, if we replace the free occurrences of `x` with `100` in

```
x + ((\ x . x + 1) (x + let x be 1 in (x + y)))
```

we obtain

```
100 + ((\ x . x + 1) (100 + let x be 1 in (x + y)))
```

Explaination:

- The first `x` is free.
- The `x` in `x + 1` is bound by the inner '`\ x .`'.
- The `x` in `x + let ...` is free.
- The `x` in `(x + y)` is bound by the inner '`let x be`'.

Free occurrences were briefly introduced here since they will be needed once in the next subsection. This same topic will be covered at length later in this book.

3.7.2 A more formal explaination

A first approximation[7] of the syntax of a `let` expression is

`let` $variable_1$ `be` $expression_1$ `in` $expression_2$

Here is the intuitive semantics: the subexpression $expression_2$ typically contains one or more occurrences of $variable_1$, even if this is not required; every free occurrence of $variable_1$ in $expression_2$ is replaced by the value which $expression_1$ reduces to, and the whole `let` expression reduces to the modified $expression_2$.

More formally:

- If $expression_1\uparrow$ then (`let` $variable_1$ `be` $expression_1$ `in` $expression_2$)$\uparrow$.
- Else $expression_1\downarrow$, and $expression_1 \Rightarrow y$; (`let` $variable_1$ `be` $expression_1$ `in` $expression_2$) $\Rightarrow expression_3$, where $expression_3$ is obtained from $expression_2$ replacing every free occurrence of $variable_1$ with y.

An example: let's show the evaluation of `let x be 1 + 2 in x + x - 1`.

- First see what $expression_1$ (here `1 + 2`) reduces to: `1 + 2` $\Rightarrow$ `3`. Ok, $expression_1\downarrow$.

[7] This first description does not cover the "multiple `let`", so it is not the complete syntax. It will be explained later.

- Replace the free occurrences of $variable_1$ (here `x`) in $expression_2$ (here `x + x - 1`) with the value we have just computed (here `3`): `3 + 3 - 1`.
- The whole `let` expression reduces to what we have just computed: `let x be 1 + 2 in x + x - 1` $\Rightarrow$ `3 + 3 - 1`
- Reduce again until possible: `3 + 3 - 1` $\Rightarrow$ `6 - 1` $\Rightarrow$ `5`

To do: computability: `let` is not needed for Turing-completeness

3.8 Recursive functions

We are now going to define the factorial function with the epsilon interpreter. The definition, as we already said in Chapter 2 [Functional programming tutorial], page 11, is

λ n . **if** n = 0 **then** 1 **else** n $\cdot$ (*fact* (n - 1))

To do

Part II - Specification

In this part we give a complete and formal description of the epsilon language, of its library and tools.

Some notions of Semantics and Languages would help to understand the mathematical parts, but they are not essential.

This is essentially reference material: feel free to skim it at a first reading.

4 Language

4.1 Lexicon

4.1.1 Whitespace

4.1.2 Comments

4.1.3 Identifiers

4.1.4 Numbers

4.1.5 Characters and strings

4.2 Basic types and expressions

4.2.1 The void expression

4.2.2 Integer expressions

4.2.3 Floating-point expressions

4.2.4 Boolean expressions

4.2.5 Character expressions

4.2.6 String expressions

4.2.7 Promise expressions

4.3 Basic constructs

4.3.1 The `declare` declaring construct

4.3.2 The `define` naming construct

4.3.3 The `if..then..else` conditional construct

4.3.4 The `let` block construct

4.4 Functions

4.4.1 lambda-notation

4.4.2 Functions with more than one argument

4.4.3 Recursion

4.4.3.1 The `fix` fixpoint operator

4.4.3.2 Mutually-recursive functions

4.5 Higher-order types

4.5.1 Lists

4.5.2 Arrays

4.5.3 Concrete types

4.6 Type inference

4.6.1 Type declarations in function parameters

4.7 Exceptions

4.8 Modules

4.8.1 Abstract types and synonym types

4.9 Polymorphism

4.10 Classes

4.11 Monads

4.12 The epsilon meta-interpreter

4.13 Foreign languages interface

4.13.1 Calling C from epsilon

4.13.2 Calling epsilon from C

5 Library

5.1 Default prelude

5.2 Containers

5.3 Meta-interpreter and meta-compiler

5.4 I/O

5.4.1 Terminal I/O

5.4.2 Curses I/O

5.4.3 Filesystem I/O

5.4.4 Sockets I/O

5.4.5 Graphics I/O with OpenGL

5.4.6 CORBA bindings

5.4.7 Bonobo bindings

6 Tools

6.1 Common command line behavior

The programs in the epsilon distribution, like all the other GNU programs, accept the following two options:

'`--help`'
'`-?`' Writes short usage information to the standard output and exits with success without executing.

'`--version`'
'`-V`' Writes program version and short license information to the standard output, and exits with success without executing.

Every option in the long form '`--XXXX`' has also the negative form '`--no-XXXX`', which does the opposite.

6.2 The `epsilon` interpreter

6.3 The `epsilonc` compiler

To do: non-option parameters

The `epsilonc` compiler translates epsilon modules source files into eAML. By default it also calls the `eamlas` assembler, the `eamld` linker, the `eamx2c` C code generator, and finally the system C compiler to generate a full native executable program.

If any of the intermediate pass fails the compiler writes an error message to the standard error and exits with failure.

The default behavior is all that is needed in simple cases; for more complex program it's conventient to direct `epsilonc` via command-line options to stop after any stage of compilation, allowing the user to manually call the other translators.

The program accepts the options described below.

'`--verbose`'
'`-v`' Makes the program verbose while executing; in particular it makes the program output the current compilation phase which is going to be executed to the standard error.

'`--show-types`'
'`-t`' Makes the program write the inferred types of all the objects that are defined, and the definitions of user-defined types. It's a useful feedback allowing the programmer to check for correctness.

This option is on by default.

'`--unescaped-string`'
'`-s`' Makes the generated program program write the strings and characters contained in the result of the evaluation of the main expression without quoting and without escapes. For example, if the main expression were `"ab\ncd"` the generated program would print '`ab`', a newline character and '`cd`', without surrounding double quotes.

'`--generate-eaml`'
'`-S`' Stops before the assembling phase.

'`--generate-eamo`'
'`-c`' Stops before the linking phase.

'`--generate-eamx`'
'`-x`' Stops before the C code generation phase.

'`--generate-eama`'
'`-a`' Generates a library containing the compiled modules instead of making an eAM executable file, then stop.

'`--generate-c`'
'`-C`' Stops before the native code generation phase executed by the system C compiler.

'`--generate-scheme`'
Generates Scheme code instead of C code from the eAM executable file, then stop; Scheme code does not normally need to be compiled.

'`--cc-options=`*XXXX*'
Passes the additional options *XXXX* to the C compiler when compiling the generated C code.

'`--main=`*XXXX*'
'`-m` *XXXX*' Specifies the name of the main module. Note that this is a *module* name, and not a file name, so it can not have extension.

'`--output=`*XXXX*'
'`-o` *XXXX*' Specifies the name of the output file, i.e. the one generated in the last phase; the extension must be in accord with the type of the file generated.

6.4 The `eamlas` assembler

6.5 The `eamld` linker

6.6 The `eamx2c` eAM executable to C compiler

6.7 The `eamx2scheme` eAM executable to Scheme compiler

6.8 The `epsilonlex` scanner generator

6.9 The `epsilonyacc` parser generator

Part III - Internals

This part contains a detailed description of the implementation of epsilon.

It is of no particular utility for simple *users* of epsilon, except for who wants to have a deep feeling of how things work "below". It's very important, instead, for programmers who want to modify epsilon to extend it or to re-use a part of its code for some other purpose[1].

The chapters of this part have somewhat stronger requisites than the rest of the book: proficiency with compiler and run-time support design, C, flex, Bison, bash and Scheme is required to fully understand the sources. Also some notions of operating systems would help.

[1] Remember that epsilon is free software ("free" in the sense of "free speech": see `http://www.gnu.org`) covered by the *GNU General Public License*. See [Copying], page 1 for the full text.

7 Internals overview

This chapter describes the *implementation* of epsilon, how it was written the way it is and why.

7.1 Architecture

As any nontrivial software project, epsilon is structured in *layers.*

To do: talk about the REPL

At the top level there is the *compiler*, translating a module written in epsilon into a lower-level form. This form is called *eAML*, for "epsilon Abstract Machine Language". The compiler outputs a textual form of the eAML language, relatively easy to read for humans. This helps the developer of epsilon to test and debug the compiler, and also allows to write code directly in eAML. The eAML language is quite low-level and resembles an assembly language. It is an **imperative** language, making explicit the control flow, the ordering of the computations and the environment management.

Under the compiler there is the second component, the *assembler*, translating an eAML module from the textual form into a binary form called *bytecode object file*, so that the computer can deal with eAML in a more efficient way. Instructions are translated one by one with no important modifications. It's simply a change of format.

The *linker* takes several bytecode object files (normally each one derives form an epsilon source module) and links them into a single larger bytecode object file; this is necessary since the lower part of the system can only deal with *one* bytecode object at the time. The linker can also read or write a *bytecode archive file*, which is a library of bytecode object files, tipically with some external references not yet resolved.

The bottom part of the system, called the *epsilon Abstract Machine* or *eAM*, supports execution via one more tranlation pass: the *bytecode-to-C translator* compiles a bytecode object file into a C source program, to be compiled by an optimizing C compiler into native machine code, and finally executed. The drawback of this approach is the delay due to the compilation of optimized C code (the delay of bytecode-to-C translation is negligible), but the runtime speed of the generated code should be very high.

7.1.1 The `epsilonc` compiler

To do: I need to rewrite the compiler in epsilon, and to document it.

7.1.2 The `eamlas` assembler

The `eamlas` assembler is a simple single-pass translator with backpatching, written in C with flex and Bison.

A few words about source file organization before starting: the assembler source files are in the `eam/` subdirectory. For each category of instructions with the same format (e.g. with an integer parameters, or with no parameters, or with a label parameter) there is a subdirectory under `eam/c_instructions/`, containing the code for the instructions, one instruction per file. For example, the code for `s_nlcl` instruction, having two integer parameters, is in the file `eam/c_instructions/integer_integer/s_nlcl`.

To ease the developing of epsilon and to provide a better structuring, the source files `eamlas.l` and `eamlas.y` are not written by hand; instead they are automatically generated by the bash scripts `make_eamlas_l` and `make_eamlas_y`. The scripts scan the `instructions/` subdirectories to find the opcodes of all the instructions and to divide them into categories after the format of parameters. The division into categories is useful while generating the frontend files `eamlas.l` and `eamlas.y`. The advantage of this approach is evident: to add, remove or rename an instruction it is sufficient to work only on the **single** file for that instruction: the assembler (together with the other low-level parts of the system) is updated automatically.

The output of the bytecode in written into the file using the module `bytecode.c` (also used by other parts of the system).

Other than `bytecode.c`, nothing more than the generated `eamlas.l` and `eamlas.y` is needed; the main logic is in `eamlas.y`: just a single scan in which all the found instructions are memoized, and all labels uses and definitions are stored in a data structure (essentially an hash table). At the end of the parsing all label references are resolved with a backpatch, and the output file is finally written.

7.1.3 The eAM abstract machine

The epsilon Abstract Machine is relatively complex, and deserves a whole chapter. See Chapter 8 [The epsilon Abstract Machine], page 45.

7.1.4 The `eamold` linker

To do: write the linker and document it

7.1.5 The `eamo2c` bytecode-to-C translator

To do: document eamo2c internals

7.1.6 The `epsilonlex` scanner generator

7.1.7 The `epsilonyacc` parser generator

7.2 Extending the eAM

To do: talk about how to create new eAM instructions.

7.3 File formats

7.3.1 eAML file format

7.3.2 bytecode object file format

7.3.3 bytecode archive file format

8 The epsilon Abstract Machine

The *epsilon Abstract Machine*, or *eAM*, is a model of the operations involved in the execution of epsilon programs.

The eAM is *imperative* and, at the time of this writing, *sequential*; many functional properties of the epsilon language such as referential transparency and indipendency from evaluation order are lost in the translation from epsilon to eAML.

The eAM is relatively low-level, based as it is on a stack, a heap and an array of registers. The garbage collector is run automatically, even if it can be tuned with some special instructions.

It is worth repeating that the eAM is an *abstract* machine, and not a virtual machine. This is to say that there is **not** necessarily a step-by-step interpretation of bytecode instructions. The eAM model is only an *abstraction* of the functionality which is available at this level; in the implementation eAM instructions are translated into C and then compiled into native code with optimizations, or into Scheme code and then passed to Guile. However, for ease of implementation and for better undertanding, it's also useful to think of the eAM as a proper machine with its registers, stack, heap and instructions. Just remember that this does not mirror the execution model.

8.1 eAM types

Any datum used by the eAM is of exactly one of these types:

integer
: Integer objects hold a limited range of integer values, roughly balanced around 0; their width is guaranteed to be at least 32-bit.

wide integer
: Wide integer objects also hold a limited range of integer values. They are guaranteed to be at least as wide as integers, and possibly wider.

float
: Float objects hold floating-point numbers. Their width is guaranteed to exactly match the width of integer objects.

wide float
: Wide float objects hold floating-point numbers at least as wide as and possibly wider than float objects.

wide wide float
: Wide wide float objects hold floating-point numbers at least as wide as and possibly wider than wide floats.

pointer
: A pointer object holds holds the memory address of another eAM datum, of itself, or of nothing; in this latest case a pointer is called a "*null* pointer". Null pointers are all seen as identical upon comparation, as `NULL`s in C. *Internal* pointers are forbidden: pointers are **not** allowed to refer to memory addresses inside a word or inside an array. Pointers are guaranteed to be exactly as wide as integers.

array
: Arrays are random-access ordered collections of word objects (see below); elements can have heterogeneous types. Arrays only hold their elements: no information about size is implicitly included. However it is common to store length information in the first element, when needed; some instructions are provided to make this fast and convenient. eAM arrays are always indicized starting from 0.

The internal representation of floats, wide floats and wide wide floats should follow the IEEE 754 Standard on modern architectures.

Note that no booleans, characters or strings are provided. Objects which in epsilon have these types, or higher-order types (such as epsilon functions, lists or tuples) are implemented using only the above eAM types.

The *integer*, *float* and *pointer* types are collectlively known as *word* types. The reason is that, in all reasonable architectures[1], any word object exactly fits in a physical machine general register. By contrast *array*, *wide integer*, *wide float* and *wide wide float* objects may be larger than a physical word. For this reason *word* objects are typically faster to manipulate.

No objects smaller than a word are provided.

Note that **no** run-time type tagging exists[2]: it's up to the compiler which generates eAML code to check for type errors at compile time, or to arrange runtime checks generating appropriate instructions when needed. When type tagging is needed on an object a (for simplicity you can think of a C `union`; all other cases can be reconducted to this one), it can be realized implementing a as an array whose (say) first element is an integer value discriminating between all possible types that a can assume at run-time; the second (and third, fourth and so on if needed) element holds the proper datum.

8.2 Memory model

References are managed via pointers, directly from runtime-support structures such as the stack or the registers, or from elements of array type.

Data structures can be realized with arrays containing pointers (and other word objects, if needed). Cyclic data structures are allowed without restrictions.

Storage allocation is realized with explicit eAM instructions, but storage reclamation is automatically managed by the garbage collector.

8.3 Representation of epsilon data in the eAM

Every epsilon object has an underlying representation in the eAM; epsilon objects of most basic types are quite easily mapped to eAM objects of word types; for higher-order objects the mapping is more complex.

The eAM deals with non-word objects using pointers, which are word objects: for example a list is represented with the usual pointer-based data structure, and the whole list is referred using a pointer to the first cons (or a null pointer if the list is empty); to summarize, *an epsilon datum which does not fit in an eAM word object is represented with non-word objects, and a pointer (word) referring "the first element"*, whatever we mean as "the first element". Note that internal pointers are forbidden in the eAM, so the first element must be a whole eAM objects (which can be an array).

We are now going to describe the mapping in its details.

8.3.1 `integer`, `character` and `boolean`

The epsilon types `integer`, `character`, and `boolean` are easily mapped into eAM *integer*s.

Note that also a character uses a full word; this enables to use modern encodings such as Unicode (even if such support is not yet implemented). GNU epsilon is a new language, and we deliberately chose not to be restricted by obsolete 8-bit encodings such as ASCII or Latin1.

---

[1] These days, and in the foreseeable future, physical processors are 32-bit or 64-bit (the GNU Project does not support 16-bit machines, since they are long obsolete). 32-bit processors should have 32-bit pointers, and 32-bit general registers. 64-bit machines should have 64-bit pointers, and they should be able to do computations with 64-bit integers at assembly level. If they aren't, it's hoped that at least the C compiler provides support for 64-bit integer operations. We don't know of any counterexample; please write us to `bug-epsilon@gnu.org` if you know some, specifiyng.

[2] opposing to Lisp and Smalltalk implementations, for example. The absence of type tags at runtime speeds up execution and reduces memory usage.

The epsilon boolean `false` is represented as the eAM integer `0`. `true` is represented by any non-zero eAM integer.

When held in the stack or in a register these objects are copied rather than referred by a pointer. The rationale behind this is that it would be a waste of time and memory to hold pointers to immutable objects (remember that epsilon is a functional language), when the pointer has the same cost as the whole object.

8.3.2 `float`

The epsilon type `float` is trivially mapped into the eAM type *float*. A float object fits in a machine word.

Floats can be directly held in the stack or in a register: there are no pointers to float objects. The rationale is the same as the one for the case above.

8.3.3 Tuples

epsilon tuples are mapped into eAM *array*s holding the representation of each element; the order of the elements in the eAM representation always reflects the order of the elements in epsilon.

No information about the length of the tuple is held at runtime, since if the program was correctly compiled *no* bound-checks are ever needed at runtime for tuples.

When held in a register or in the stack the tuple is always referred by a pointer to the eAM array which represents it.

8.3.4 Arrays

epsilon `array`s of size n are mapped into eAM *array*s of size $n + 1$, where the first element is an integer holding the size of the array, and the following elements are the representation of the actual elements.

Holding the information about the lenght at runtime enables the eAM to make bound checks.

Empty arrays follow the general rule: they are represented as eAM arrays of size 1, where the only element is an integer with value 0.

When held in a register or in the stack the array is always referred by a pointer to the eAM array which represents it.

8.3.5 Strings

epsilon `string`s are representated just as if they were arrays of characters.

Note that this representation allows computing the size of a string in $O(1)$.

8.3.6 Lists

epsilon `list`s are represented with the usual pointer-structure: each cons holds a pointer to the next one.

Each cons is represented as an eAM array of size 2, where the first element (the `head`, or *car*) holds the representation of an actual list element, and the second element (the `tail`, or *cdr*) holds a pointer to the rest of the list, or a null pointer if the rest of the list is `[]` (*nil*).

The empty list `[]` has no representation.

When held in a register or in the stack the list is always referred by a pointer to the eAM array which represents its first cons, or by a null pointer if the list is empty.

8.3.7 Objects of concrete types

To do

8.3.8 Objects with behavior

To do

8.3.8.1 Promises

8.3.8.2 Functions

8.3.8.3 Actions

8.3.9 Objects of abstract types

The objects of an abstract type actually have another type (said the *implementation type*), which is usually hidden in the module which defines operations on them.

epsilon objects of abstract types are represented as objects of their implementation type; abstract types have **no penality** on representation. They are essentially gratis.

8.3.9.1 Examples

We are now showing the representation of some epsilon objects as eAM objects. We describe what appears in a register holding each of the sample epsilon objects:

- `12`

  The eAM integer `12`.
- `1.323`

  The eAM float `1.323`.
- `(1, 0.4)`

  A pointer to an eAM array of two elements: the first element holds the eAM integer `1`, the second one holds the eAM float `0.4`. Note that the size of the tuple (2) is not needed at runtime, so it is not explicitly stored anywhere.
- `<| 1, 0.4 |>`

  A pointer to an eAM array of three elements: the first element holds the eAM integer `2`, which represents the size of the array; the following elements are the eAM objects `1` and `0.4`.

  Note how in the eAM the proper elements of the array are indicized starting at 1: the zero-th element holds the size, which can be extracted very efficently (one or two assembler instructions in most processors).
- `<| |>`

  A pointer to an eAM array of one element holding the eAM integer `0`.

  It's worth repeating that epsilon empty arrays are **not** represented as eAM null pointers. This convention saves some nullity tests at runtime and makes the representation more uniform.

- `"Abc"`

  A pointer to an eAM array of four elements: the first element holds the eAM integer `3`, which represents the size of the string. The following elements are eAM integers holding the integer representation of the characters `'A'`, `'b'` and `'c'`.
- `[ 1; 2 ]`, which is an abbreviation of `(1 :: 2 :: [])`

  A pointer to an eAM array *c1* of two elements: the first element of *c1* holds the eAM integer `1`. The second element of *c1* holds a pointer to the eAM array of two elements *c2*.

  The first element of *c2* holds the eAM integer `2`. The second element of *c2* holds a null pointer, i.e. a "reference" to the representation of `[]`.

  Note that the elements of the list are integers in this example. If they were something more complex, for example strings, the first elements of *c1* and *c2* would be *pointers*.

To do: examples of objects with behaviour.

8.4 Runtime support structures

Most non-word objects are stored in a garbage-collected *heap*. The management of the heap is entirely transparent even at the eAM level: many instructions allocate a datum on the heap and return a pointer to it, storing the pointer on the *stack* or in a *register*. The stack and the registers are provided to hold temporary data.

The stack is a simple LIFO container of word objects, divided into *frames*. Each frame represents an activation of a subprogram or a block, and is essential especially to implement recursion. Many eAM instructions work on the stack, taking operands from it or using it to return a computed value. Other instructions push or pop entire frames on the stack: they are needed to enter or exit a block, and to implement subprogram calls. The stack is not limited in size and can *not* overflow.

Some eAM instructions use the registers instead of the stack to make computations. Registers are faster to manipulate than the stack, but are provided in a limited number and are not sufficient by themselves to handle recursion.

Registers are created at initialization time; their number is defined by the program and cannot grow at runtime. Registers are divided in several groups, according to the data they can hold:

word registers
: A word register, also called *general register*, can contain an object of any word type; this is very useful to implement polymorphic procedures, whose semantic does not depend on the type of the argument.

  Word registers are also the only ones used by eAM instructions operating with registers to do compuations on integers and pointers. Computations[3] with floats can not be done in word registers, even if word registers can hold float values (since they are word-sized).

wide integer registers
: A wide integer register can only contain a wide integer object.

float registers
: Float registers are the only ones used by the eAM instructions operating with registers to do computations on floats. Computations with integers and pointers can not be done in float registers.

[3] The idea of "computation" does not include copying: any word-sized value can be passed in a word register and hence blindly copied into the stack, the heap or another register. This *generic* operation does not depend on the type of the operand but only on its size, and it is reasonable to allow it in general registers.

wide float registers
: A wide float register can only contain a wide float object.

wide wide float registers
: A wide wide float register can only contain a wide wide float object.

A similar distinction does not exist for the stack: there is only *one* stack, and it is limited to word objects (including floats): you can not directly push a non-word object onto the stack: you can only push a *pointer* to it.

This is an eAM program example using the stack to evaluate the expression (2 + 5) - 1:

```
pshci   2       # Push 2
pshci   5       # Push 5
s_addi          # Pop two values, sum them and push the result
# The value of (2 + 5) is on the top of the stack
pshci   1       # Push 1
s_subi          # Subtract top from undertop, pop both and push
                # the result of the subtraction
# The result is on the top of the stack
```

This program also evaluates the same expression, but uses registers instead of the stack:

```
ldci    $g1 2       # $g1 := 2
ldci    $g2 2       # $g2 := 5
addi    $g3 $g1 $g2 # $g3 := $g1 + $g2
# Now $g3 holds (2 + 5), i.e. 7
ldci    $g4 1       # $g4 := 1
subi    $g5 $g3 $g4 # $g5 := $g3 - $g4
# Now $g5 holds the result
```

The previous examples are meant to illustrate the two possible styles of evaluation, and nothing more. Both could be heavily optimized.

8.5 Subroutines and blocks

To do: talk about frame pointer, stack pointer, frame format

8.5.1 Calling conventions with operands in the stack

To do: document To do: an example

8.5.2 Calling conventions with operands in the registers

To do: design, implement and document To do: an example

8.6 The eAM garbage collector

The eAM includes a pseudo-generational mark and sweep collector with conservative pointer finding, not incremental at the time of this writing.

The implementation is relatively simple; it can be roughly divided into two parts: the *allocator* and the *collector*.

Objects are allocated from buffers called *pages*. There are two sorts of pages: *homogeneous* pages and *large* pages.

8.6.1 Homogeneous pages

A homogeneous page contains objects (called *homogeneous objects*) of the same size k (in words); all homogeneous pages have exactly the same size S (tunable via a C macro `#define`); hence the number of objects in a single homogenous page depends on k: pages with a smaller k contain more objects, and pages with a larger k contain fewer objects.

In each homogeneous page non-allocated objects are linked via a simple unidirectional free-list. For each object there is an associated *allocated bit*, also stored in the same page, which is set to 1 if and only if the object is allocated. Each page finally contains a field holding the number of its allocated objects.

These simple data structures allow to perform the following operations with time complexity $O(1)$:

- check whether an object is allocated in a given page
- allocate an object from a given page
- free an object, returning it to its page
- check whether a page is empty
- check whether a page is full

Each large page also holds a *GC bit* for each object, used by the collector.

All homogeneous pages are allocated with alignment S using `memalign()`; this allows to find the page of an homogeneous object with a bit-masking of its address, a very fast operation. In the implementation the value of S is computed from the value of the C macro `PAGE_OFFSET_WIDTH` defined in '`eam/gc/heap.h`'.

A pointer to each homogeneous page is stored in the hash table `set_of_homogeneous_pages` (defined in '`eam/gc/homogeneous.c`'). This makes possible to check whether an object is homogeneous with complexity $O(1)$ in the average case (one bit-masking plus one hash table access).

Homogeneous pages with various values of k are created during initialization, but of course not covering every possibile size. So, if an object of a given size is asked, the allocator could return a slightly larger object: in this case we speak about *inexact allocation* (in the other case we speak about *exact allocation*). The little waste of space implied by inexact allocation seems not to be a problem in practice.

There is an array indicized by any given possible k from 0 to the maximum allowed value `MAXIMUM_HOMOGENEOUS_SIZE`, `homogeneous_pages`, defined in '`eam/gc/gc.c`'; among the rest it contains, for each size, its best approximation. So also inexact allocation has complexity $O(1)$ when a non-full page of the right size exists.

The other fields of `homogeneous_pages` are, for each k, the bidirectional list of non-full homogeneous pages for objects of size k and the bidirectional list of full homogeneous pages for objects of size k. The list structures make easy adding or removing homogeneous pages as needed.

A final note: even if the GNU system allows to free a block allocated with `memalign()` there is no portable way to do it; so homogeneous page are actually destroyed only on GNU systems[4]; on the other systems they are created and kept allocated forever, in the hope that they will be needed again.

8.6.2 Large pages

Sometimes objects larger than `MAXIMUM_HOMOGENEOUS_SIZE` words, or even larger than S words, are needed. Some other structure is needed, since those objects can not fit in homogeneous pages some other structure is needed.

[4] The type of system is automatically determined at configure time.

Managing a *large page* is simpler than managing a homogeneous page: a large page holds one and only one *large object*: large pages are created when allocating a large object, and destroyed when a large object is freed: there is no need to keep free-lists or allocated bits. Moreover freeing a large object immediately releases storage; this can be an advantage when *really* large sizes are involved.

Each large page also holds the GC bit[5] for its object.

Large pages do not need a specific alignment: they are simply allocated with `malloc()` and freed with `free()`.

A pointer to each homogeneous object is kept in the hash table `set_of_large_objects`, defined in '`eam/gc/large.c`'. This enables to check whether an object is large with complexity $O(1)$ in the average case (one hash table access).

All large pages are linked in the bidirectional list `list_of_large_pages`, defined in '`eam/gc/gc.c`'. Note that *all* large pages are full, since when they become empty they are immediately destroyed, and there is no intermediate condition: large pages can only be full or empty.

A large page is just a thin shell enclosing its large object and the little bookkeeping information needed.

8.6.3 Allocator

After the initialization performed by `void initialize_garbage_collector()`, declared in '`eam/gc/gc.h`', all data structures are set up and a homogeneous page is created for the values of k belonging to a certain predefined set Q[6]. No large pages are created at initialization time. They are created at runtime, just when needed.

The interface of the allocator is very simple; all the needed functions are declared in the header '`eam/gc/gc.h`'.

Exact allocation is performed by `word_t allocate_exact(integer_t words_no)`.

`allocate_exact()` works by allocating an object from the first page of the list of non-full homogeneous pages at `homogeneous_pages[`*`words_no`*`]`; the list is always kept non-empty. When the page gets full it is moved from the list of non-full pages to the list of full pages at `homogeneous_pages[`*`words_no`*`]`; if the list of non-full pages becomes empty then a new page is created.

Inexact allocation is performed by `word_t allocate_inexact(integer_t desired_words_no)`. If `desired_words_no` is not greater than `MAXIMUM_HOMOGENEOUS_SIZE` then `allocate_inexact()` computes the best approximated size by simply looking at the field `inexact_size` of `homogeneous_pages[`*`desired_words_no`*`]`, then calls `allocate_exact()`; else it creates a new large page and returns its object.

Note that allocation of large objects is *always* considered inexact.

Exact allocation is slightly faster than inexact allocation, but can be used only when the requested size k belongs to Q; if the requested size does not belong to Q *the behaviour is undefined*, which is a nice way to say that the program will most probably **crash**, and the collector will surely **not** work.

---

[5] The "bit" is effectively implemented with a word. In C it makes sense to use a bit *vector*, but a bit vector of just one element effectively takes more space than a bit.

[6] The current algorithm allocates pages for $S = 1, 2, 3, \ldots$ `MAXIMUM_SMALL_HOMOGENEOUS_SIZE` and $S=$ $2\cdot$`MAXIMUM_SMALL_HOMOGENEOUS_SIZE`, $4\cdot$`MAXIMUM_SMALL_HOMOGENEOUS_SIZE`, $8\cdot$`MAXIMUM_SMALL_HOMOGENEOUS_SIZE`, ... `MAXIMUM_HOMOGENEOUS_SIZE`.

`MAXIMUM_SMALL_HOMOGENEOUS_SIZE` and `MAXIMUM_HOMOGENEOUS_SIZE` are defined in '`eam/gc/gc.h`'.

8.6.4 Collector

The header '`eam/gc/gc.h`' also contains the interface to the collector.

`void initialize_garbage_collector()` transparently starts a new concurrent thread which from time to time[7] checks whether a collection would be needed, and in the affermative case sets the flag `int should_we_collect` to a nonzero value.

The mutator has the responsibility to periodically[8] check the flag, and request a collection if the flag is nonzero. Note that there is no need of synchronization here: one thread reads the flag but does not write it, the other one writes it but does not read it.

For each collection cycle the mutator must explicitly notify the collector about all roots, calling `void add_gc_root(word_t p)` or, when there is more then one root in a single buffer, `void add_gc_roots(word_t* buffer, size_t words_no)`.

The roots of the eAM are:

- All the elements of the stack
- Word registers
- The I/O register
- Globals
- The environment register
- `exception_value`
- `exceptions_stack[i].environment`, for each element i of the exception stack

String constants are *not* roots: there is no need to keep them in the garbage-collected heap, so they are simply allocated with `malloc()` at startup time. This saves a little time when marking.

After notifying the collector about the roots a call to `void garbage_collect()` performs marking and sweeping[9].

To do: more details? Implementation is quite "conventional" here...

8.6.4.1 Pseudo-generational garbage collection

It would be slow to perform a full mark at every garbage collection cycle, so the eAM collector implements a *pseudo-generational* marking algorithm. What we call "pseudo-generational" garbage collection is a particular case of the *generational garbage collection*, particularly simple but quite effective. The heap is conceptually partitioned into two generations:

- the *old generation* contains all the object which survived the last garbage collection cycle;
- the *young generation* contains the objects which were allocated after the last collection.

Minor collections, performed relatively often, only scan the young generation, making the marking phase noticeably faster; note that marking time usually dominates over sweeping time.

Major collections, performed less often[10], scan both the young and the old generation. Major collection are slower than minor collection but free more storage.

[7] In the current implementation the concurrent thread wakes up every `GC_TEST_TIMEOUT` nanoseconds.

[8] In the current implementation the mutator checks the flag right after each application of a *recursive* function.

[9] In the current implementation `garbage_collect()` suspends the mutator.

[10] The current implementation uses a rough heuristic: one collection every `MINOR_GC_CYCLES_NO` minor collections is major (`MINOR_GC_CYCLES_NO` is defined in the header '`eam/gc/gc.h`'). This will be improved.

The main idea of pseudo-generational collection is that *the GC bits are cleared only at the beginning of major collections*: in minor collactions the old generation objects are seen as already marked, so they are not recursively[11] scanned.

The mutator has no direct control over the generations. It can only request a collection, and it's the function `garbage_collect()` to decide whether a minor or major collection is needed.

8.6.5 Safe points

To do

8.7 eAM instructions

The instructions of the eAM are described in full detail in Chapter 9 [eAM instructions], page 55.

[11] However old generation objects can be scanned from the roots. This is rarely a problem: roots should have not a very large size (at least if stack usage is not high, and high stack usage usually indicates subotimal use of tail-recursion).

9 eAM instructions

This chapter describes in detail all the instruction of the epsilon Abstract Machine. Familiarity with the epsilon memory model and runtime structures is assumed.

This chapter is of no particular use for writing programs in epsilon. It is useful, instead, to understand how the internals of epsilon work and especially to *extend* the language or the runtime system.

9.1 Naming conventions

Each eAM instruction is identified by a unique *mnemonic* in the textual form of eAML. We are going to introduce the rules which were used to choose the mnemonics names to make them more consistent and easier to remember.

By convention, when an instruction works on operands with fixed type, a *suffix* of the mnemonic identifies that type: `i` for integers, `f` for floats and `s` for strings. For example the `s_addi` instruction executes an addition on integer operands.

When more than one instruction exists doing the same work, but with a version taking a parameter from a register, another taking a parameter from the stack and still another taking one or more immediate parameters, the versions are easily recognizable from the *prefix* or *desinence* in their mnemonic:

- The version with immediate parameters has the `_i` desinence in its mnemonic.
- The version with parameters from the stack has the `s_` prefix in its mnemonic.
- The version with parameters in the registers has no prefix or suffix.

For example, `s_addi` adds two integer taken from the stack, and `s_addi_i` adds an integer taken from the stack to the immediate integer which is the parameter of the instruction.

Some instructions are provided in two versions, one *safe* (i.e. making runtime checks) but less fast, the other *faster* but not safe. "Fast" versions are identified by a `f_` prefix in their mnemonic. If the `s_` prefix is also present, `s_` preceeds `f_` in the mnemonic, such as in `s_f_divi`.

In the following immediate integer parameters are indicated by n, m, x, y or z; strings are indicated by S, and labels by L:. Register parameters are indicated by a dollar-sign (`$`) followed by an letter, such as *$a* or *$x*.

9.2 Writing conventions for stack and registers configurations

In this chapter we also describe instructions updating the stack or the registers: for brevity we follow these writing conventions:

- Stack configurations are written horizontally, with the bottom of the stack at the left side. The bottom is identified by a "`||`" symbol, the top by a "`|`" symbol, and elements are separated by a "`|`" symbol. For example `||`*a*`|`*b*`|` is a two-elements stack containing the *a* object at the bottom, and the *b* object on the top. The "..." symbol stands for "some elements whose value is not specified", except when found in sequences like "`|`*a1*`|`...`|`*an*`|`", where the dots stand for exactly n-2 elements which have as values the elements of the succession *a* from 2 to n-1.
- Register configurations are written horizontally as semicolon-separated sequences surrounded by backets, such as [-1;`"a"`;3.7]. The leftmost element represents the content of the `$0` register, the second one of `$1`, and so on. For the ...-notation we follow the same conventions as in writing the stacks.

- For illustrating the effect of executing an instruction we show the configuration of stack, registers or anything relevant *before* the execution, the instruction name with parameters, and finally the configuration *after* the execution. When the ...-notation is used in both the before and after configurations, the unexpressed elements are considered to be the same, if not otherwise noted.

  For example:

  | | ... | *a* |

  `s_addi_i` *n*

  | | ... | *a*+*n* |

  For brevity's sake register updates can also be noted as

  $\$a := EXP(\$b, ..., \$z)$

  where $\$a$ is the updated register, and $EXP(\$b, ..., \$z)$ is any expression involving registers $\$b$, ..., $\$z$. Note that in the expression at the right of the ":=" symbol "$\$x$" represents the *content* of the $\$x$ register, not its address.

9.3 Writing conventions for structures

If we stay at the level of the eAM, epsilon structures are not anything more than arrays (or tuples).

Arrays are shown in a visual way as sequences of objects separated by commas and surrounded by angular parentheses. Null pointers are written as "*null*". For example this

<1, <2, <3, <4, *null*>>>>

is the eAM representation of the epsilon list `[1; 2; 3; 4]`.

This, instead,

<<`"test"`, *null*>, *null*>

is the eAM representation of the epsilon object `(["test"], [])`, and also of the epsilon object `[["test"]]`.

9.4 Instructions classification

eAM instructions are divieded into several *categories*:

- *arithmetic/logic* instructions make computations of arithmetic or logic nature. Some instructions are available to work with integer values, others for working with float values.
- *conversion* instructions translate data from one representation to another.
- *structures management* instructions work with memory structures bigger than one word such as arrays and conses.
- *stack management* instructions update the stack, copying or removing elements from the top, or copying objects from the registers to the stack or from the stack to the registers.
- *flow control* instructions control the flow of the execution, with unconditional or conditional jumps.
- *subprograms management* instructions manage entering and exiting from subprograms and blocks.
- *variables management* instruction deal with program variables, be they locals, nonlocals or globals.
- *input/output* instructions do input and output, and allow interfacing with code written in other languages.

- *exception handling* instructions are used to signal error conditions and to manage failures.
- *special* instructions are the few instructions which do not fit in the above categories.

9.5 Arithmetic/logic instructions on integers

Arithmetic/logic instructions operating on integer values are essential since they are used even in the simplest programs. They do not involve memory management, and for this reason they are fast compared to other ones.

Some instructions taking two operands from the stack exist also in a version with the second operand as an immediate parameter (for example `s_addi` and `s_addi_i` *n*). The versions with immediate parameters are not always applicable, but faster.

In the following subsections we are going to describe every instruction in detail.

9.5.1 addi *$a* *$b* *$c*

The `addi` *$a* *$b* *$c* instruction adds the content of the *$b* register to the content of the *$c* register, storing the result into the *$a* register:

$a := *$b* + *$c*

9.5.2 addi_i *$a* *$b* *n*

The `addi_i` *$a* *$b* *n* instruction adds *n* to the content of the *$b* register, storing the result into the *$a* register:

$a := *$b* + *n*

9.5.3 andi *$a* *$b* *$c*

The `andi` *$a* *$b* *$c* instruction stores a nonzero value into *$a* if both *$b* and *$c* have nonzero value, else it stores zero into *$a*.

9.5.4 divi *$a* *$b* *$c*

The `divi` *$a* *$b* *$c* instruction divides the content of the *$b* register by the content of the *$c* register, storing the result into the *$a* register:

$a := *$b* / *$c*

If the content of the *$c* register is zero the execution terminates reporting an error.

9.5.5 divi_i *$a* *$b* *n*

The `divi_i` *$a* *$b* *n* instruction divides the content of the *$b* register by *n*, storing the result into the *$a* register:

$a := *$b* / *n*

No division-by-zero check is made, since it would make never sense to use this instruction with *n*=0.

9.5.6 `f_divi` *$a $b $c*

The `f_divi` *$a $b $c* instruction divides the content of the *$b* register by the content of the *$c* register, storing the result into the *$a* register:

$a := *$b* / *$c*

No divide-by-zero check is made, so the program may **crash** if the content of the *$c* register is zero. This instruction is faster than `divi` *$a $b $c*, but you should only use it when you are **definitively sure** that *$c* can not hold a zero value.

9.5.7 `f_modi` *$a $b $c*

The `f_divi` *$a $b $c* instruction divides the content of the *$b* register by the content of the *$c* register, storing the rest of the division into the *$a* register:

$a := *$b* mod *$c*

No divide-by-zero check is made, so the program may **crash** if the content of the *$c* register is zero. This instruction is faster than `divi` *$a $b $c*, but you should only use it when you are **definitively sure** that *$c* can not hold a zero value.

9.5.8 `ldci` *$r n*

The `ldci` *$r n* instruction updates the content of the *$r* register to the integer constant n:

$r := n

9.5.9 `modi` *$a $b $c*

The `modi` *$a $b $c* instruction divides the content of the *$b* register by the content of the *$c* register, storing the rest of the division into the *$a* register:

$a := *$b* mod *$c*

If the content of the *$c* register is zero the execution terminates reporting an error.

9.5.10 `modi_i` *$a $b n*

The `modi_i` *$a $b n* instruction divides the content of the *$b* register by n, storing the rest of the division into the *$a* register:

$a := *$b* mod n

No division-by-zero check is made, since it would make never sense to use this instruction with n=0.

9.5.11 `muli` *$a $b $c*

The `muli` *$a $b $c* instruction multiplies the content of the *$b* register for the content of the *$c* register, storing the result into the *$a* register:

$a := *$b* ·*$c*

9.5.12 `muli_i` *$a $b n*

The `mul_i` *$a $b n* instruction multiplies the content of the *$b* register for n, storing the result into the *$a* register:

$a := *$b* ·n

9.5.13 nxori *$a* *$b* *$c*

The `nxori` *$a* *$b* *$c* instruction stores a nonzero value into *$a* if either both *$b* and *$c* have nonzero content, or both *$b* and *$c* have zero content, else it stores zero into *$a*.

9.5.14 ori *$a* *$b* *$c*

The `ori` *$a* *$b* *$c* instruction stores a nonzero value into *$a* if at least one of *$b* and *$c* has nonzero content, else it stores zero into *$a*.

9.5.15 s_f_divi

This instruction is identical to `s_divi`, except that it does not check for the divison-by-zero error condition.

The program may **crash** if the top of the stack is 0; you should use this instruction only if you are **definitively sure** that the divisor is not zero.

`s_f_divi` is faster than `s_divi`.

9.5.16 s_addi

The `s_addi` instruction replaces the two integer objects on the top of the stack with a single object with their sum as value.

| | ... | *a* | *b* |

`s_addi`

| | ... | *a* + *b* |

9.5.17 s_addi_i *n*

The `s_addi_i` instruction replaces the integer object on the top of the stack with the sum of it and the *n* parameter.

| | ... | *a* |

`s_addi_i` *n*

| | ... | *a* + *n* |

`s_addi_i` *n* is faster than `pshci` *n* `; s_addi`.

9.5.18 s_andi

The `s_andi` instruction replaces the two integer objects *a* and *b* on the top of the stack with a single object with a nonzero value if *both* *a* and *b* have nonzero value, otherwise with a single object with zero value.

For example:

| | ... | -34 | 0 |

`s_andi`

| | ... | 0 |

9.5.19 s_divi

The s_divi instruction replaces the two integer objects on the top of the stack with a single object with their quotient as value.

In case of division by zero this instruction prints an error message and terminates the execution of the program.

||...|a|b|

s_divi

||...|a / b|

9.5.20 s_divi_i n

The s_divi_i instruction replaces the integer object on the top of the stack with the quotient of it and the n parameter.

The program may **crash** if n is 0; no divide-by-error check is done since it makes *never* sense to use s_divi_i 0.

||...|a|

s_divi_i n

||...|a / n|

s_divi_i n is faster than pshci n; s_divi, and even than pshci n; s_f_divi.

9.5.21 s_eqi

The s_eqi instruction replaces the two integer objects on the top of the stack with a single object with a nonzero value if the integer objects are equal, otherwise with zero.

For example:

||...|a|a|

s_eqi

||...|1|

9.5.22 s_gti

The s_gti instruction replaces the two integer objects a and b on the top of the stack with a single object with a nonzero value if a is greater than b, otherwise with zero.

For example:

||...|172|200|

s_gti

||...|0|

9.5.23 s_gtei

The s_gtei instruction replaces the two integer objects a and b on the top of the stack with a single object with a nonzero value if a is greater than or equal to b, otherwise with zero.

For example:

||...|172|172|

s_gtei

||...|-1|

9.5.24 s_lti

The `s_lti` instruction replaces the two integer objects a and b on the top of the stack with a single object with a nonzero value if a is less than b, otherwise with zero.

For example:

||...|172|200|

`s_lti`

||...|-1|

9.5.25 s_ltei

The `s_ltei` instruction replaces the two integer objects a and b on the top of the stack with a single object with a nonzero value if a is less than or equal to b, otherwise with zero.

For example:

||...|172|172|

`s_ltei`

||...|-1|

9.5.26 s_modi

The `s_modi` instruction replaces the two integer objects on the top of the stack with a single object with the rest of their division as value.

In case of division by zero this instruction prints an error message and terminates the execution of the program.

||...|a|b|

`s_divi`

||...|a mod b|

9.5.27 s_modi_i *n*

The `s_modi_i` instruction replaces the integer object on the top of the stack with the rest of the divion of it by the n parameter.

The program may **crash** if n is 0; no divide-by-error check is done since it makes *never* sense to use `s_divi_i 0`.

||...|a|

`s_modi_i` *n*

||...|a mod n|

`s_modi_i` *n* is faster than `pshci` *n* `;` `s_modi`, and even than `pshci` *n* `;` `s_f_modi`.

9.5.28 s_muli

The `s_muli` instruction replaces the two integer objects on the top of the stack with a single object with their product as value.

||...|a|b|

`s_muli`

||...|$a \cdot b$|

9.5.29 `s_muli_i` *n*

The `s_muli_i` instruction replaces the integer object on the top of the stack with the product of it and the n parameter.

||...|a|

`s_muli_i` *n*

||...|$a \cdot n$|

`s_muli_i` *n* is faster than `s_pshci` *n* ; `muli`.

9.5.30 `s_noti`

The `s_noti` instruction replaces the integer object a on the top of the stack with a nonzero value if a has zero value, else with zero.

For example:

||...|-34|

`s_noti`

||...|0|

9.5.31 `s_neqi`

The `s_neqi` instruction replaces the two integer objects on the top of the stack with a single object with value zero if the integer objects are equal, otherwise with a single object with a nonzero value.

For example:

||...|a|a|

`s_neqi`

||...|0|

9.5.32 `s_nxori`

The `s_nxori` instruction replaces the two integer objects a and b on the top of the stack with a single object with zero value if *exactly one* of a and b has a nonzero value, otherwise with a single object with a nonzero value.

||...|-34|0|

`s_nxori`

||...|0|

9.5.33 `s_ori`

The `s_ori` instruction replaces the two integer objects a and b on the top of the stack with a single object with a nonzero value if *at least one* of a and b has nonzero value, otherwise with a single object with zero value.

||...|0|-23|

`s_ori`

||...|1|

9.5.34 s_subi

The `s_subi` instruction replaces the two integer objects on the top of the stack with a single object with their difference as value.

| | ... | *a* | *b* |

`s_subi`

| | ... | *a* - *b* |

9.5.35 s_subi_i *n*

The `s_subi_i` instruction replaces the integer object on the top of the stack with the difference of it and the *n* parameter.

| | ... | *a* |

`s_subi_i` *n*

| | ... | *a* - *n* |

`s_subi_i` *n* is faster than `pshci` *n* `;` `s_subi`.

9.5.36 s_xori

The `s_xori` instruction replaces the two integer objects *a* and *b* on the top of the stack with a single object with a nonzero value if *exactly one* of *a* and *b* has a nonzero value, otherwise with a single object with zero value.

| | ... | -34 | 0 |

`s_xori`

| | ... | 1 |

9.5.37 subi $a $b $c

The `subi` *$a* *$b* *$c* instruction subtracts the content of the *$c* register from the content of the *$b* register, storing the result into the *$a* register:

$a := *$b* - *$c*

9.5.38 subi_i $a $b n

The `subi_i` *$a* *$b* *n* instruction subtracts *$n* from the content of the *$b* register, storing the result into the *$a* register:

$a := *$b* - *n*

9.5.39 swp $a $b

The `swp` *$a* *$b* instructions swaps the contents of the *$a* register and the *$b* register. The semantics of this instruction might be summarized as:

$a := *$b*, *$b* := *$a*

where the two assignments take place *in parallel*.

9.5.40 xori $a $b $c

The `xori` *$a* *$b* *$c* instruction stores a nonzero value into *$a* if exactly one of *$b* and *$c* has nonzero content, else it stores zero into *$a*.

9.6 Arithmetic/logic instructions on floats

9.7 Conversion instructions

9.8 Structures management instructions

9.8.1 `mka $a $b`

The `mka $a $b` instruction assigns to the $a register a new array with undefined content and with the content of the $b register as size.

9.8.2 `mka_i $a n`

The `mka_i $a $b` instruction assigns to the $a register a new array with undefined content and with size n.

9.8.3 `s_mka`

The `s_mka` instruction replaces the integer n on the top of the stack with a new uninitialized array of size n:

For example:

||...|3|

`s_mka`

||...|<???, ???, ???>|

9.8.4 `s_mka_i n`

The `s_mka_i n` instruction pushes a new uninitialized array of size n on the top of the stack:

For example:

||...|

`s_mka_i 2`

||...|<???, ???>|

9.8.5 `cns $a $b $c`

9.8.6 `s_cns`

9.8.7 `car $a $b`

9.8.8 `s_car`

9.8.9 `cdr` *$a* *$b*

9.8.10 `s_cdr`

9.8.11 `lkp` *$a* *$b* *$c*

9.8.12 `lkp_i` *$a* *$b* *n*

9.8.13 `s_lkp`

9.8.14 `s_lkp_i` *n*

9.8.15 `f_lkp` *$a* *$b* *$c*

9.8.16 `f_lkp_i` *$a* *$b* *n*

9.8.17 `s_f_lkp`

9.8.18 `s_f_lkp_i` *n*

9.8.19 `upd` *$a* *$b* *$c*

9.8.20 `upd_i` *$a* *n* *$b*

9.8.21 `f_upd` *$a* *$b* *$c*

9.8.22 `f_upd_i` *$a* *n* *$b*

9.8.23 `s_upd`

9.8.24 `s_upd_i` *n*

9.8.25 `s_f_upd`

9.8.26 `s_f_upd_i` *n*

9.9 Stack management instructions

9.9.1 `cpy`

9.9.2 `grw` *n*

9.9.3 `pop`

9.9.4 `popm` *n*

9.9.5 `s_swp`

9.9.6 `ld` *$r*

9.9.7 `st` *$r*

9.10 Flow control instructions

9.10.1 `j` *L:*

9.10.2 `jnz` *$r* *L:*

9.10.3 `jz` *$r* *L:*

9.10.4 `s_jnz` *L:*

9.10.5 `s_jz` *L:*

9.11 Subprograms management instructions

9.11.1 `cll` *n*

9.11.2 `clltr` *n*

9.11.3 ret *$r*

9.11.4 s_ret

9.12 Variables management instructions

9.12.1 lcl *$r n*

9.12.2 s_lcl *n*

9.12.3 nlcl *$r n m*

9.12.4 s_nlcl *n m*

9.13 Input/output instructions

9.14 Exception handling instructions

9.15 Special instructions

9.15.1 ccod *S*

The `ccod` *S* instruction exectutes the C code contained in the immediate string parameter *S*.

9.15.2 gc

The `gc` instruction performs a full garbage collection cycle, temporarily suspending the program.

This instruction is used when there is an urgent need of free memory. Normally the garbage collector would run concurrently with the program, without the need of executing eAM instructions for this purpose.

9.15.3 hlt *n*

The `hlt` *n* instruction halts the program. The integer parameter *n* is returned to the operating system as the process exit code.

9.15.4 nop

The `nop` instruction does absolutely nothing. It is used in contexts where an instruction is expected but no effect is needed, such as after a label which is the target of a jump.

For example:

```
    # Compute a number and store it into $4:
    ...
    jz $4 END_OF_PROGRAM:
    ...
  END_OF_PROGRAM:
    nop
```

`nop` instructions have **no** effect on runtime performance.

Part IV - Extending epsilon

This part deals with the problems of extending GNU epsilon with code written in other languages, and with the ways of interfacing epsilon with other languages.

Knowledge of the eAM (especially data representation, see Section 8.3 [Representation of epsilon data in the eAM], page 46) and experience with C programming are assumed.

10 C libraries

Say you want to access a database, or to do some 3D graphics in an epsilon program. These are not unusual requirements.

It's relatively easy to do that in C: there is an interface someone else has written (for example the client library of PostgreSQL `libpq`, or Mesa3D); you just have to call some already defined functions in your code.

Most other languages, such as C++, Java, Python and most Lisp implementations have some feature allowing you to call code written in C; this is a solution to the problem: instead of, for example, natively supporting PostgreSQL, most languages allow you to call your C code in some way, and your C code can make use of all the needed libraries.

This is also the solution provided by epsilon.

10.1 A wrong solution

Let us start with a *wrong* solution, which was actually implemented in an early version of epsilon. An "easy" way to extend the library is to extend the eAM: for example, if you need access to the C function `system()`, you can just add a new eAM instruction `sstm`, which takes a string, converts it from the epsilon format into the C format, and calls the `system()` function of the C library. They you must write some "glue" code to call the eAM instruction `sstm`: you will need an epsilon function taking a string and returning an action of integer:

```
execute_program : string -> i/o of integer
```

Note that now **`execute_program` will have to be written in eAML!** There is no way around this[1], since you need to use the eAM instruction `sstm`, which is not generated by the epsilon compiler.

Also note that the aAM has changed: the new instruction has made the abstract machine incompatible with the old version; all epsilon libraries must be recompiled, too.

This solution is wrong for several different reasons:

- It incompatibly changes the eAM
- It is complicated and error-prone to write eAM instructions
- It makes the generated C code grow
- Writing the glue function in eAML is **very** tedious and error prone
- For linking new libraries at compile time you also have to update the Autoconf/Automake machinery
- ...

10.1.1 The right solution

The right solution is writing the extension code at a level which is somewhere in the middle between eAML and epsilon; you must be able to do that without recompiling the eAM, to retain compatibility.

The eAM provides a mechanism for linking a *C library* at initialization time. The C library can define some functionalities (either limited to pure calculus or comprising I/O) to be made available to epsilon. Often a C library needs to refer to some other shared library written to be called from C (such as `libpq`). This can be linked at initialization time, too, and is called a *dynamic library*. Dynamic libraries are not directly visible from epsilon.

To do: I don't like this chapter. Rewrite most of it.

[1] Except modifying the compiler.

11 Using epsilon with Scheme

Part V - Examples

In this part some nontrivial examples of epsilon programs are presented; they are not necessarily useful by themselves or complete, but nonetheless they show some usage patterns quite common in epsilon programming.

In this context formal elegance and readability is considered more important than efficiency.

Of course also the example programs are free software, covered by the GNU General Public License (see [Copying], page 1). The complete source code for all these examples is distributed along with GNU epsilon.

12 mu-lisp

μ-lisp is a simple Lisp[1] interpreter with static scoping, fully written in epsilon.

Like any other Lisp implementation μ-lisp is interactive: the implementation of its REPL constitues a good example of purely functional I/O.

It is a Lisp/1, i.e. it has a single namespace for variables and functions.

To do: scanner and parser...

To do...

The predefined symbols are...

To do: source overview

To do: example programs.

[1] For more information about Lisp, and for realistic implementations, see `http://www.lisp.org`.

13 mu-basic

To do

Appendices

This part will be a collection of various important non-technical information related to epsilon, licenses and references.

Appendix A Copying This Manual

A.1 GNU Free Documentation License

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover

Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to

A.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

P

Q

R

S

T

U

W

Y