Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

Programa de Pós-Graduação em Engenharia Elétrica

MAURO JACOB HONORATO

WAM BASED SPACE EFFICIENT PROLOG IMPLEMENTATION IN LISP

Banca Examinadora:

Luciano Vieira Lima, Dr. UFU. Orientador (UFU)

Marcelo Rodrigues Sousa, Dr. (UFU)

Antônio Eduardo Costa Pereira, Dr. (UFU)

José Lopes de Siqueira Neto, Dr. (UFMG)

Reny Cury Filho, Dr. (PMU)

Uberlândia - MG

2016

MAURO JACOB HONORATO

WAM BASED SPACE EFFICIENT PROLOG IMPLEMENTATION IN LISP

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia como requisito parcial para obtenção do título de Doutor em Ciências.

Área de concentração: Processamento da Informação

Orientador: Prof. Dr. Luciano Vieira Lima

Uberlândia - MG

2016

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

H77
2016

Honorato, Mauro Jacob, 1981-
Wam based space efficient Prolog implementation in Lisp / Mauro Jacob Honorato. - 2016.
97 f. : il.

Orientador: Luciano Vieira Lima.
Tese (doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.

1. Engenharia elétrica - Teses. 2. Prolog (Linguagem de programação de computador) - Teses. 3. COMMON LISP (Linguagem de programação de computador) - Teses. I. Lima, Luciano Vieira, 1960- II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

WAM BASED SPACE EFFICIENT PROLOG IMPLEMENTATION IN LISP

MAURO JACOB HONORATO

Tese apresentada por Mauro Jacob Honorato à Universidade Federal de Uberlândia (UFU) como parte dos requisitos para obtenção do título de doutorado.

Luciano Vieira Lima, Dr.	Darizon Alves de Andrade, Dr
Orientador	Coordenador do
	Curso de Pós-Graduação

Dedico esse trabalho aos meus pais, Maurício e Maria, à minha esposa Leiliane, às minhas filhas Olívia e Lavínia, e ao meu irmão Márlon.

Contents

List of Figures

List of Tables

1 Agradecimentos

Em primeiro lugar agradeço à Deus que me conheceu a substância ainda informe no útero de minha mãe e me amou antes da criação do mundo, não levando em conta todo o meu pecado ofereceu seu único filho na cruz, tornando-se Ele mesmo maldito em meu lugar. Pois eu também sei que o meu Redentor vive!

Agradeço ao meu pai e à minha mãe, Maurício e Maria, que com muita luta e sacrifício me forneceram o necessário para chegar até aqui. Além disso, mesmo com um aperto no coração sempre apoiaram meus projetos para um futuro melhor.

À minha esposa, Leiliane, agradeço por seu amor, por estar sempre ao meu lado não importando a loucura que resolvo fazer ou para onde escolho nos levar, por suportar todo o enfado que foi esse trabalho e por comprar minhas ideias e mais ainda minhas mudanças de planos fornecendo-me fôlego redobrado. Às minhas filhas, Olívia e Lavínia, por todo o tempo com o papai que perderam para esse projeto, agora sim, o papai não mora mais na "fufu"!

Aos professores Luciano e Costa por me aceitarem em seus projetos e orientarem, conduzindo-me ao êxito não só na academia, mas também na vida profissional, saibam que sou grato a vocês dois pelo professor que me tornei e por tudo o que minha família pode desfrutar através desse emprego.

2 Abstract

This thesis proposes the implementation of a space efficient Prolog implementation based on the work of David H. D. Warren and Hassan Aït-Kaci. The Common Lisp is the framework used to the construction of the Prolog system, it was chosen both to provide a space efficient environment and a rich programming language in the sense that it supply the user with abstractions and new ways of thinking. The resulting system is a new syntax to the initial language that runs on top of the SBCL Common Lisp implementation and can abstract away or exploit the underlying system.

2.1 Keywords

Warren Abstract Machine, Common Lisp, Prolog

3 Resumo

Esse trabalho propõe a implementação de um sistema Prolog eficiente no espaço, o mesmo é baseado nos trabalhos de David H. D. Warren e Hassan Aït-Kaci. A Common Lisp é a estrutura usada para a construção do sistema Prolog, ela foi escolhida tanto por fornecer um ambiente eficiente no espaço quando por ser uma linguagem de programação rica no sentido de que fornece ao usuário abstrações e novas maneiras de pensar. O sistema resultante consiste em uma nova sintaxe aplicada à linguagem inicial que funciona sobre a implementanção Common Lisp chamada SBCL e é capaz de abstrair ou explorar o sistema subjacente.

3.1 Palavras-chave

Warren Abstract Machine, Common Lisp, Prolog

4 Introduction

The present work is the result of four to five years of work in the Artificial Intelligence Laboratory of Faculty of Electrical Engineering at Federal University of Uberlândia (UFU). In the meantime we studied a bunch of other subjects including Kindle device programming, Financial Calculators design, Android device programming, item response theory, optical recognition systems (Lush), and Stenography, one of the few things these subjects had in common was the use of the Common Lisp language.

During the study of the Common Lisp language it became obvious that write domain-specific languages in it is not an option but the way it works. When the Stenography subject claimed a Prolog system in order to implement chunk grammars and connect it later with Emacs, we take into account the easiness of extending the first-order logic in Prolog, the build-in depth-first search algorithm, and pattern matching. Then the Prolog system on top of Common Lisp became important enough to deserve all this discussion.

Although the Prolog audience is broad enough, we had to narrow the objectives, the consequences and its causes are described in the next section Computers and limitations, there we explain the chosen operating system, the Gnu Emacs and how it smoothes the interaction with a so called inferior Lisp implementation though its Slime mode. The *inferior* word denotes an external Lisp session that run as a subprocess or *inferior process* of Emacs and has nothing to do with the power of the implementation.

The implementation of another programming language on top of Common Lisp requires a change in the Lisp syntax and two important facilities from the language to do this are described in the section New syntax. More than defining functions we are changing the way the language works and looks like. Even the Common Lisp language specification provides another syntax called Loop that does not resembles Lisp but an English-like construction.

After these sections is time to analyze the space efficiency of the implementation where the Prolog system resides, the section Space efficency treat this. The Java[1] language environment (language and virtual machine) was chosen as the analysis parameter. The use of Common Lisp vectors showed to be the right choice and were employed in the Prolog system.

The culmination of the work is explained in the section The logic system. It is described the more important things and how the system was adapted from the Warren's Abstract Machine (WAM). The main abstractions and structures rest there near a modest Prolog clarification.

[1] Java is trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and certain other countries.

5 Computers and limitations

At the time of this writing, the computer industry seams to be at least a couple of decades away from reaching hard physical limits, according to Nature[9] and Arstechnica[14]. Reaching that limit would not just stuck but halt the progress in computer processing power.

First we had a mechanical computer with Charles Babbage, which gave Ada Lovelace the title of the first programmer, then came the vacuum tubes powered computers, after that transistors, and today the integrated circuits set the pace. But we are always begging for more, the people are already wearing computer devices and that is not enough.

A team of Stanford University is working hard to change the way the computation happens[10], bringing logical circuits in the form of biological cells, with the possible creation of what would be a biological computer, this is one that lays on your hand skin or even in your front head, the important thing is that such a computer is part of your body, living and dying with you.

In the same pace of processing power improvements is the use of data, with deep learning and big data, computers are doing things that go from automatic face recognition in personal photographs to beat human beings at ancient games[5]. The next logical step seams to be to predict whether a couple of visitors in a shopping mall would cause people or money damage. This kind of decision brings logic programming to decision trees.

Given the wide range of computers today, we must show what kind of computer hardware and software support our logic system.

5.1 Operating system

In order to abstract and manage the hardware we need an operating system (*OS*), today three of the most used are Mac OS[2], Windows[3] and GNU/Linux. We choose the Ubuntu[4] distribution out of the GNU/Linux world given that it is open source, secure and accessible through its translations. Our logic system was written in the Common Lisp language and there are few *OS* dependent features and someone willing to portate it to another *OS* will have to read the compiler documentation.

Some years ago we could choose a Lisp Machine, but today there are few of them available, besides that you can use Emacs from Gnu[5], it resambles a Lisp Machine in the sense that it is an operating system with lots of applications written in Lisp. You can rest assured that if something doesn't looks like what you wanted you will have a Lisp language at your disposal to

[2]Mac OS is a registered trademark of Apple Inc.

[3]Windows operating system is a registered trademark of Apple Inc.

[4]Ubuntu is a registered trademark of Canonical Ltd.

[5]Available at `https://www.gnu.org/software/emacs/`

fix that, the language is called Emacs Lisp and every single Emacs Module is written in it.

5.2 Application software

Despite the fact that our proposed logic programming system runs on top of the SBCL implementation of the Common Lisp language, you still need application software, just the one called Emacs, and as an application Emacs has a purpose, let you work without interruptions. As long as you stay within Emacs, you can work with directories, files, source code, version control systems and a special interaction Mode for your Common Lisp implementation called Slime.

Slime is an Emacs Mode specially designed to the development in Common Lisp. There is a list of its best features and we narrow it here:

- Code evaluation
- Online documentation
- Integration of REPL with Emacs

5.2.1 Code evaluation

Using Slime you can run code snippets in a file without the copying/paste sequence. The alternative to this would be to comment all the code but the chosen and use the load function, as an example you could load just the following **subtrai** function by commenting the code this way:

```
#|
(defun soma(x y)
   (+ x y))
|#
(defun subtrai(x y)
   (- x y))
#|
(defun multiplica(x y)
   (* x y))

(defun divide(x y)
   (/ x y))
|#
```

And loading the file inside the SBCL with the next LISP expression:

```
(load "operacoes.lisp")
```

If you were using Slime, with the following buffer opened, what you need to do is just press the sequence `C-c C-c` after the subtrai's closing parentheses in the last column of line 5:

```
 1  (defun soma(x y)
 2     (+ x y))
 3
 4  (defun subtrai(x y)
 5     (- x y))
 6
 7  (defun multiplica(x y)
 8     (* x y))
 9
10  (defun divide(x y)
11     (/ x y))
```

And the Common Lisp implementation (SBCL) gives a hint of what is going on in the Slime REPL buffer:

```
CL-USER> ; compiling (DEFUN SUBTRAI ...)
```

Only subtrai's definition were given to the **read** function and no comments or copy/paste were needed.

5.2.2 Online documentation

What happens when you forget a Common Lisp keyword in the middle of an algorithm translation? First you would open a browser and point to **The Common Lisp HyperSpec** searching for the function name from the full text of the ANSI standard. If find nothing or the function was developed by yourself, you could try to deduce from the documentation string with the **documentation** function.

```
(documentation 'subtrai 'function)

NIL
```

If you were using Slime, nothing of the previous options would be need, as long as you just forget the name of a keyword, Slime gives you a hint as soon as it receives a function name, for instance if you start typing a call to **with-open-file** the hint below would be shown in the mini-buffer:

```
(with-open-file
   (stream filespec &rest options
   &key (direction :input) (element-type 'base-char)
```

```
      if-exists if-does-not-exist
      (external-format :default) (class 'sb-sys:fd-stream))
  &body body)
```

Even with your **subtrai** function you would not be totally lost given that typing `(subtrai` you would receive the following Slime hint:

```
(subtrai x y)
```

The reader must agree that this hint is much better than the **nil** received from the previous **documentation** command.

5.2.3 Integration of REPL with Emacs

There is an endless cycle inside all LISP systems that is called REPL, it is always reading LISP expressions, evaluating them and printing the result. SLIME is an Emacs mode that was developed to integrate the REPL from LISP systems (or implementations) within the Emacs editor, providing also "shortcut" commands.

In order to install Slime, the Emacs user needs to add the following lines to the **.emacs** initialization file. It resides on the root folder reached with ∼/.

```
1  (add-to-list 'load-path "~/progs/2016/slime")
2  (require 'slime-autoloads)
3  (setq inferior-lisp-program "/usr/local/bin/sbcl")
4  (setq slime-contribs '(slime-fancy))
```

The first and third lines are the most important. The first specifies where you did put the SLIME files. The third one indicates the LISP implementation that is to be integrated into Emacs through the SLIME mode.

5.2.4 The MELPA installation

Today, no Emacs documentation is complete if it does not say anything about MELPA, this is a better way to do installation on Emacs. The reader does not need to extract installation files by hand and interact with the Emacs initialization file anymore, in the same extent that the Linux users are doing less and less installations like the needed by the **Linux from scratch** [6].

What one do when it needs to install another software in his Linux distro? He will probably do something that resembles the Ubuntu **apt-get install name** command. Emacs has evolved and the Milkypostman's Emacs Lisp Package Archive also known as **MELPA** lets the install work become easier, like **apt-get** does in Linux. You still need to open the ∼/**.emacs** file and add the following lines, but once and for all.

[6] Follow `http://www.linuxfromscratch.org/` and build your own Linux system.

```
(require 'package) ;; You might already have this line
(add-to-list 'package-archives
             '("melpa" . "https://melpa.org/packages/"))
```

In the next Emacs initialization, after it reads those lines, the world of MELPA will be at you fingers, and you will find and install Emacs packages like SLIME with the following Emacs command:

```
M-x package-install slime-mode
```

There is the possibility of listing all the packages available with:

```
M-x package-list-packages
```

You can search the results with **C-s slime-mode**, when you find the package, click in its name to initiate the installation buffer. Only your mouse device will be needed, no more **tar** nor **mv** commands just button clicks. The SLIME tighten integration of the Emacs with the Common Lisp implementation raises the question of which implementation to choose and that is the matter of the next section.

5.3 Common Lisp Implementation

The Common Lisp language is defined by its standard and the language is implemented by many entities. Even the reader could start small with a book like Build Your Own Lisp from Mr Daniel Holden[7] and then read the ANSI Common Lisp standard implementing all the features that define the language producing a conforming Common Lisp implementation.

We have not the time and resources needed to write a conforming implementation so we had to choose between the following list of Common Lisp systems:

- CLISP
- CMUCL
- ECL
- CCL
- SBCL
- ABCL
- MKCL

Our choice was SBCL, given that we are familiar with it, it is open source, and provides native compiler. SBCL is thus complete to our system and is alive in the sense that the developers are still working on it[7]. Besides that it does native code compilation very fast.

5.3.1 A taste of native compiler:

Let's show you that native compiler matters. In SBCL, every single form given to **eval** is translated into native code by the native code compiler. If someone declare the following function:

```
(defun soma(x y)
   (+ x y))
```

He could even see the generated assembly code with:

```
(disassemble 'soma)
```

And would receive something like the following:

```
; 98:       48894DF8         MOV [RBP-8], RCX
; 9C:       488BD6           MOV RDX, RSI
; 9F:       488BFB           MOV RDI, RBX
; A2:       41BBC0010020     MOV R11D, 536871360
; A8:       41FFD3           CALL R11
; AB:       488B5DE8         MOV RBX, [RBP-24]
; AF:       488B75F0         MOV RSI, [RBP-16]
; B3:       488BE5           MOV RSP, RBP
; B6:       F8               CLC
; B7:       5D               POP RBP
; B8:       C3               RET
; B9:       CC10             BREAK 16
```

The third column shows the Assembly code that was generated by SBCL and consists of everything needed by the `soma` function.

Even if the reader doesn't know anything about that encrypted code, understanding that the Assembly language is the programming language closest to the Machine language is enough to know that such a snipet of code would in no doubt execute much more fast than any interpreted counterpart given that there is a strong correspondence between the generated code and the architecture's machine code instructions.

The ABCL LISP implementation, for example, is a system that generates Java bytecodes not machine code. In order to see those bytecodes running

[7] At the time of writting this document, the most recent version was released July 31, 2016.

you need another layer in-between, something that translate those bytecodes into machine language, this is the work of the Java Virtual Machine. Some projects can waste running time and memory using the Java environment, our logic system must be fast and ABCL does not fit in.

After the description of why a system that does not work, we will go straight to the chosen system, SBCL.

5.4 SBCL and CMUCL

Actually the SBCL implementation is what would be called a *fork* of the CMUCL implementation, CMUCL is not outdated but is less actively developed than SBCL. CMUCL, as SBCL, is a implementation of the Common Lisp programming language that conforms to the ANSI Common Lisp standard.

However, as stated in its own FAQ, CMUCL does not have Unicode support nor native threads on Linux/x86 platforms and SBCL is closer to the ANSI Common Lisp specification than CMUCL. Both compiled code runs at a similar speed and CMUCL would be the right one only if one needs a faster compiler.

Before the description of the logic system, it is necessary to show a couple of important characteristics of LISP that allows the construction of new languages by changing the syntax of itself. Far away from providing just list processing facilities, the Lisp implementations provide new ways of thinking at the moment of programming, this is the element of the next section.

6 New syntax

It's remarkable that Lisp provides facilities to add new constructs to the language, the defmacro function allows someone to define new operators, and these operators can circumvent the idiosyncrasies of the so called LISt Processing language.

One of these useful Common Lisp idioms is backquote, very well described by Doug Hoyte in Let Over Lambda[8], it stops the evaluation of a form, except on the unquoting areas where an expression is evaluated and the result inserted where the unquote mark appears. We took advantage of this feature in our logic system to create code templates that are filled at runtime like the `add_cl` calls in the following `addnot` function definition.

```
 1  (defun addnot(s)
 2    (let* ( (n (length (cdr s)))
 3    (args (mknotvars 0 (cdr s)))
 4     (pred (car s))
 5     (notpred (not-symbol-name pred)) )
 6      (when (not (get notpred 'def))
 7        (add_cl notpred
 8         `(,n ( ,notpred ,@args)
 9      ( ,pred ,@args) (! ,n) (|fail|)) 'def)
10        (add_cl notpred  `(,n (,notpred ,@args)) 'def )
11        (format t "notdef: ~a :- ~a~%" notpred (get notpred 'def)))))
```

The previous `addnot` function, shows an example of backquote in action, the variable `n` is the length of the rest of the `s` list, as its peers, dependent of the value that `addnot` receive as argument at the moment of the call. And the lines 8, 9 and 10 are the result of the kind of improvement the former List Processing language acquired, actually a change on its syntax.

Another two important expressions that change the LISP syntax and deserve descriptions are `defmacro` and the `loop` macro.

6.1 DEFMACRO

In order to talk about defmacro we can go straight to the point, for instance, instead of writing a series of nested IFs, the user can get ride of typing a lot (and lots of typos) with WHEN, UNLESS and COND macros. Writing part of a program to choose the right package from correios, a newbie Lisp programmer could do this:

```
1  (if (< volume (* 18 13 9))
2      (print "Tipo 1")
3      (if (< volume (* 27 18 9))
4   (print "Tipo 2")
```

```
5   (if (< volume (* 36 27 18))
6       (print "Tipo 3")
7       (print "Tipo 4"))))
```

But with two more missing types of packages, known for its volume, its clear that the following code would be much more clear and concise.

```
1  (cond
2    ((< volume (* 18 13 9)) (print "Tipo 1"))
3    ((< volume (* 27 18 9)) (print "Tipo 2"))
4    ((< volume (* 36 27 18)) (print "Tipo 3"))
5    (t (print "Tipo 4")))
```

Thank God that Lisp is not like Python, as Tim Peters advocates in the Python Developer's Guide,

> There should be one– and preferably only one –obvious way to do it.

Using defmacro the Lisp language user can find himself common constructions inside his code and give birth to new operators in order to become more efficient by using the new abstraction. He is not even tied to the Lisp syntax or language standard itself, as he is actually adding the abstraction to the language, he just need to be concise. Thus there are lots of ways to do one thing in Lisp.

When reading the source code of our logic system one will understand that a list is a two arg functor with descritors given by `(\.)` for originals, and `(\. . t)` for copies, a new predicate was defined with defmacro in the following code from our logic system:

```
1  (defmacro list? (x)
2    `(eq (functor (functorDescription ,x)) '\.))
```

Instead of writting the full expression in line 2, the programmer needs just to use the new expression `list?`.

6.2 LOOP

The most prominent of these other syntax operators is the LOOP macro, by providing a distinct way to express convolution it lets a Lisp hacker to express himself in an English-like language in order to achieve a goal. See this loop example:

```
1  (loop for i from 1 to 10 collecting i)
```

The same result could be achieved with a DO macro, starting with a null list, iterating from 1 to 10, PUSHing every step number into the list, and reversing the list to give as the result. It's as difficult to express as to write and read the following code:

```
1  (do ((lista nil) (i 1 (1+ i)))
2      ((> i 10) (nreverse lista))
3    (push i lista))
```

Someone may argue that the LOOP way is not LISP like, but the English-like constructions in the form of LOOP macros allows everyone to guess what is going on even if he does not master the LOOP language yet. Hence it decreases the time needed to someone else trying to understand the original code.

```
1  (loop for x from 1 to 10 summing x)
```

Despite the fact that the LOOP syntax is different from Lisp, all the Lisp code inside LOOP is applied in the same way it is in the outside. This shows that there is a connection between the Lisp syntax and new syntaxes provided by DEFMACRO, thus extending the Lisp language not taking it off the play.

```
1  (loop for i below 10
2        and a = 0 then b
3        and b = 1 then (+ b a)
4        finally (return a))
```

Here `(+ b a)` and `(return a)` are side by side with the english-like expressions, the user acquainted with both LISP and LOOP syntax will apply the obvious one in the right place.

In the following code snippet, it is clear that a list of x is being constructed from the call to `getOutputVariables`:

```
(defun getOutputVariables(c)
  (let* (  (fn (car c))
  (mods (cdr (get fn 'mod)))
  (args (cdr c)))
    (loop for x in args
  for m in mods
 when  (equal m '+) collect x)))
```

A cryptic function would be the outcome of the use of `setf` and `cons` to construct the same previous result. The Common Lisp language and its environment, like any other programming language, should be used by its advantages not only because of any cumbersome language syntax. We proceed in analyze its advantage in terms of space efficiency.

7 Space efficiency

In order to analyze space efficiency of a system one needs to establish memory space goals as a system parameter, it is becoming consensus to the contemporary computer users the fact that extra memory use is not a huge concern as important as is the time efficiency. A good example of this is the proliferation of Java applications despite its memory usage.

The following Java program (`hello.java`) contains the source code necessary to the Java compiler produce another file called `hello.class`. The class file contains `bytecodes` that are executed by the Java Virtual Machine (JVM).

```
import java.util.Scanner;

public class hello {
    public static void main(String[] args) {
String nome;

Scanner user_input = new Scanner( System.in );
nome = user_input.next();
    }
}
```

The size of the class file is just 580 bytes:

```
$ ls -lia hello.class
21890030 -rw-rw-r-- 1 mauro mauro 580 Jul 18 09:23 hello.class
```

At the execution time the class file must be loaded consuming 580B and because there is a wait for input, we can infer that two objects would be alive in memory plus the program class through its static method, but looking for the memory usage with the `top` command, we get this:

```
 1  $top u mauro
 2  top - 09:25:46 up 4 days, 15:36,  3 users,  load average: 0,85, 0,68, 1,58
 3  Tarefas: 219 total,   1 executando, 218 dormindo,   0 parado,   0 zumbi
 4  %Cpu(s): 11,1 us,  2,3 sy,  0,3 ni, 85,3 id,  1,0 wa,  0,0 hi,  0,0 si,  0,0 st
 5  KiB Mem:   3943036 total,  3148888 used,   794148 free,   180336 buffers
 6  KiB Swap:  4087804 total,   957356 used,  3130448 free.   924112 cached Mem
 7
 8    PID USUÁRIO  PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 9  12659 mauro     20   0  881868 105872  32988 S  32,8  2,7   1:11.96 chrome
10   2057 mauro     20   0 1543204  64496  14268 S   7,0  1,6  23:59.23 compiz
11   6791 mauro     20   0 1480960 371712  51688 S   6,3  9,4  13:03.02 chrome
12   6715 mauro     20   0 1243992 141200  35208 S   4,3  3,6  24:19.91 chrome
```

```
13  27953 mauro     20   0 1333332  72080   5484 S   1,7  1,8   2:16.06 mono
14  22679 mauro     20   0  632576  15632   8564 S   1,3  0,4   0:09.24 gnome-term+
15   1927 mauro     20   0   20360    416    416 S   0,3  0,0   0:43.08 syndaemon
16  11061 mauro     20   0 1036112  89616  19548 S   0,3  2,3   4:04.28 chrome
17  12797 mauro     20   0   29220   1804   1248 R   0,3  0,0   0:01.79 top
18  13430 mauro     20   0 3335040  23976  11152 S   0,3  0,6   0:00.26 java
19  29902 mauro     20   0  532516  12584   3304 S   0,3  0,3   0:22.97 chrome
```

The important thing to note is that the RES column indicates the Resident memory used by a process (KiB), with only our `hello` application running as a Java application (line 18), 23MB are being used. A very simple program is consuming 0.6% of the physical memory and 0.3% of the CPU time. One can think that the Scanner class is very big and this justify the memory usage, but if we create an infinity loop without the Scanner object like the one in the following class:

```
public class hello {
    public static void main(String[] args) {
for( ; true ;  ) {}
    }
}
```

The resulting RES is still high, now near 18MB, this is because of the Java Virtual Machine machinery and this is an intrinsic characteristic of Java applications:

```
top - 09:51:49 up 4 days, 16:02,  3 users,  load average: 0,81, 0,56, 0,73
Tarefas: 220 total,   1 executando, 219 dormindo,   0 parado,   0 zumbi
%Cpu(s): 26,0 us,  0,2 sy,  0,0 ni, 72,8 id,  1,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   3943036 total,  3374724 used,   568312 free,   190468 buffers
KiB Swap:  4087804 total,   898592 used,  3189212 free.   989224 cached Mem

  PID USUÁRIO  PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13808 mauro     20   0 3334888  17924  10508 S  99,7  0,5   0:27.84 java
```

Applying Lars Vogel method[15] to obtain the total application used memory, we apply the `totalMemory` and `freeMemory` methods from the Runtime class. According to the Java API Specification[1], the `totalMemory` method returns the total amount of memory in the Java Virtual Machine, this is the total amount of memory currently available for current and future objects and is measured in bytes, while the `freeMemory` method returns the amount of free memory in the Java Virtual Machine and is an approximation to the total amount of memory currently available.

```
import java.util.Scanner;
```

```
public class hello {
   public static void main(String[] args) {
Runtime runtime = Runtime.getRuntime();
String nome;
Scanner user_input = new Scanner( System.in );

runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();
System.out.println("Used memory in bytes: " + memory);

nome = user_input.next();
    }
}
```

The amount of Java Virtual Machine (JVM) memory spent become clear, while the JVM is consuming 23MB only 0.4MB were actually employed inside the hello class:

```
$ java hello
Used memory in bytes: 445936
```

The following stripped hello class shows that the Scanner object consumes less than 0.2MB of memory:

```
public class hello {
    public static void main(String[] args) {
Runtime runtime = Runtime.getRuntime();
runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();

System.out.println("Used memory in bytes: " + memory);
for( ; true ;  ) {}
    }
}
```

The used memory inside the program is near 0.3MB without the Scanner class, while the Java application is consuming near 18MB of the system memory:

```
$ java hello
Used memory in bytes: 263984
```

Some Java-based systems, like for example Trader Workstation[8], being aware of the memory issues inherent to the Java Virtual Machine's automatic storage management system, warns their users and teach them how to increase the memory available to the Java environment as heap space.

Notwithstanding the fact that such modifications are easy like the following command, what should be noticed is that values like 3GB of memory for the Java environment only[9] are nowadays accepted without questioning:

```
java -cp j.jar:t.2.jar -Xmx768M -XX:MaxPermSize=256M jc.LoginFrame
```

The Heap profiling for space-efficient Java[13] article shows that there must be some level of concern about space efficiency even among Java programmers given that this can also positively impact the runtime efficiency of the program. The Java garbage collector (GC) automatically dealocates memory but the time of the GC operation is not controlled. Hereupon the Java programmer must be aware that his control over the application memory usage inside the Java environment is limited.

These Java levels were chosen as benchmark for our system parameters. An easy test can be applied to compare the way that the Java application behaves compared to Lisp, in order to minimize the influence of our expertise into the tests we must keep the programs and problems as small as possible assuming the risk of touching border issues.

It can be claimed that the difference between the Common Lisp language and its peers makes the comparison very hard. However as *a programmable programming language*[12] LISP has easily adopted new paradigms, one of this is the object system through the Common Lisp Object System (CLOS) that we will employ to construct a Java equivalent Lisp program, trying to compare oranges and oranges.

Our logic system has to deal with stacks, a stack as stated by the CLRS book on Introduction to Algorithms[3] is a dynamic set that implements a last-in, first-out policy (LIFO) in which the element removed is the one most recently inserted. The backtracking system (the mechanism for finding multiple solution) will handle the stack.

The Node object in the Java language is defined by the next Node class file:

```
public class Node{
    private int position;
    public int getPosition() { return position; }

    public Node next;
    public Node(int p, Node n) {
```

[8] See `https://ibkb.interactivebrokers.com/article/2170`.

[9] See `http://www.wikihow.com/Increase-Java-Memory-in-Windows-7`.

```
position = p;
next = n;
    }
}
```

Note that the result is the following linked list, the `node` variable points to the top of the stack, the pointer to null marks the bottom of the stack:

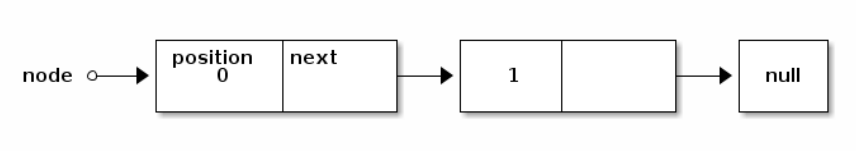

Figure 1: Stack

The following Lista class is responsible for the creation of the stack and a modest memory measurement in Java, the resulting data of changing the size of the list is organized and showed in table 1 and figure 2.

```
import java.util.Scanner;

public class Lista {
    public static void main(String[] args) {
Scanner next = new Scanner(System.in);
Node no = new Node(0,null);

for(int i = 1; i < 100000000; i++) {
    no = new Node(i,no);
}

Runtime runtime = Runtime.getRuntime();
runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();

System.out.println("Used memory in bytes: " + memory);
next.nextInt();
    }
}
```

The Java program allocates more memory (RSS) than it is really necessary, and the allocation does not increase in the same rate as the bytes consumption function. The slope of RSS line (m_{rj}) is 68% greater than bytes consumption line (m_{bj}).

$$m_{bj} = \frac{240425336 - 24425336}{10000 - 1000} = 24 \tag{1}$$

Table 1: Java program, measure of bytes consumption and memory allocation (RSS)

List length	Bytes used	RSS(kB)	RSS(Bytes)
0.01k	446672	22568	23109632
0.1k	448832	22732	23277568
1k	470432	22864	23412736
10k	686432	26080	26705920
100k	446520	26944	27590656
1000k	24425336	67212	68825088
10000k	240425336	432072	442441728
100000k	-	-	

$$m_{rj} = \frac{\Delta y}{\Delta x} = \frac{432072 - 67212}{10000 - 1000} = 40.54 \tag{2}$$

Another important thing is that when the length of the stack comes close to one hundred million a Java unchecked exception occurs:

```
$ java Lista
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at Lista.main(Lista.java:9)
```

The Lisp code corresponding to the Java Node class is the following one, it has two slots `position` and `next`, both of them serve the same purpose as the equivalent written in Java. The `main` function produces the stack and we vary the length in order to evaluate the memory consumption. The result of the change in the length and execution of the Lisp program are summarized in table 2 and displayed in figure 3.

```
(defclass Node ()
  ((position
    :initarg :position
    :accessor pos)
   (next
    :initarg :next
    :accessor nxt)))

(defparameter *node* (make-instance 'Node :position 0 :next nil))

(defun main()
  (loop for i from 1 to 1000000 do
       (setf *node* (make-instance 'Node :position i :next *node*)))
  (format t "!")
  (finish-output nil)
```

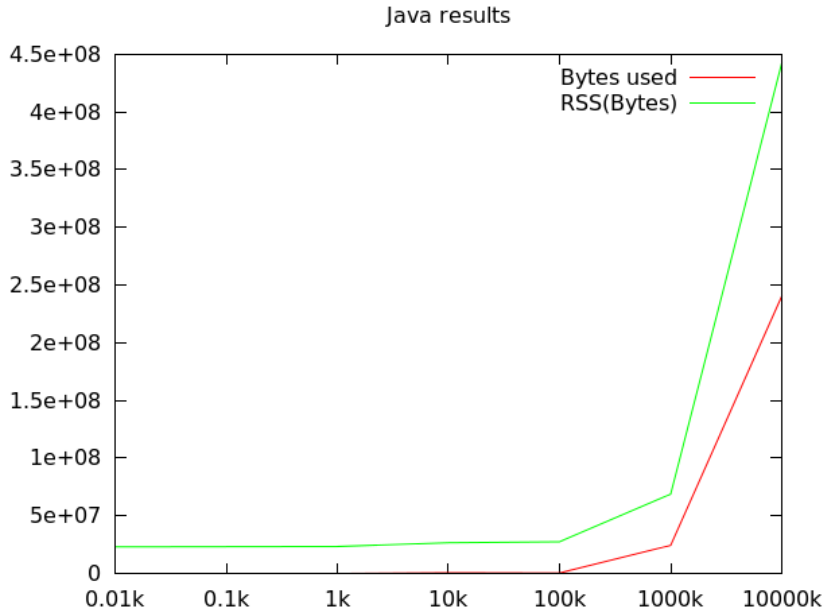

Figure 2: Java program, measure of bytes consumption and memory allocation (RSS)

```
(gc :full t)
(read-line))
```

Although the consumption and allocation of memory in Lisp programs increase in a similar rate after 1MB, the slope of RSS line (m_{rl}) is equal to the slope of bytes consumption line (m_{bl}), despite this behavior it is greater than the Java equivalent and could not alone justify the Common Lisp usage.

$$m_{bl} = \frac{640869472 - 64545808}{10000 - 1000} = 64 \tag{3}$$

$$m_{rl} = \frac{\Delta y}{\Delta x} = \frac{696112 - 120000}{10000 - 1000} = 64 \tag{4}$$

Again the large stack length near one hundred million overflows the heap size which leads to a fatal error:

```
fatal error encountered in SBCL pid 17056(tid 140737353971520):
Heap exhausted, game over.
```

It is possible, as is the case for Java applications, to adjust the heap size of the SBCL Common Lisp implementation at the time of its execution, the flag `--dynamic-space-size` is the responsible for the newer size, as long as there is available resources the user can increase this value:

```
rlwrap sbcl --dynamic-space-size 2048 --load plist.lisp
```

We used the `save-lisp-and-die` function to produce the executable file, along with the name of the executable file the name of the first function to execute by the toplevel is informed. In order to save the configuration the keywords `save-runtime-options t` and `executable t` were added:

```
(defun tempo() (time (main)))

(save-lisp-and-die "htime"
   :executable t
   :toplevel #'tempo
   :save-runtime-options t)
```

Table 2: Lisp program, measure of bytes consumption and memory allocation (RSS)

List length	Bytes used	RSS(kB)	RSS(Bytes)
0.01k	549920	15384	15753216
0.1k	549920	15384	15753216
1k	615456	15384	15753216
10k	1205280	15912	16293888
100k	6972448	22584	23126016
1000k	64545808	120000	122880000
10000k	640869472	696112	712818688
100000k	-	-	

A humble comparison between bytes consumption in Java and Lisp can be seen in table 3 and displayed in figure 4.

Table 3: Comparison between the claimed bytes consumption.

List length	Bytes used in Java	Bytes used in Lisp
0.01k	446672	549920
0.1k	448832	549920
1k	470432	615456
10k	686432	1205280
100k	446520	6972448
1000k	24425336	64545808
10000k	240425336	640869472

The bytes used inside the applications can be counted differently, given that Java and Lisp deal with typing differently, and this may influence the Bytes used results, but the RSS is the allocated memory from the system and the method to measure it for both applications is the same, the difference

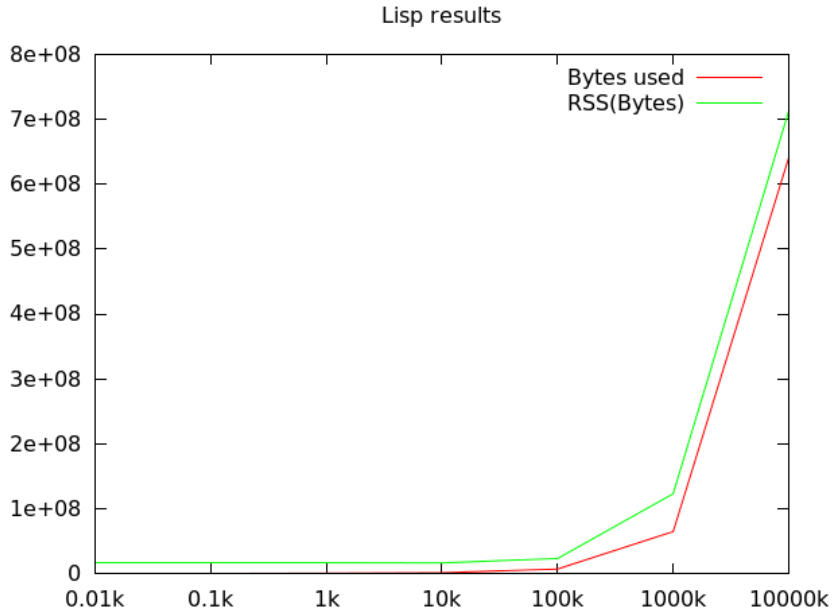

Figure 3: Lisp program, measure of bytes consumption and memory allocation (RSS)

between the performance of the Java application towards the Common Lisp one is smaller as shown in table 4 and figure 5.

Table 4: Comparison between the measured RSS

List length	Java RSS	Lisp RSS
0.01k	22568	15384
0.1k	22732	15384
1k	22864	15384
10k	26080	15912
100k	26944	22584
1000k	67212	120000
10000k	432072	696112

With these results we can understand that CLOS is not a good rival to Java objects, but as a List Processing language Common Lisp goes side by side with Java first if we use linked lists with less than 100000 elements which is plausible given that the list is created based on user inputs, second if the right data structure is employed on stacks (Lisp lists) over our logic system and the space efficiency gets better as we will show in the following section. The reader will encounter the full system in the appendix section.

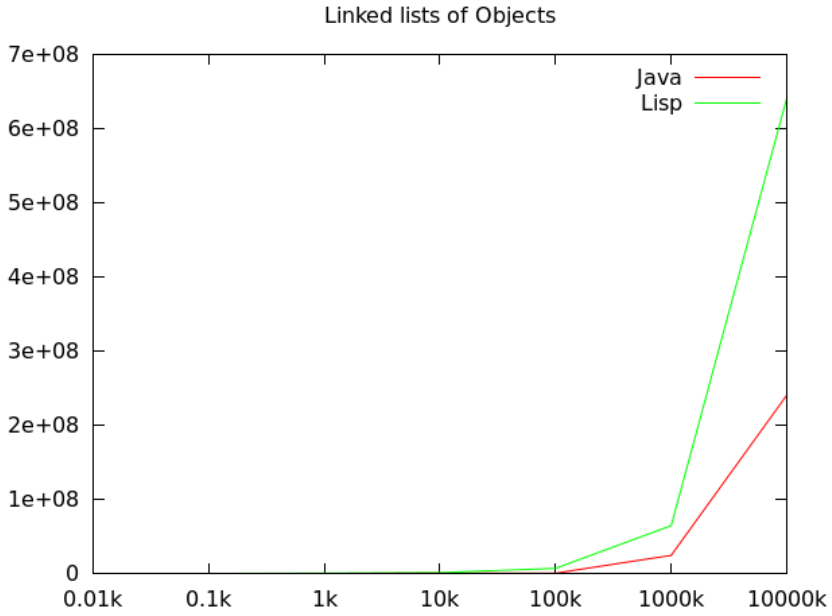

Figure 4: Comparison between Java and Lisp memory usage

7.1 Proper Lisp lists usage

Given that a *proper list* is one of the most basic data structure included in Common Lisp, the language provides built-in procedures for access and control of lists components. Besides that two lists can point to the same memory locations which improves the space efficiency of Common Lisp systems. As an example we can see the following code available in Wikipedia[10]:

```
(setf foo (list 'a 'b 'c))
(setf bar (cons 'x (cdr foo)))
```

The *bar* list is actually sharing structure with the *foo* list, as *b* and *c* are in the same memory locations for both lists, see figure 6. Programming with proper lists is another possibility out of the Common Lisp language repertory. Paul Graham in ANSI Common Lisp[6] advocates that the Lisp programmer should use lists in the initial version of a Lisp program replacing them with specialized data structures in later versions.

In order to compare which implementation is better, as the table 5 and figure 7 show, we measured the bytes consumption and allocation first in a program using proper lists then in an implementation with linked lists constructed with the CLOS (Objects) technology. The construction of the list was left to the Loop dialect through collect as shown below:

[10]See `https://en.wikipedia.org/wiki/Lisp_(programming_language)`.

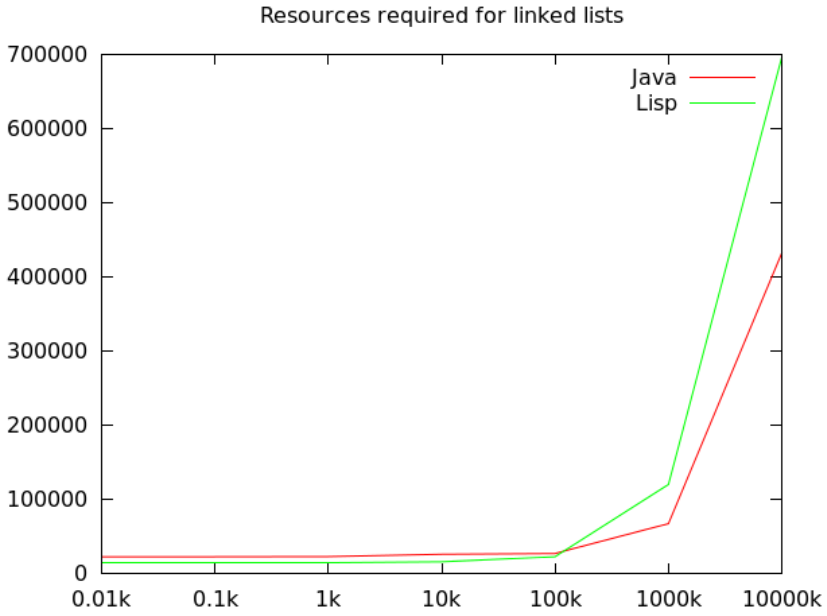

Figure 5: Comparison between Java and Lisp RSS

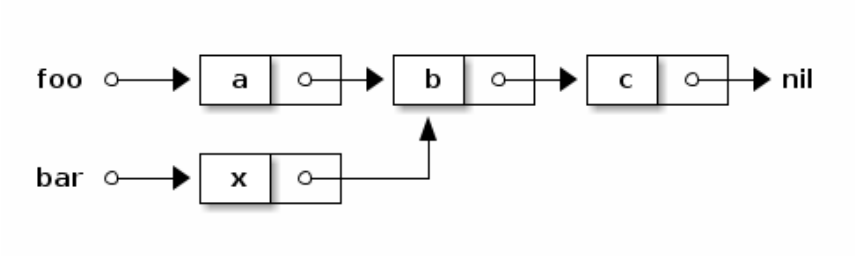

Figure 6: Sharing structures

```
(defparameter *no* nil)

(defun main()
  (setf *no* (loop for i from 1 to 100000000 collect i)))
```

We concluded that the best data structure for the first version of our logic system stack is not object lists in Java nor Common Lisp (CLOS) but proper lists. Even when not using specialized data structures Common Lisp beats the Java technology as summarized in table 6 and figure 8. The bytes usage also corroborates to the conclusion as shown in figure 9.

Despite the fact that it is already clear that Common Lisp should be the first choice to one producing a space efficient system, we have to discuss the use of a specialized data structure as it appears in the final version of our

Table 5: Comparison between Proper lists and Object lists

List length	Bytes used	RSS Proper(kB)	P(%)	Bytes used	RSS Object(kB)	O(%)
0.01K	0	7724	0	549920	15384	0.035
0.1k	0	7724	0	549920	15384	0.035
1k	20176	7724	0.003	615456	15384	0.040
10k	151248	3688	0.041	1205280	15912	0.075
100k	1593040	7304	0.218	6972448	22584	0.308
1000k	16010960	23288	0.687	64545808	120000	0.537
10000k	160113312	165424	0.967	640869472	696112	0.920

source code and is proposed by Paul Graham[6].

Table 6: Comparison between Java and Lisp program using proper lists

List length	Bytes used (Java)	RSS Java(kB)	Bytes used (Lisp)	RSS Proper(kB)
0.01k	446672	22568	0	7724
0.1k	448832	22732	0	7724
1k	470432	22864	20176	7724
10k	686432	26080	151248	3688
100k	446520	26944	1593040	7304
1000k	24425336	67212	16010960	23288
10000k	240425336	432072	160113312	165424

7.2 Specialized data structure

Following Paul Graham's advices, we proceed by showing that even though arrays are less versatile as data structure, they take less space than proper lists and can make access faster. The Common Lisp program that makes use of vectors to compare with the proper lists is the following one:

```
1  (defparameter *node* nil)
2  (defparameter *size* 10)
3
4  (defun main()
5    (setf *node* (make-array *size*))
6    (loop for i from 0 to (- *size* 1) do (setf (aref *node* i) i)))
7
8  (defun tempo() (time (main)))
9
10 (save-lisp-and-die "vtime" :executable t :toplevel #'tempo :save-runtime-options
```

The parameter `*size*`, defined on line 2, controls the length of the vector `*node*`. The specialized data structure (vector) being used reduces memory

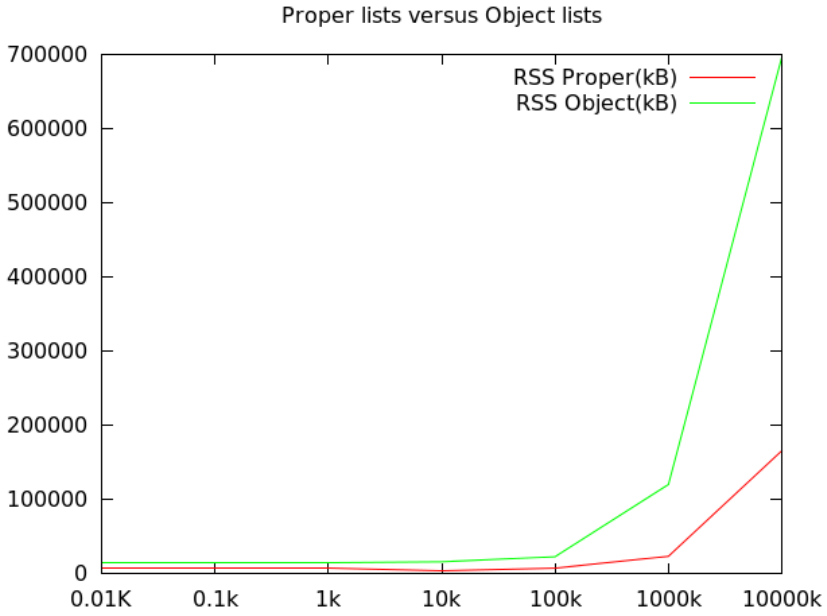

Figure 7: Comparison between Proper lists and Object lists

usage even more than proper lists, as one can see from table 7 and figure 10. The reduction become prominent after the length of the vector achieves 1000k elements.

Our verification of the advantages of using specialized data structures from Lisp over Object lists from Java can be seen in table 8 and figure 11. It is clear that if our logic system makes use of Lisp vectors it would be much more space efficient.

Table 7: Comparison between vectors and proper lists

List length	Bytes used	RSS Vectors(kB)	V(%)	Bytes used	RSS Proper(kB)	P(%)
0.01K	0	7728	0	0	7724	0
0.1k	0	7728	0	0	7724	0
1k	0	7728	0	20176	7724	0.003
10k	11536	7728	0.001	151248	3688	0.041
100k	800016	8520	0.093	1593040	7304	0.218
1000k	8000016	15452	0.517	16010960	23288	0.687
10000k	80024400	87452	0.915	160113312	165424	0.967

One could claim that instead of Java Objects we should try a Java list, so there is room for another comparison, now the better-performing implementation of Lists in Java called ArrayList. The Java class used was the following one, it creates an ArrayList of Integers, the initial capacity of the

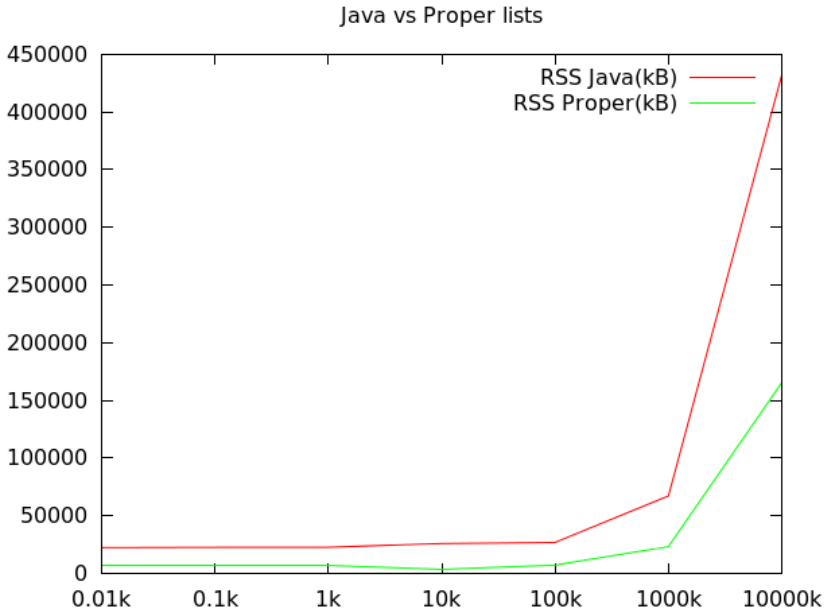

Figure 8: Comparison between Java and Lisp program using proper lists (RSS)

list is specified when the constructor receives size:

```
import java.util.*;

public class Slista {
    public static void main(String[] args) {
Scanner next = new Scanner(System.in);
int size = 10;

List<Integer> lista = new ArrayList<Integer>(size);
```

Table 8: Comparison between Java Objects and Lisp vectors

List length	Bytes used (Java)	RSS Java(kB)	Bytes used(Lisp)	RSS Vectors(kB)
0.01K	446672	22568	0	7728
0.1k	448832	22732	0	7728
1k	470432	22864	0	7728
10k	686432	26080	11536	7728
100k	446520	26944	800016	8520
1000k	24425336	67212	8000016	15452
10000k	240425336	432072	80024400	87452

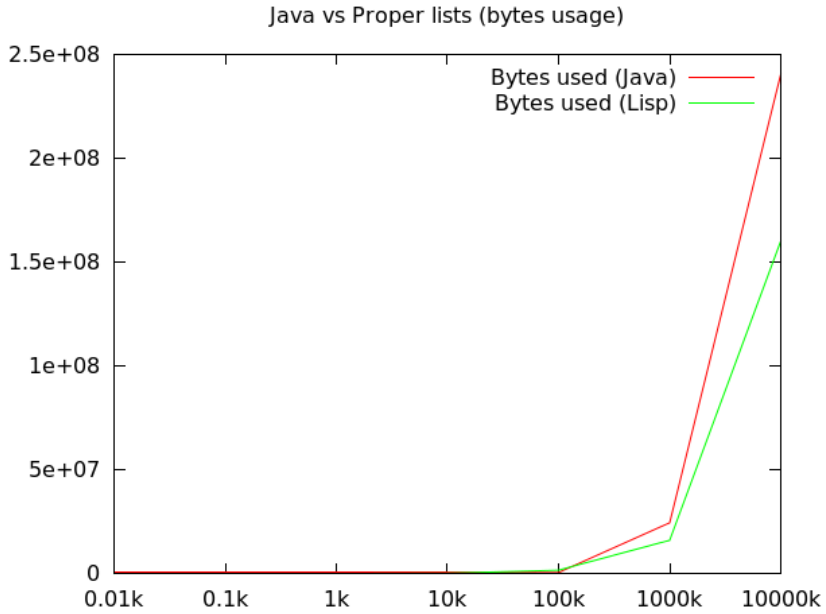

Figure 9: Comparison between Java and Lisp program using proper lists (Bytes usage)

```
for(int i = 1; i <= size; i++) {
    lista.add(i);
}

Runtime runtime = Runtime.getRuntime();
runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();

System.out.println("Used memory in bytes: " + memory);
next.nextInt();
    }
}
```

Although this program saves memory with respect to the initial Java version, it presents no improvement over the Lisp vectors version as can be seen in table 9 and figure 12.

One last try employees vectors in Java as stated in the Jvector class below, the results are summarized in table 10 and figure 13. It is now clear that no Java alternative beats Lisp, except when unprepared CLOS objects were used in Lisp.

```
import java.util.*;
```

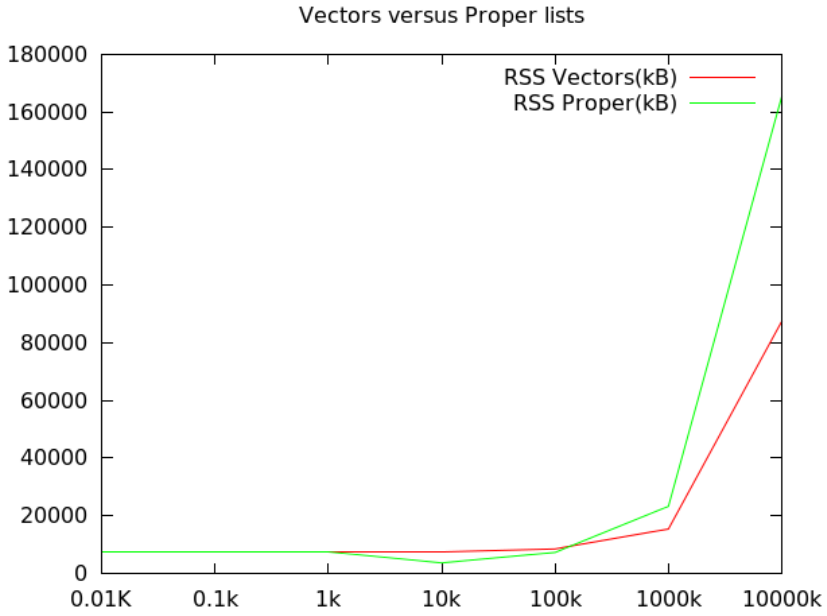

Figure 10: Comparison between vectors and proper lists

```
public class Jvector {
    public static void main(String[] args) {
Scanner next = new Scanner(System.in);
int size = 10000000;

Vector<Integer> v = new Vector<Integer>(size);

for(int i = 1; i <= size; i++) {
    v.add(i);
}
```

Table 9: Comparison between Java Lists and Lisp vectors

List length	Bytes used (Java)	RSS Java(kB)	Bytes used(Lisp)	RSS Vectors(kB)
0.01K	446536	34536	0	7728
0.1k	446896	35476	0	7728
1k	464448	36240	0	7728
10k	645824	35528	11536	7728
100k	447920	34828	800016	8520
1000k	20424976	77476	8000016	15452
10000k	200424976	384668	80024400	87452

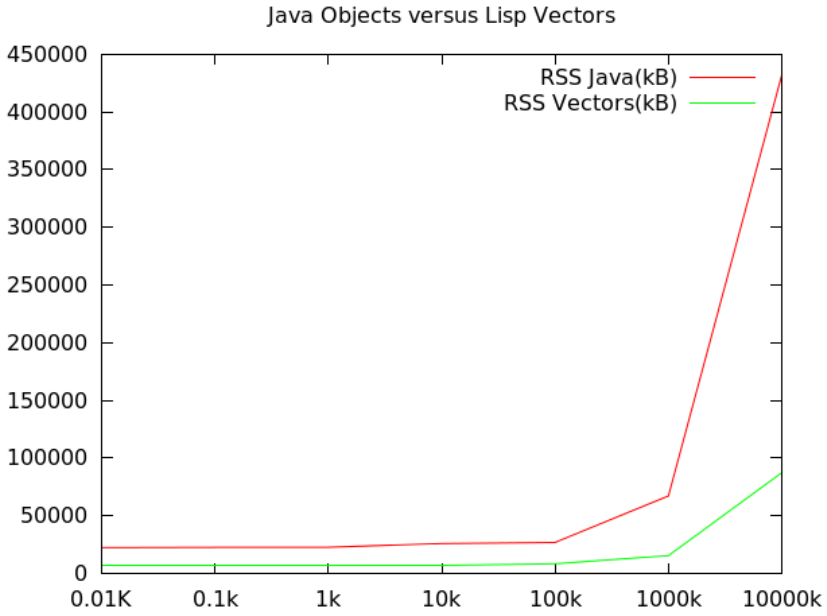

Figure 11: Comparison between Java Objects and Lisp vectors

```
Runtime runtime = Runtime.getRuntime();
runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();

System.out.println("Used memory in bytes: " + memory);
next.nextInt();
    }
}
```

Table 10: Comparison between Java Vectors and Lisp vectors

List length	Bytes used (Java)	RSS Java(kB)	Bytes used(Lisp)	RSS Vectors(kB)
0.01K	446368	35372	0	7728
0.1k	446728	36148	0	7728
1k	464296	37532	0	7728
10k	645568	34976	11536	7728
100k	447640	37132	800016	8520
1000k	20424720	76768	8000016	15452
10000k	200424720	387492	80024400	87452

Hence, our logical system uses one-dimensional arrays, also called *vectors*, one can see on the source code that they were created by calling `make-array`

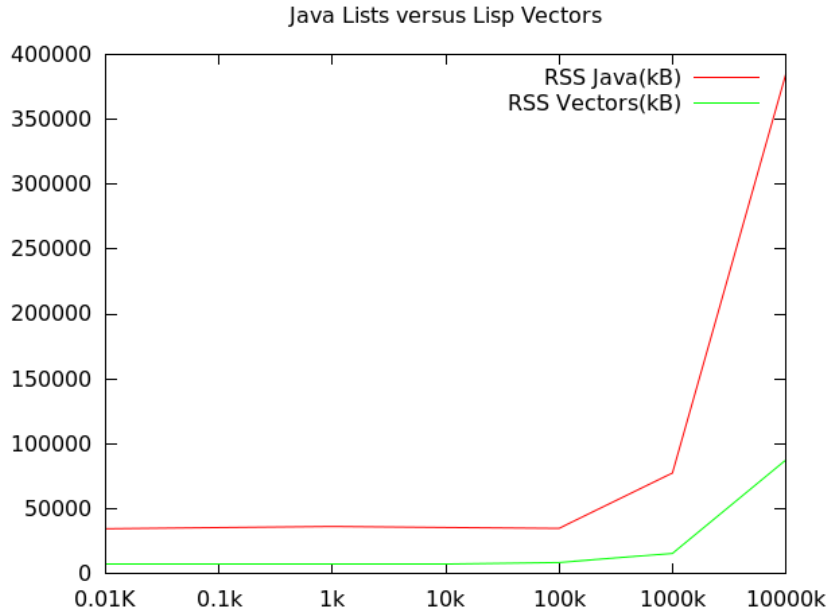

Figure 12: Comparison between Java Lists and Lisp vectors

as stated in the next code snippets:

```
 1    (defvar Memory
 2      (make-array +LocalStackOverflow+
 3   :initial-element 0 ))
 4
 5    (defvar trailMem
 6      (make-array +TrailOverflow+
 7   :initial-element 0
 8   :element-type 'fixnum))
 9
10    (defvar xArgs
11      (make-array 50
12   :initial-element 0))
13
14  (defun exec(str)
15    (setf *keep-going* nil)
16    (in-package :mini)
17    (let ((oldin *standard-input*)
18   (oldout *standard-output*)
19   (fstr (make-array '(0)
20     :element-type 'base-char
21     :fill-pointer 0
```

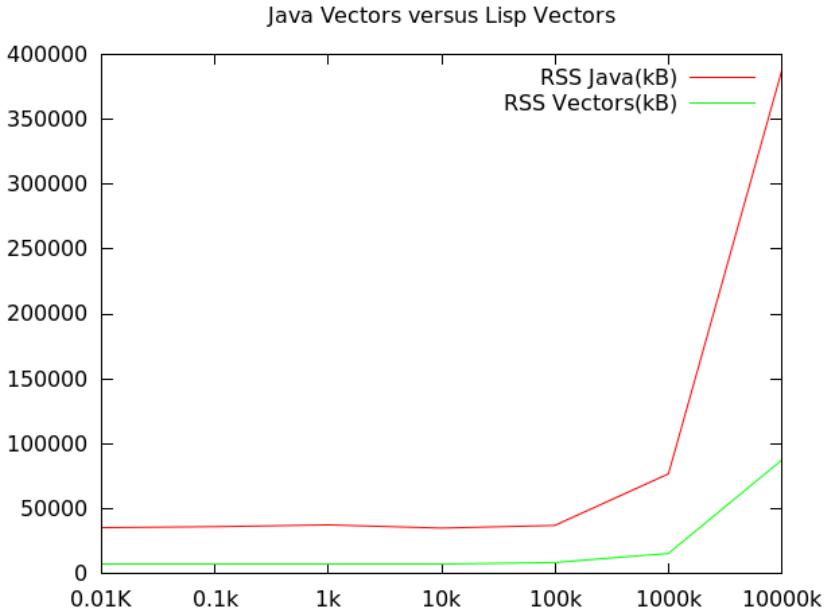

Figure 13: Comparison between Java Vectors and Lisp vectors

```
22      :adjustable t)))
23       (with-input-from-string (s (format nil "~a " str))
24         (setf *standard-input* s)
25         (with-output-to-string
26       (z fstr)
27     (setf *standard-output* z)
28     (handler-case (lisploop (read_code_tail))
29       (error(e) (format t "toplevel error: ~a~%" e)  ))))
30        (setf *standard-input* oldin
31       *standard-output* oldout)
32        (in-package :cl-user)
33        fstr))
```

The first argument of `make-array` is called *dimensions* and it is a list of integers that defines the dimensions of the array, lines 2, 6, and 11 are defining one-dimensional arrays (vectors) since `+LocalStackOverflow+`, `+TrailOverflow+` and 50 are numbers. Although line 19 provides an one-element list, not a number, as the dimension, it is defining a vector too.

Our logic system is written in Common Lisp, taking advantage of its syntax and semantics, the Warren Abstract Machine (WAM) is the cornerstone for logic or Prolog systems. In order to make clearly comprehensible how the Common Lisp technology shaped this logic system we present the next section.

8 The logic system

The Prolog language is a theorem prover for predicate logic, in it the predicate's name and the formal parameters (number and structure) receive a meaning in the same manner as other atomic propositions in the form of internal structures.

A Prolog compiler is a process that receives a collection of Prolog clauses and produces a description of the structure from this clauses in an executable way. Over the validation process, the Prolog interpreter needs to travel through complex data structures.

The representation of the Prolog term is a symbol containing a value and a *tag*. The tag discriminates the type of the term. The main types are references, structures, lists and constants. A not ajusted variable is represented by a reference to itself.

8.1 How a Prolog system works

The Prolog programming can be summarized as follows:

- A Prolog program is a set of procedures.
- A Prolog procedure is a set of clauses.

Each clause has the following form:

```
P :- Q1,Q2,...,Qn.
```

The previous `P` form is true if `Q1` is true, `Q2` is true and so on until `Qn` is true. If `n` is equal to 0 the clause is written as `P` only. In this case `P` is true. In the following form:

```
P :- Q1,Q2,Q3.
```

P is the head.

:- is the neck.

Q1,Q2,Q3 form the body.

. is the foot.

In Prolog, the predicate definition is represented by a process and the declarative semantics goes straight to the point:

```
masculino(willian).
masculino(george).
feminino(kate).
feminino(charlotte).
```

Conditions can be defined with predicates in the following manner:

```
filho(PAI,FILHO) :- genitor(PAI,FILHO), masculino(PAI).
avo(AVO,NETO) :- genitor(AVO,PAI), genitor(PAI,NETO).
```

The terms `AVO`, `PAI`, `FILHO`, `NETO` are variables. The scope of a variable is the clause where it is used.

A way to visualize the running of Prolog procedures is through a search tree. The search and the pattern matching limits the Prolog performance. In order to execute a process the Prolog system must find through the process candidates verifying the matching, doing backtracking when it does not find a match through the path and passing information through deunification or pattern matching.

The Prolog procedures may contain parameters that works like input and output pipes depending on the call type. Each provided argument by a procedure call is a description.

At the time of the procedure call, the call term arguments are compared with the parameter terms from the procedure called. The pattern matching process (called unification) tests if two terms match.

Prolog procedures may contain parameters on terms. A term may be a constant, a structure or variable. Constants are atomic objects. Structures consist on a functor applied to the terms as arguments, `pai(willian,george)`, the variables denote arbitrary objects, the convention is to use variable names starting with uppercase letters. In the following code:

```
pai(william,george)
```

The functor is `pai` and the arity is equal to 2 `(william,george)`. The compilation behave with Prolog code being translated into a set of executable actions.

8.2 WAM adaptations

Now we proceed with the description of the idiosyncrasies that makes our logic system based on the WAM virtual machine, the reader should be aware that:

1. We choose a implementation language (Common Lisp).
2. The system makes use of specialized data structures (Lisp vectors).
3. Our logic system can make calls to the host language.

The *WAM* system consists of a memory architecture and instruction set tailored to Prolog. It was developed by David H. D. Warren[16] and described in his technical report. This technical report was written for an expert reader and contains only a "bare bones" definition thus the need for the Hassan's work.

The frameword that supports our system is the Warren's Abstract Machine (*WAM*) and Hassan Aït-Kaci in [2] did an excellent work transforming the *WAM* system into something more palatable to the rest of us. Our work was to bring the WAM to the LISP world.

8.2.1 Memory vector and registers

The Memory vector implements both the local and heap (copy) stack. A fixed-size vector was choosen over a resizable one, given that they do not abstract the actual storage, and we can use the more general function **MAKE-ARRAY** defining the size of the vector.

```
(defconstant +LocalStackOverflow+ (the fixnum 1500000))
```

Note the lack of the keyword `:adjustable` thus this vector will not be resized as needed consuming less memory resources.

```
(defvar Memory (make-array +LocalStackOverflow+
:initial-element 0 ))
```

The Memory vector will not expand if one tries to push an element when its length `+LocalStackOverflow+` is already filled. Instead, the following throw clause will unwind the stack and cause the matching catch to return immediately.

```
(defmacro push_Environment (n)
  `(let ((top (+ *LocalPointer* 3 ,n)))
     (if (>= top +LocalStackOverflow+)
 (throw 'debord (print "Local Stack Overflow")))
```

```
    (vset Memory (+ *LocalPointer* 2) Cut_pt)
    (dotimes (i ,n top) (vset Memory (decf top) (cons 'V top)))))

(defun readProvePrintLoop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (forward))
  (cond ( *keep-going*
 (handler-case (readProvePrintLoop (read_prompt))
   (error(e) (format t "Error: ~a" e) (readProvePrintLoop (read_prompt)))))
(t (format t "Bye!~%")
   (setf *keep-going* t))))
```

As a remainder to the kind of association that exists between a catch and throw in the Common Lisp language one can note that the following catch is tied to `push_Environment` or `readProvePrintLoop` and this will be determined at runtime conforming the `lispforward` call.

```
(defun lisploop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (lispforward)))

(defun readProvePrintLoop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (forward))
  (cond ( *keep-going*
 (handler-case (readProvePrintLoop (read_prompt))
   (error(e) (format t "Error: ~a" e) (readProvePrintLoop (read_prompt)))))
(t (format t "Bye!~%")
   (setf *keep-going* t))))
```

Being a Lisp vector, Memory can accept any kind of data, and we retrieve its contents with `(svref Memory i)`, the index `i` must be non-negative and less than the length of the vector.

```
(defmacro CL (b) `(svref Memory ,b))
(defmacro CP (b) `(svref Memory (1+ ,b)))
(defmacro Cut (b) `(svref Memory (+ ,b 2)))

(defmacro TR (b) `(svref Memory (1- ,b)))
(defmacro BP (b) `(svref Memory (- ,b 2)))
(defmacro BL (b) `(svref Memory (- ,b 3)))
(defmacro BG (b) `(svref Memory (- ,b 4)))
(defmacro BCL (b) `(svref Memory (- ,b 5)))
(defmacro BCP (b) `(svref Memory (- ,b 6)))
(defmacro AChoice (b) `(svref Memory (- ,b 7)))

(defun load_A2 ()
  (let ((deb (- *LocalPointer* (size_C *LocalPointer*))))
    (dotimes (i (AChoice *LocalPointer*) (vset xArgs 0 i))
      (declare (fixnum i))
(vset xArgs (+ i 1)
      (svref Memory (+ deb i))))))
```

The command `(setf (svref Memory i) Val)` should be applied by one trying to store data on Memory, and we created the macro **VSET** to abstract away the details and reduce typing. While **svref** returns a place, **setf** change its value. **VSET** was defined on line 1 from the following code and used on lines 4, 5, 11, 12, 16, 18, 24, 29 and 36.

```
 1  (defmacro vset (v i x) `(setf (svref ,v ,i) ,x))
 2
 3  (defmacro push_continuation ()
 4    `(progn (vset Memory *LocalPointer* CL)
 5     (vset Memory (1+ *LocalPointer*) CP)))
 6
 7  (defmacro push_Environment (n)
 8    `(let ((top (+ *LocalPointer* 3 ,n)))
 9       (if (>= top +LocalStackOverflow+)
10    (throw 'debord (print "Local Stack Overflow")))
11       (vset Memory (+ *LocalPointer* 2) Cut_pt)
12       (dotimes (i ,n top) (vset Memory (decf top) (cons 'V top)))))
13
14  (defun save_args ()
15    (dotimes (i (svref xArgs 0)
```

```
16          (vset Memory (incf *LocalPointer* i) i))
17       (declare (fixnum i))
18       (vset Memory (+ *LocalPointer* i)
19      (svref xArgs (+ i 1)))))
20
21  (defmacro push_Global (x)
22    `(if (>= (incf *GPointer*) BottomL)
23         (throw 'debord (print "Heap Overflow"))
24       (vset Memory *GPointer* ,x)))
25
26  (defmacro poptrail (top)
27    `(do () ((= TR ,top))
28       (let ((v (aref trailMem (decf TR)) ))
29         (vset Memory v (cons 'V v)))))
30
31  (defmacro bind (x te)
32    `(progn
33       (if (or (and (> ,x BottomL) (< ,x BL))
34        (<= ,x BG))
35    (pushtrail ,x ,te))
36       (vset Memory ,x ,te)))
37
38  (defun ult (m)
39    (declare (fixnum m))
40    (do* ( (n m (cdr te))  (te (svref Memory n) (svref Memory n)))
41         ( (not (and (var? te) (/= (cdr te) n))) te)
42        (declare (fixnum n))))
```

The Memory vector will work as the global block of storage needed to implement the addressable heap used as a stack for building terms. The top of this global stack receive the term parts as they are incrementally constructed. This vector will store variables and structures.

There is no need to pointer arithmetic in LISP. The concept of **places** in Common Lisp defines location in memory, those locations are handled directly when a function applies **setf** to work with places not values. The **aref** built-in (accessor) function can be used to provide the place where setf will work.

In Lisp, a vector containing only fixnums is vastly more efficient than a general vector, as the specialization of the vector restrict the type of elements that may be received. Therefore, one has defined the trail stack trailMem as a vector of fixnums. The local stack and the copy stack share an inefficient general vector Memory.

The trail stack was implemented as a separated vector. Its type was specified with the `element-type` keyword. This specification improves the

efficiency of the `trailMem` vector regulating the memory use.

```
(defconstant +TrailOverflow+ 1000000)
(defvar trailMem
  (make-array +TrailOverflow+ :initial-element 0
      :element-type 'fixnum))
```

The `trailMem` receives values less than the length `+TrailOverflow+` and starting from 0.

```
(defconstant BottomTR (the fixnum 0))
```

The `setf` operation on this vector gave birth to another Lisp macro:

```
(defmacro trailset(v i x) `(setf (aref ,v ,i) ,x))
```

This **trailset** macro is identical to the `vset` macro, both abstract the insertion of an element into a vector place through the use of `setf`, but as they are applied to different vectors, they exist only to improve the reading comprehension.

The global variables `TR` (line 1), `*LocalPointer*` (line 3) and `*GPointer*` (line 6) logically mark the top of the corresponding stacks, trail, local, and copy stack.

```
1  (defvar TR (the fixnum 0))
2
3  (defvar *LocalPointer*
4    (the fixnum 0))
5
6  (defvar *GPointer*
7    (the fixnum 0))
```

Let `g[i],g[i+1],...,g[n]` be a sequence of sibling goals. Let `cl` be the clause whose head unifies with `g[i]`. In this case, one has that:

CL mother block of this sequence of goals.

CP current continuation: `g[i],g[i+1],...,g[n]`.

Environment part of the action block allocation in the local stack:

BL field previous choice in the local stack

BP field set of clauses in the remaining choices

TR field top of the trail

BG field top of the copy stack before allocation of the top stack. In the pair (BL, BG), BG designates the top of the copy stack associated with the previous choice BL.

```
1  (defvar CP (the fixnum 0))
2  (defvar CL (the fixnum 0))
3  (defvar Cut_pt (the fixnum 0))
4  (defvar BL (the fixnum 0))
5  (defvar BG (the fixnum 0))
6  (defvar PC (the fixnum 0))
7  (defvar PCE (the fixnum 0))
8  (defvar Duboulot)
```

An original `fp(a,b)` functor is stored as `((|fp|) a b)` a so called proper list. A copy of this functor is stored as `((|fp| . t) a b)`, a proper list that contains a dotted list as its first argument given that the `cdr` of this list is not another list nor the empty list (`nil`) but just the atom t.

```
1  (defmacro functorCopy (des largs)
2    `(cons (cons (car ,des) t) ,largs))
```

Let be the `fp(a,b)` functor. We had chosen to represent it as `((|fp|) a b)` for an original functor, and as `((|fp| . t) a b)` for a copy. The description of the functor is given by `(|fp|)` in the case of the original, or `(|fp| . t)` the case of the copy. Anyway `car` will return the expected result.

```
1  (defmacro functorDescription (te) `(car ,te))
2  (defmacro functor (description) `(car ,description))
```

Let us have an `largs` macro for retrieving predicate args, and `fargs` for functor args. In the present representation, these macros are defined identically. However, one may choose a different representation for functors. For example, it may be a good idea to represent functors as a CLOS class, but space efficiency must be verified.

```
1  (defmacro largs (x) `(cdr ,x))
2  (defmacro fargs (x) `(cdr ,x))
```

A `var` consists of a well formed lists of numbers:

```
1  (defmacro var? (x)
2     `(and (consp ,x)
3    (numberp (cdr ,x))))
```

List is a two arg functor with descritors given by (\.) for originals, and (\. . t) for copies.

```
1  (defmacro list? (x)
2    `(eq (functor (functorDescription ,x)) '\.))
```

The next section deals with the coupling of Prolog code and Lisp code.

8.2.2 Prolog syntax into LISP

Our system has to define a non-lisp syntax. The Prolog syntax must be processed by a different reader other than the lisp reader. By controlling the lisp reader behaviour with the read macro we are extending the reader. This is done to turn our *Prolog* code into lisp code.

```
 1  (set-macro-character
 2   #\$
 3   #'(lambda (stream char)
 4       (declare (ignorable char))
 5       (let* ( (*standard-input* stream) (c (read_code_cl)))
 6         (add_cl (if (symbolp (car c)) (car c)
 7      (pred (head c ))) c 'def)
 8         (if (largs (head c))
 9      (let ((b (nature (car (largs (head c))))))
10        (if (eq b 'def)
11    (mapc
12     #' (lambda (x) (add_cl (pred (head c)) c x))
13        '(atom empty list fonct))
14    (add_cl (pred (head c)) c b)))))
15       (values)))
```

We somehow have to convert the Prolog predicates' id into LISP id. For instance, if `xs = (|fib| |n| |?f|)` is read it needs to be translated to `(FIB N ?F)`. This is the job of `read_clause` along with `fix`.

```
 1  (defun fix(xs)
 2    (mapcar #'mkLispID xs ))
 3
 4  (defun read_clause (ch &optional (stream *standard-input*))
 5    (let ((tete (read_pred ch stream))
 6   (neck (rchnsep stream)))
 7      (if (char= neck #\.)
 8   (cond (  (modedeclarationp tete)
 9          (list 'mdef (fix  tete  )))
10         (t (list tete)))
```

```
11   (let ((nneck (read-char stream)))
12     (cond (  (and (char= neck #\:) (char= nneck #\-)
13   (get (mkLispId (car tete)) 'mod) )
14    (let ((tail
15   (xread_tail (rchnsep stream) stream)))
16      (cons 'def  (cons (fix tete) tail))))
17   (t (cons tete
18    (read_tail (rchnsep stream)
19       stream))))  )) ))
```

Lisp uses uppercase ids while Prolog uses lowercase. An expression like `(mkLispID '|fib|)` translates `|fib|` to `FIB`, where `|fib|` may be replaced by any Prolog id.

```
1  (defun mkLispID(n)
2    (intern (string-upcase
3      (symbol-name n))))
```

8.2.3 The main function

We created a package **:mini** that can access all the Common Lisp facilities defined in another package, **:cl**, in order to start the logic system, one must use the `wam` function.

```
(defpackage :mini (:use :cl)
    (:export :wam :exec))

(defun wam ()
  (setf *keep-going* t)
  (banner)
  (in-package :mini)
  (readProvePrintLoop (read_prompt))
  (in-package :cl-user))
```

The main function is `readProvePrintLoop`, it works like the Common Lisp REPL adapted to Prolog, after setting up the environment, it loops through reading Prolog code from the standard input stream, processing its tokens, as indicated in the Warren's work, proving those statements and printing the results.

```
(defun readProvePrintLoop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
```

```
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (forward))
  (cond ( *keep-going*
 (handler-case (readProvePrintLoop (read_prompt))
   (error(e) (format t "Error: ~a" e) (readProvePrintLoop (read_prompt)))))
(t (format t "Bye!~%")
   (setf *keep-going* t))))
```

The user can interact with the logic system stopping the process that shows all the matched options or to continue to see the rest of them after backtrack answering the `yes/no` function.

```
(defun lispforward ()
  (do () ((null Duboulot))
      (cond ((null CP) (answer))
    ( (load_PC)
     (cond
       ((partial? PC)
(funcall (car PC)))
       ((user? PC)
(let ((d (def_of PC)))
  (if d (pr2 d) (backtrack))))
       ((builtin? PC)
(if (eq (apply (car PC) (cdr PC)) 'fail)
    (backtrack)
  (cont_eval)))
       ((backtrack)))))))

(defun yes/no()
  (princ "More : ")
  (finish-output)
  (member (rchnsep) '(#\o #\y #\s #\d #\j #\;)))

(defun answer ()
  (printvar)
  (if (zerop BL)
      (setq Duboulot nil)
    (if (yes/no)
(backtrack)
      (setq Duboulot nil))))

(defun backtrack ()
  (if (zerop BL)
```

```
      (setq Duboulot nil)
      (progn (setq *LocalPointer* BL *GPointer* BG Cut_pt (BL BL)
   CP (BCP *LocalPointer*) CL (BCL *LocalPointer*))
   (load_A2)
   (poptrail (TR BL))
   (pr_choice (BP *LocalPointer*)))))
```

One of the most important things of unification is the specialization of the unification algorithm in order to handle special cases. The non-deterministic Prolog implementation is a good example.

```
pai(a, b).
pai(b, c).
pai(c, d).

mae(d, g).

avo(P, X, Y) :- pai(X, Z), call(P, Z, Y).

equ(X,X).
```

A typical interaction with the Prolog system would be like the following one, the user creates a file called pai, like the previous one, where it defines some relationships between names, make a query and receives the system response:

```
CL-USER(3): (mini:wam)

mini-Wam

| ?- consult(pai)
.
Yes
no More

| ?- avo(pai, X, Y).
call=(pai (V . 3) (V . 2))
X = a
Y = c
More : ;
call=(pai (V . 3) (V . 2))
X = b
Y = d
More : ;
```

```
call=(pai (V . 3) (V . 2))
no More

| ?- avo(mae, X, Y).
call=(mae (V . 3) (V . 2))
call=(mae (V . 3) (V . 2))
call=(mae (V . 3) (V . 2))
X = c
Y = g
no More

| ?- halt.
Yes
no More
Bye!
```

The `(mini:wam)` function starts the loaded system, the first thing it does is prompt the user (`| ?-`). In order to receive other possible answer the user needs to press `;` until he receives `no More`. To stop the system `halt.` does the job. And the Common Lisp implementation comes back again.

9 Conclusion

The computer became pervasive, part of the life and almost of the body of the human beings. Nowadays, it is used like an extension of the person and not as a mere tool stuck in an office desk, they are smaller and more powerful. In [11] it was reported that nearly half (47%) of respondents couldn't last more than one day without their phone.

Along with the increase in use, new applications emerged. Banks are operating through apps, even check deposit can be made with the smarthphone cameras. Games are being played with the real world as a background. The smart TVs are listening to the users and responding to voice commands. All this and other applications needs processing power, some even needs to solve logic problems.

Our Prolog system is a good option, a space efficient one given it is three or four times more efficient than the chosen parameter. The system is, in short, a domain-specific language written on top of the Common Lisp language, this provides an interface to the Common Lisp implementation and gives the user the option to write Lisp code when the algorithm in it is more evident or maintainable and write Prolog otherwise to express logic. Thus the user would in no doubt implement the Ninety-Nine Lisp Problems[11] in Common Lisp, while the eight queens problem[12] would be implemented in the Prolog system like the following one:

```
queen(N,R) :- range(1,N,Ns), queens(Ns,[],R).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :- plus(M1, M,1), range(M1,N,Ns).

queens([],R,R).
queens(Unplaced,Safe,R) :- del(Q,Unplaced,U1),
     \+ att(Q,Safe),
     queens(U1,[Q|Safe],R).

att(Q, Safe) :- attack(Q, 1, Safe).

attack(X,N,[Y|_]) :-  plus(X, Y,N), !.
attack(X,N,[Y|_]) :-  plus(Y, X,N), !.
attack(X,N,[_|Ys]) :- plus(N1, N,1), attack(X,N1,Ys).

% ?- cputime(X1), tqueen(8, Q), cputime(X2), minus(Time, X2, X1).
tqueen(X, R) :- queen(X, R), fail.
```

[11] See http://www.ic.unicamp.br/~meidanis/courses/mc336/2006s2/funcional/L-99_Ninety-Nine_Lisp_Problems.html

[12] See http://www.javaist.com/blog/2008/11/06/eight-queens-problem-in-prolog/

```
tqueen(X, R) :- queen(X, R), !.
```

All the found common constructions became new operators thanks to defmacro available in Common Lisp. Even those important only because of the program logic like `largs` and `fargs` took advantage of that macro given that this enriches the code and makes it more readable and maintainable. The reader must be aware that the language used in Lisp macros is Lisp and the way it works has nothing to do with C macros where the language does not works like C.

With the Prolog system available in the appendix A, it is possible to do logic programming with the benefit of the Common Lisp environment, for instance, now the user may adapt a Computer Algebra System like Maxima, written in Common Lisp, to work with its logic problem.

The direct research that benefits from this resulting system is the application of chunk grammars, like those described by Michael Covington in his book [4], in Stenography. Chunk grammars work with the principle that a partial solution is better than a complete failure. In fact, it seems that human beings work with chunk grammars. People do what they can with what they got.

References

[1] Oracle and/or its affiliates. Java™ platform, standard edition 6 api specification. `http://docs.oracle.com/javase/6/docs/api/overview-summary.html`. Accessed: 2016-07-29.

[2] Hassan Aït-Kaci. *Warren's abstract machine : a tutorial reconstruction*. Logic programming. Cambridge, Mass. MIT Press, 1991.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[4] Michael A. Covington. *Natural Language Processing for PROLOG Programmers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1993.

[5] Elizabeth Gibney. Google ai algorithm masters ancient game of go. *Nature*, 529:445–446, 2016.

[6] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[7] Daniel Holden. *Build Your Own Lisp*. CreateSpace Independent Publishing Platform, 1 edition, 2014.

[8] Doug Hoyte. *Let Over Lambda*. Doug Hoyte, 1 edition, 2010.

[9] Igor L. Markov. Limits on fundamental limits to computation. *Nature*, 512:147–154, 2014.

[10] Andrew Myers. Biological transistor enables computing within living cells, study says. `https://med.stanford.edu/news/all-news/2013/03/biological-transistor-enables-computing-within-living-cells-study-says.html`. Accessed: 2016-03-26.

[11] Bank of America. Trends in consumer mobility report. Technical report, 2014.

[12] Peter Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 1st edition, 2012.

[13] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient java. *SIGPLAN Not.*, 36(5):104–113, May 2001.

[14] John Timmer. Are processors pushing up against the limits of physics? `http://arstechnica.com/science/2014/08/are-processors-pushing-up-against-the-limits-of-physics/`. Accessed: 2016-03-26.

[15] Lars Vogel. Java performance - memory and runtime analysis - tutorial. http://www.vogella.com/tutorials/JavaPerformance/article.html. Accessed: 2016-07-18.

[16] David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

A The logic system source code

```
(defpackage :mini (:use :cl)
    (:export :wam :exec))
(in-package :mini)

(declaim  (optimize (speed 3) (debug 0) (safety 0) (space 0))
  (ftype (function (*) t) addit)
  (ftype (function (* * *) t) add_cl)
  (ftype (function (* *) t) bindte)
  (ftype (function (*) t) reduce-infix)
  (ftype (function (* * *) t) reduce-matching-op)
  (ftype (function () t) read_prompt)
  (ftype (function () t) banner)
  (ftype (function (&optional *) t) rchnsep)
  (ftype (function (&optional *) t) read_code_tail)
  (ftype (function (*) t) readProvePrintLoop)
  (ftype (function (* &optional *) t) read_args)
  (ftype (function (* &optional *) t) xread_args)
  (ftype (function (* &optional *) t) read_term)
  (ftype (function (* &optional *) t) xread_term)
  (ftype (function (* &optional *) t) read_pred)
  (ftype (function (* *) t) ultimate)
  (ftype (function (*) t) vvarp)
  (ftype (function (* *) t) cop)
  (ftype (function (*) t) pr_det)
  (ftype (function (*) t) genvar)
  (ftype (function (*) t) ult)
  (ftype (function () t) load_pc)
  (ftype (function () t) cont_eval)
  (ftype (function () t) load_A2)
  (ftype (function (*) t) shallow_backtrack)
  (ftype (function () t) backtrack)
  (ftype (function (*) t) pr_choice)
  (ftype (function (*) t) pr)
  (ftype (function (*) t) pr2)
  (ftype (function (*) t) lisploop)
  (ftype (function (* *) t) unif)
  (ftype (function (*) t) impl)
  (ftype (function (*) t) expl)
  (ftype (function () t) printvar)
  (ftype (function (* *) t) writesl)
  (ftype (function (* *) t) writesf)
  (ftype (function (*) t) write1)
```

```
  (ftype (function () t) read_terme)
  (ftype (function (*) t) write1))

(defparameter *infix-ops*
  '((([ list match ]) ({ elts match }) (|(| nil match |)|))
    ((not not unary) (~ |neg| unary) )
    ((*) (/))
    ((+) (-))
    ((<) (>) (<=) (>=) (=) (/=))
    ((and) (& and) (^ and))
    ((or) (\| or))
    ((=>))
    ((<=>))
    ((|,|)))
  "A list of lists of operators, highest precedence first.")

(defun length>1(x) (> (length x) 1))

(defun replace-subseq (sequence start length new)
  (nconc (subseq sequence 0 start) (list new)
 (subseq sequence (+ start length))))

(defun op-token (op) (first op))
(defun op-name (op) (or (second op) (first op)))
(defun op-type (op) (or (third op) 'BINARY))
(defun op-match (op) (fourth op))
(defun length=1(s) (= (length s) 1))

(defun ->prefix (infix)
  "Convert an infix expression to prefix."
  (loop
     (when (not (length>1 infix))
       (RETURN (first infix)))
     (setf infix (reduce-infix infix))))

(defun change_infix (s)
  (cond ((null s) s)
((null (cdr s)) s)
((and (eq (car s) '|(|)
      (eq (cadr s) '-))
(cons '|(| (cons '\~ (change_infix (cddr s)))))
((and (eq (car s) '*)
      (eq (cadr s) '-))
 (cons '* (cons '\~ (change_infix (cddr s))))))
```

```
((and (eq (car s) '/)
      (eq (cadr s) '-))
 (cons '/ (cons '\~ (change_infix (cddr s)))))
(t (cons (car s) (change_infix (cdr s ))))  ))


(defun prefx(s) (list (->prefix
       (change_infix
(if (and (consp s)
(eq (car s) '-))
   (cons '~ (cdr s))
   s))) ))

(defun reduce-infix (infix)
  "Find and reduce the highest-precedence operator."
  (dolist (ops *infix-ops*
   (error "Bad syntax for infix expression: ~S" infix))
    (let* ((pos (position-if #'(lambda (i) (assoc i ops))
     infix
     :from-end (eq (op-type (first ops))
   'MATCH)))
   (op (when pos (assoc (elt infix pos) ops))))
      (when pos
(RETURN
  (case (op-type op)
    (MATCH (reduce-matching-op op pos infix))
    (UNARY (replace-subseq infix pos 2
   (list (cons (op-name op) 1)
 (elt infix (+ pos 1)))))
    (BINARY (replace-subseq infix (- pos 1) 3
    (list (cons (op-name op) 2)
  (elt infix (- pos 1))
  (elt infix (+ pos 1)))))))))))
(defun op(s) (car s))
(defun arg1(s) (cadr s))
(defun arg2(s) (caddr s))

(defun remove-commas (exp)
  "Convert (|,| a b) to (a b)."
  (cond ( (eq (op exp) '|,|)
 (nconc (remove-commas (arg1 exp) )
(remove-commas (arg2 exp))))
(t (list exp))))
```

```
(defun handle-quantifiers(x) x)
(defun function-symbol? (x)
  (and (symbolp x) (not (member x '(and or not ||)))
       (alphanumericp (char (string x) 0))))

(defun reduce-matching-op (op pos infix)
  "Find the matching op (paren or bracket) and reduce."
  (let* ((end (position (op-match op) infix :start pos))
 (len (+ 1 (- end pos)))
 (inside-parens (remove-commas
 (->prefix (subseq infix (+ pos 1) end)))))
    (cond ((not (eq (op-name op) '|(|)) ;; handle {a,b} or [a,b]
   (replace-subseq infix pos len
   (cons (op-name op) inside-parens))) ; {set}
  ((and (> pos 0)  ;; handle f(a,b)
(function-symbol? (elt infix (- pos 1))))
   (handle-quantifiers
    (replace-subseq infix (- pos 1) (+ len 1)
    (cons (elt infix (- pos 1))
  inside-parens))))
  (t (replace-subseq infix pos len
     (first inside-parens))))))



;; mini-Wam
;; boot
;;

(defparameter *keep-going* nil)

(defun exec(str)
  (setf *keep-going* nil)
  (in-package :mini)
  (let ( (oldin *standard-input*)
(oldout *standard-output*)
 (fstr(make-array '(0) :element-type 'base-char
  :fill-pointer 0 :adjustable t)))
    (with-input-from-string (s (format nil "~a " str))
      (setf *standard-input* s)
      (with-output-to-string
  (z fstr)
(setf *standard-output* z)
(handler-case (lisploop (read_code_tail))
```

```
  (error(e) (format t "toplevel error: ~a~%" e)  ))))
    (setf *standard-input* oldin
  *standard-output* oldout)
    (in-package :cl-user)
    fstr))

(defun wam ()
  (setf *keep-going* t)
  (banner)
  (in-package :mini)
  (readProvePrintLoop (read_prompt))
  (in-package :cl-user))

(defun read_prompt ()
  (terpri)
  (format t "| ?- ")
  (finish-output)
  (read_code_tail))

(defun banner ()
  (dotimes (i 2) (terpri))
  (format t "mini-Wam~%")
  (dotimes (i 2) (terpri)))

(defun l ()
  (format t "Back to mini-Wam top-level~%")
  (readProvePrintLoop (read_prompt)))

(defvar *lvar nil)
(set-macro-character #\% (get-macro-character #\;))

(defun rch (&optional (stream *standard-input*))
  (do ((ch (read-char stream) (read-char stream)))
      ((char/= ch #\Newline) ch)))

(defun ignore-to-eol(xch &optional (stream *standard-input*))
  (do ((ch xch (read-char stream)))
      ((char= ch #\Newline) (rchnsep stream))))

(defun commts(ch &optional (stream *standard-input*))
  (if (char= #\% ch) (ignore-to-eol (read-char  stream) stream)
      ch))

(defun rchnsep (&optional (stream *standard-input*))
```

```
  (do ((ch (rch stream) (rch stream)))
      ( (and (char/= ch #\space)
     (char/= ch #\tab)) (commts ch stream)  ) ))

(defun spcial (ch) (char= ch #\_))
(defun alphanum (ch) (or (alphanumericp ch)
 (spcial ch)))
(defun valdigit (ch) (digit-char-p ch))

(defun read_frac(ch x acc &optional (stream *standard-input*))
  (cond((digit-char-p (peek-char nil stream))
(read_frac (read-char stream)
   (/ x 10.0)
   (+ acc (* (valdigit ch) x)  )
   stream))
       (t (+ acc (* (valdigit ch) x))  )))

(defun read_fr(ch x acc &optional (stream *standard-input*))
  (declare (ignorable ch))
  (if (not (digit-char-p (peek-char nil stream)))
      (progn (unread-char  #\. stream)  0.0)
      (read_frac (read-char stream) x acc stream)))

(defun read_number (sign ch &optional (stream *standard-input*))
  (do ((v (valdigit ch) (+ (* v 10) (valdigit (read-char stream)))))
      ((not (digit-char-p (peek-char nil stream)))
       (if (char= (peek-char nil stream) #\.)
   (* sign (+ v  (read_fr (read-char stream)
  0.1 0.0 stream) ))
   (* v sign))) ))

(defun implode (lch) (intern (map 'string #'identity lch)))

(defun read_atom (ch &optional (stream *standard-input*))
  (cond  ((char= ch #\-)  '-)
 ((char= ch #\*)  '*)
 ((char= ch #\/)  '/)
 ((char= ch #\+)  '+)
 ((and (char= ch #\<)
       (char= (peek-char nil stream) #\()) '|lt|)
 ((and (char= ch #\>)
       (char= (peek-char nil stream) #\()) '|gt|)
 ((and (char= ch #\=)
       (char= (peek-char nil stream) #\()) '|eqn|)
```

```
 ((and (char= ch #\\)
       (char= (peek-char nil stream) #\+))
  (read-char stream) '|not|)
 ((and (char= ch #\<)
       (char= (peek-char nil stream) #\=))
  (read-char stream) '|<=|)
 ((and (char= ch #\>)
       (char= (peek-char nil stream) #\=))
  (read-char stream) '|>=|)
 (t (do ((lch (list ch) (push (read-char stream) lch)))
((not (alphanum (peek-char nil stream)))
 (implode (reverse lch)))  ))))

(defun read_at (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((char= (peek-char nil stream) #\') (read-char stream)
       (implode (reverse lch)))))

(defun do_l (x) (if (atom x) x
    (list '(\. . 2)
  (car x)
  (do_l (cdr x)))))

(defun read_string (ch &optional (stream *standard-input*))
  (do ((lch (list (char-int ch)) (push (char-int (read-char stream)) lch)))
      ((char= (peek-char nil stream) #\") (read-char stream)
       (do_l (reverse lch)))))

(defun read_var (type_var ch &optional (stream *standard-input*))
  (let ((v (read_atom ch stream)))
    (cons type_var
  (position v (if (member v *lvar)
  *lvar
  (setq *lvar (append *lvar (list v)))))) ))

(defun read_token (ch &optional (stream *standard-input*))
  (cond
    ((and (eq ch #\n) (upper-case-p (peek-char nil stream)))
     (read_var 'N (read-char stream) stream))
    ((or (spcial ch) (upper-case-p ch)) (read_var 'V ch stream))
    ((digit-char-p ch) (read_number 1 ch stream))
    ((and (char= ch #\-) (digit-char-p (peek-char nil stream)))
     (read_number -1 (read-char stream) stream ))
    ((char= ch #\") (read_string (read-char stream) stream))
```

```
    ((char= ch #\') (read_at (read-char stream) stream))
    ((char= ch #\#) (unread-char ch stream) (read stream))
    ((char= ch #\.) '|.|)
    ((char= ch #\,) '|,|)
    ((char= ch #\() '|(|)
    ((char= ch #\)) '|)|)
    (t (read_atom ch stream))))

(defun read_tokens (ch &optional (stream *standard-input*))
  (let ((tkn (read_token ch stream))
(c (rchnsep stream)))
    (if (char= c #\.) (list tkn)
(cons tkn (read_tokens c stream)))))

(defun read_tokens_cl(&optional (stream *standard-input*))
  (setq *lvar nil)
  (read_tokens (rchnsep stream) stream))

(defun read_simple (ch &optional (stream *standard-input*))
  (cond
    ((and (eq ch #\n) (upper-case-p (peek-char nil stream)))
     (read_var 'N (read-char stream) stream))
    ((or (spcial ch) (upper-case-p ch)) (read_var 'V ch stream))
    ((digit-char-p ch) (read_number 1 ch stream))
    ((and (char= ch #\-) (digit-char-p (peek-char nil stream)))
     (read_number -1 (read-char stream) stream ))
    ((char= ch #\") (read_string (read-char stream) stream))
    ((char= ch #\') (read_at (read-char stream) stream))
    ((char= ch #\#) (unread-char ch stream) (read stream))
   (t (read_atom ch stream))))

(defun read_fct (ch &optional (stream *standard-input*))
  (let ((fct (read_simple ch stream))
(c (rchnsep stream)))
    (if (char= c #\()
(let ((la (read_args (rchnsep stream) stream)))
  (cons (list fct ) la))
(progn (unread-char c stream) fct))))

(defun read_args (ch &optional (stream *standard-input*))
  (let ((arg (read_term ch stream)))
    (if (char= (rchnsep stream) #\,)
(cons arg (read_args (rchnsep stream) stream))
(list arg))))
```

```
(defun read_factor (ch &optional (stream *standard-input*))
  (cond
    ((or (spcial ch) (upper-case-p ch)) (read_var 'V ch stream))
    ((digit-char-p ch) (read_number 1 ch stream))
    ((char= ch #\") (read_string  (read-char stream) stream))
    ((char= ch #\') (read_at (read-char stream) stream))
    ((char= ch #\#) (unread-char ch stream) (read stream))
    ((char= ch #\() '|(|)
    ((char= ch #\)) '|)|)
    (t (read_atom ch stream))))


(defun read_expr (ch &optional (stream *standard-input*))
  (let ((arg (read_factor ch stream)))
    (let ( (next-ch (rchnsep stream)))
      (cond ( (eql next-ch #\,)
     (list arg) )
    ( (eql next-ch #\.)
     (unread-char next-ch stream)
      (list arg))
    (t (unread-char next-ch stream)
       (cons arg (read_expr (rchnsep stream) stream)) ))) ))

(defparameter *cns* '(\.)) ;;(the fixnum 2)))

(defun read_list (ch &optional (stream *standard-input*))
  (if (char= ch #\])
      ()
      (let ((te (read_term ch stream)))
(case (rchnsep stream)
  (#\, (list *cns* te (read_list (rchnsep stream) stream)))
  (#\| (prog1 (list *cns* te
    (read_term (rchnsep stream) stream))
 (rchnsep stream)))
  (#\] (list *cns* te nil))))))

(defun read_term (ch &optional (stream *standard-input*))
  (if (char= ch #\[)
      (read_list (rchnsep stream) stream)
      (read_fct ch stream)))

(defun mknotvars(n args)
  (if (null args) args
```

```
      (cons (cons 'V n)
    (mknotvars (+ n 1) (cdr args))) ))

(defun not-by-failing(s)
  (if (consp s)
      (cons (intern (format nil "not_~a" (string (car s))))
    (cdr s))
      s))

(defun not-symbol-name(s)
  (if (symbolp s)
      (intern (format nil "not_~a" (string  s)))
      s))

(defun addnot(s)
  (let* ( (n (length (cdr s)))
 (args (mknotvars 0 (cdr s)))
  (pred (car s))
  (notpred (not-symbol-name pred)) )
    (when (not (get notpred 'def))
      (add_cl notpred
      `(,n ( ,notpred ,@args)
   ( ,pred ,@args) (! ,n) (|fail|)) 'def)
      (add_cl notpred  `(,n (,notpred ,@args)) 'def )
      (format t "notdef: ~a :- ~a~%" notpred (get notpred 'def)))))

(defun read_tail (ch &optional (stream *standard-input*))
  (let ((tete (read_pred ch stream)))
    (cond( (equal tete '(|one|))
  (let* ((solvendum (read_pred (rchnsep stream) stream))
 (chr (rchnsep stream)))
    (cond ( (char= chr #\.) (list tete solvendum '(|nt|)))
  ( (char= chr #\,)
   (cons tete (cons solvendum
    (cons '(|nt|)
  (read_tail
   (rchnsep stream) stream)))))
  (t (unread-char chr stream)
     (cons tete (read_tail (rchnsep stream) stream))))))
 ( (equal tete '(|not|))
  (let* ((solvendum (read_pred (rchnsep stream) stream))
 (chr (rchnsep stream)))
    (addnot solvendum)
    (cond ( (char= chr #\.) (list (not-by-failing solvendum)))
```

```
  ( (char= chr #\,)
   (cons (not-by-failing solvendum)
 (read_tail (rchnsep stream) stream)))
  (t (unread-char chr stream)
     (cons tete (read_tail (rchnsep stream) stream))))))
 (t (let ((chr (rchnsep stream)))
      (cond ( (char= chr #\.) (list tete))
    ( (char= chr #\,)
     (cons tete (read_tail (rchnsep stream) stream)))
    (t (unread-char chr stream)
       (cons tete (read_tail (rchnsep stream) stream))))))  )))

(defun vname(v)
  (cond ((not (vvarp v)) v)
(t (let ((x (symbol-name v)))
     (intern (subseq x 1 (length x)))))))

(defun mkLispID(n)
  (intern (string-upcase
   (symbol-name  n))))

;; Read deterministic Prolog

(defun ximplode (lch) (intern (string-upcase
       (map 'string #'identity lch) )))
(defun xread_atom (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((not (alphanum (peek-char nil stream)))
       (ximplode (reverse lch)))))

(defun xread_at (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((char= (peek-char nil stream) #\')
       (read-char stream) (implode (reverse lch)))))

(defun xread_string (ch &optional (stream *standard-input*))
  (do ((lch (list (char-int ch)) (push (char-int (read-char stream)) lch)))
      ((char= (peek-char nil stream) #\") (read-char stream)
       (coerce (reverse lch) 'string))))

(defun xread_simple (ch &optional (stream *standard-input*))
  (cond
    ((digit-char-p ch) (read_number 1 ch stream ))
    ((char= ch #\-) (read_number -1 (read-char stream) stream ))
```

```
    ((char= ch #\") (xread_string (read-char stream) stream))
    ((char= ch #\') (xread_at (read-char stream) stream))
    ((char= ch #\#) (unread-char ch stream) (read stream))
    (t (xread_atom ch stream))))

(defun xread_fct (ch &optional (stream *standard-input*))
  (let ((fct (xread_simple ch stream)) (c (rchnsep stream)))
    (if (char= c #\()
(let ((la (xread_args (rchnsep stream) stream)))
  (cons (list fct (length la)) la))
(progn (unread-char c stream) fct))))

(defun xread_args (ch &optional (stream *standard-input*))
  (let ((arg (xread_term ch stream)))
    (if (char= (rchnsep stream) #\,)
(cons arg (xread_args (rchnsep stream) stream))
(list arg))))

(defun xread_list (ch &optional (stream *standard-input*))
  (if (char= ch #\])
      ()
      (let ((te (xread_term ch stream)))
(case (rchnsep stream)
  (#\, (cons te (xread_list (rchnsep stream) stream)))
  (#\| (prog1 (cons te (read_term (rchnsep stream) stream))
 (rchnsep stream)))
  (#\] (cons te nil))))))

(defun xread_term (ch &optional (stream *standard-input*))
  (if (char= ch #\[)
      (xread_list (rchnsep stream) stream)
      (xread_fct ch stream)))

(defun xread_pred (ch &optional (stream *standard-input*))
  (let ((nom (xread_atom ch stream)) (c (rchnsep stream)))
    (if (char= c #\()
(cons nom (xread_args (rchnsep stream) stream))
(progn (unread-char c stream) (list nom)))))


(defun xread_tail (ch &optional (stream *standard-input*))
  (let ((tete (xread_pred ch stream)))
    (if (char= (rchnsep stream) #\.)
(list tete)
```

```
(cons tete (xread_tail (rchnsep stream) stream)))))

;;end read deterministic Prolog

;; Tools for Lisp-Prolog communication:

(defun fix(xs)
  (mapcar #'mkLispID xs ))

(defun sfx(s) (car (last s)))
(defun rmsfx(s) (butlast s))

(defun pfx(s) (car (cdr s)))
(defun rmpfx(s) (cons (car s) (cddr s)))


(defun vvarp(v)
  (and  (symbolp v)
(eql (aref (symbol-name v) 0) #\?)))

(defun modedeclarationp(s)
  (cond ((null s) nil)
((and (consp s) (vvarp (car s))) t)
(t (modedeclarationp (cdr s))) ))

(defun notStructure(x)
  (not (and (consp x)
    (not (eq (car x) 'V)))))

(defun hasNoStructure(pred)
  (or (atom pred)
      (eq (car pred) '|is|)
      (every #'notStructure pred)))

(defun onlyLocals(cl)
  (every #'hasNoStructure cl))

(defun changePred(p)
  (cond ((null p) p)
((atom p) p)
((eq (car p) 'V)
 (cons 'L (changePred (cdr p))))
(t (cons (changePred (car p)) (changePred (cdr p)))) ))
```

```
(defun changeClause(cl)
  (mapcar #'changePred cl))

(defun fixVar(cl)
  (if (onlyLocals cl)
      (changeClause cl)
      cl))

(defun read_clause (ch &optional (stream *standard-input*))
  (let ((tete (read_pred ch stream))
(neck (rchnsep stream)))
    (if (char= neck #\.)
(cond (  (modedeclarationp tete)
       (list 'mdef (fix  tete  )))
      (t (list tete)))
(let ((nneck (read-char stream)))
  (cond (  (and (char= neck #\:) (char= nneck #\-)
(get (mkLispId (car tete)) 'mod) )
 (let ((tail (xread_tail (rchnsep stream) stream)))
   (cons 'def  (cons (fix tete) tail))))
(t (cons tete
 (read_tail (rchnsep stream)
    stream))))  )) ))

(defun processIs(l)
  (cond
    ( (and l   (consp (cdr l))
   (consp (cadr l))
   (equal (car (cadr l)) '|is|))
     (cons (cons '|is|
 (cons (car l) (cdr (cadr l))))
   (processIs (cddr l))))
    (l (cons  (car l) (processIs (cdr l))))))

(defun processCut (l)
  (when l
    (cons
     (if (or (eq (caar l) '!)
     (eq (caar l) '|nt|))
 `(,(caar l) ,(length *lvar) ,@(cdar l))
 (car l))
     (processCut (cdr l)))))

(defun read_code_cl (&optional (stream *standard-input*))
```

```
  (let ((*lvar ()))
    (let ((x (read_clause (rchnsep stream) stream)))
      (cond ((member (car x)  '(pdef! pdef sdef! sdef mdef def))  x)
    (t  (cons (length *lvar)
      (processCut (processIs x))))))))

(defun read_code_tail (&optional (stream *standard-input*))
  (setq *lvar ())
  (let ((x (read_tail (rchnsep stream) stream)))
    (cons (length *lvar) (append (processCut (processIs x))
 (list '(|true|))))))

(defun listNom(X)
  (cond ( (and (consp X)
       (eq (car X) 'V)) X)
( (member X '(- + * /)) X)
(t (list X))))

(defun read_pred (ch &optional (stream *standard-input*))
  (let ((nom (read_simple ch stream))
(c (rchnsep stream)))
    (cond ( (equal nom '|is|)
   (cons nom (prefx (read_expr c stream) ) ))
  ( (char= c #\() (cons nom (read_args (rchnsep stream) stream)))
  ( (char= c #\,) (listNom nom))
  (t (unread-char c stream) (listNom nom)))))

(defun rdc(stream)
  (handler-case (read_code_cl stream)
    (error (e) (declare (ignorable e))  nil)))

(defun rd_file(fileName)
  (with-open-file (s fileName)
    (loop for l = (rdc s) then
 (rdc s)
       while l
       collect l)))

(defun arithmeticp(p)
  (and (consp p)
    (member (car p) '(+ - * /))))

(defun cnsname(c)
  (cond ( (symbolp c) c)
```

```
( (and (consp c) (equal (car c) 'quote))
  (cadr c))
((numberp c)
 (intern (format nil "~a" c))) ))

(defun ispred(p)
  (and (consp p)
       (member (car p) '(> < >= <= = eq eql equal))))

(defun apply-mode(xs ys)
  (cond ((and (null xs) (null ys)) nil)
((null xs) (error "wrong arity: ~a~%" ys))
((null ys) (error "wrong arity: ~a~%" xs))
((equal (car xs) (car ys))
 (cons (car xs) (apply-mode (cdr xs) (cdr ys))))
((equal (car xs) '-)
 (cons (car ys) (apply-mode (cdr xs) (cdr ys))))
((and (equal (car xs) '+)
      (symbolp (car ys)))
 (cons (list 'quote (car ys))
       (apply-mode (cdr xs) (cdr ys))))))

(defun chain(hd tail)
  (cond ((null tail) `(values ,@(cdr hd)   ))
((arithmeticp (car tail))
 `(multiple-value-bind
      ,(mapcar #'cnsname (cdr (car tail)))
      (values (,(car (car tail))
,@(cddr (car tail)))
      ,@(cddr (car tail)))
    (declare (ignorable ,@(mapcar #'cnsname
       (cdr (car tail)))))
    ,(chain hd (cdr tail))))
((ispred (car tail))
`(progn (when (not ,(car tail)) (return-from et nil))
   (return-from vel ,(chain hd (cdr tail))   )))
( (and (consp (car tail))
       (equal (car (car tail)) 'univ))
 `(let (( ,(cadr (car tail)) ,(caddr (car tail))))
    ,(chain hd (cdr tail))))
((and (consp (car tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
 `(multiple-value-bind
```

```
  ,(mapcar #'cnsname (cdr (car tail)))
      ,(apply-mode (get (car (car tail)) 'mod)
   (car tail))
      (declare (ignorable ,@(mapcar #'cnsname
    (cdr (car tail)))))
       ,(chain hd (cdr tail))) ))  )

(defun funChain(hd tail)
  (cond ((null tail)  (cadr hd))
((arithmeticp (car tail))
 `(let (( ,(cadr (car tail))
  ,(cons (car (car tail)) (cddr (car tail)) )))
  ,(funChain hd (cdr tail))))
((ispred (car tail))
 `(progn (when (not ,(car tail)) (return-from et nil))
 ,(funChain hd (cdr tail))   ))
( (and (consp (car tail))
       (equal (car (car tail)) 'univ))
 `(let (( ,(cadr (car tail)) ,(caddr (car tail))))
    ,(funChain hd (cdr tail))))
((and (consp (car tail))
      (null (cdr tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
  (cons (car (car tail)) (cddr (car tail))))
((and (consp (car tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
 `(let (( ,(cadr (car tail))
 ,(cons (car (car tail)) (cddr (car tail)) )))
,(funChain hd (cdr tail))))  ))


(defun mkFun(clause)
  `(block et ,(funChain (car clause) (cdr clause))))

(defun mkAND(clause)
  `(block et ,(chain (car clause) (cdr clause))))

(defun shw(x)
  (format t "Code= ~a~%" x) x)

(defmacro clauseSetFunctor(s) `(caar ,s))
```

```
(defun checkFunMode(m)
  (let (( md (get (clauseSetFunctor m) 'mod)))
  (and (eql (cadr md) '+)
       (every (lambda(x) (eql x '-)) (cddr md)) )))

(defun mkdef(args clauses &optional (funMode (checkFunMode (car clauses))))
   (if funMode
   (list 'lambda (cdr args)
 (cons 'or
      (loop for x in clauses collect
   (mkFun x))))
   (list 'lambda args
 `(declare (ignorable ,@args))
 (cons 'block (cons 'vel
      (loop for x in clauses collect
   (mkAnd x))))))  )

(defun ck(nm args p clauses)
  (if (and (consp p) (equal args p)
   (every #'symbolp p))
      clauses
      (error "(def ~a ~a|~a...)?" nm args p)))

(defun getfun(s) (car s))
(defun getargs(s)  (cdr s))

(defmacro def (&rest clause)
  (let* ((hd (car clause))
 (fun (getfun hd))
 (args (getargs hd)))
  (setf (get fun 'clauses)
     `(,@(get fun 'clauses)
 ,(ck fun (get fun 'vs) args clause )))
  `(setf (symbol-function ',fun)
 ,(mkdef args (get fun 'clauses)) )))

(defun vmode(v)
  (cond ((vvarp v) '+)
(t '-)))

(defun argnames(vs)
  (loop for v in vs collect (vname v)))

(defun declaremodes(vs)
```

```
  (loop for v in vs collect (vmode v)))

(defmacro mdef(hd)
  (let ((fun (getfun hd))
(args (getargs hd)))
    `(progn (setf (get ',fun 'clauses) nil)
 (setf (get ',fun 'vs) (argnames ',args))
 (setf (get ',fun 'mod) (cons ',fun (declaremodes ',args))))))

;; WAM Virtual Machine


;; I. Registers
;;

;; Bottom of the heap, or copy stack
(defconstant BottomG (the fixnum 0))
(defconstant BottomL (the fixnum 500000))
(defconstant +LocalStackOverflow+ (the fixnum 1500000))

;; Memory implements the local stack and the heap.
(defvar Memory (make-array +LocalStackOverflow+
:initial-element 0 ))

;; The trail stack is implemented in a separate vector.
(defconstant BottomTR (the fixnum 0))
(defconstant +TrailOverflow+ 1000000)
(defvar trailMem
  (make-array +TrailOverflow+ :initial-element 0
      :element-type 'fixnum))

;; The arguments of a predicate are stored in xArgs.
(defvar xArgs (make-array 50 :initial-element 0))

(defvar TR (the fixnum 0)) ;;top of the trail stack

(defvar *LocalPointer*     ;;top of the local stack
  (the fixnum 0))

(defvar *GPointer*         ;;top of the copy stack
  (the fixnum 0))

(defvar CP (the fixnum 0))     ;;current continuation
(defvar CL (the fixnum 0))     ;;mother block
```

```
(defvar Cut_pt (the fixnum 0)) ;;specific cut
(defvar BL (the fixnum 0))     ;;last choice point
(defvar BG (the fixnum 0))
(defvar PC (the fixnum 0))     ;;current goal
(defvar PCE (the fixnum 0))    ;;current environment
(defvar Duboulot)



(defmacro functorCopy (des largs)
  `(cons (cons (car ,des) t) ,largs))

(defmacro functorDescription (te) `(car ,te))
(defmacro functor (description) `(car ,description))

(defmacro largs (x) `(cdr ,x))
(defmacro fargs (x) `(cdr ,x))
(defmacro var? (x)
  `(and (consp ,x) (numberp (cdr ,x))))

(defmacro list? (x)
  `(eq (functor (functorDescription ,x)) '\.))

;; II. Local Stack
;;

;; The WAM environment contains [CL CP Cut E]
;;
(defmacro CL (b) `(svref Memory ,b))
(defmacro CP (b) `(svref Memory (1+ ,b)))
(defmacro Cut (b) `(svref Memory (+ ,b 2)))
(defmacro Environment (b) `(+ ,b 3))

(defmacro vset (v i x) `(setf (svref ,v ,i) ,x))

(defmacro push_continuation ()
  `(progn (vset Memory *LocalPointer* CL)
  (vset Memory (1+ *LocalPointer*) CP)))

(defmacro push_Environment (n)
  `(let ((top (+ *LocalPointer* 3 ,n)))
     (if (>= top +LocalStackOverflow+)
 (throw 'debord (print "Local Stack Overflow")))
     (vset Memory (+ *LocalPointer* 2) Cut_pt)
```

```
     (dotimes (i ,n top) (vset Memory (decf top) (cons 'V top)))))

(defmacro max_Local (nl) `(incf *LocalPointer* (+ 3 ,nl)))


;;choice-point : [a1 .. an A BCP BCL BG BL BP TR]
;;
(defmacro TR (b) `(svref Memory (1- ,b)))
(defmacro BP (b) `(svref Memory (- ,b 2)))
(defmacro BL (b) `(svref Memory (- ,b 3)))
(defmacro BG (b) `(svref Memory (- ,b 4)))
(defmacro BCL (b) `(svref Memory (- ,b 5)))
(defmacro BCP (b) `(svref Memory (- ,b 6)))
(defmacro AChoice (b) `(svref Memory (- ,b 7)))

(defun save_args ()
  (dotimes (i (svref xArgs 0)
      (vset Memory (incf *LocalPointer* i) i))
    (declare (fixnum i))
    (vset Memory (+ *LocalPointer* i)
  (svref xArgs (+ i 1)))))

(defun push_choice ()
  (save_args)
  (vset Memory (incf *LocalPointer*) CP)
  (vset Memory (incf *LocalPointer*) CL)
  (vset Memory (incf *LocalPointer*) *GPointer*)
  (vset Memory (incf *LocalPointer*) BL)
  (vset Memory (incf *LocalPointer* 2) TR)
  (setq BL (incf *LocalPointer*) BG *GPointer*))

(defun push_bpr (resto) (vset Memory (- BL 2) resto))
(defmacro size_C (b) `(+ 7 (AChoice ,b)))

(defun pop_choice ()
  (setq *LocalPointer* (- BL (size_C BL))
BL (BL BL)
BG (if (zerop BL)
       BottomG (BG BL))))

;; III. Copy Stack
;;

(defmacro push_Global (x)
```

```
  `(if (>= (incf *GPointer*) BottomL)
       (throw 'debord (print "Heap Overflow"))
     (vset Memory *GPointer* ,x)))

(defmacro adr (v e) `(+ (cdr ,v) ,e))

(defun copy (x e)
  (cond
   ((var? x) (let ((te (ult (adr x e))))
       (if (var? te) (genvar (cdr te)) te)))
   ((atom x) x)
   ((functorCopy (functorDescription x)
  (mapcar (lambda(x) (copy x e)) (fargs x))) )))


(defmacro recopy (x e) `(push_Global (copy ,x ,e)))
(defmacro copy? (te)
  `(cdr (functorDescription ,te)))

;;IV. Trail
;;

(defmacro trailset(v i x) `(setf (aref ,v ,i) ,x))

(defmacro pushtrail (x te)
  (declare (ignorable te))
  `(cond ((>= TR +TrailOverflow+)
  (throw 'debord (print "Trail Overflow")))
 ( (trailset trailMem TR ,x) (incf TR) )))

(defmacro poptrail (top)
  `(do () ((= TR ,top))
     (let ((v (aref trailMem (decf TR)) ))
       (vset Memory v (cons 'V v)))))



;; mini-wam
;; utilities
;;

(defmacro nvar (c) `(car ,c))
(defmacro head (c) `(cadr ,c))
(defmacro tail (c) `(cddr ,c))
```

```
(defmacro pred (g) `(car ,g))

(defmacro partial? (g) `(get (pred ,g) 'partial))
(defmacro user? (g) `(get (pred ,g) 'def))
(defmacro builtin? (g) `(get (pred ,g) 'evaluable))

(defmacro def_of (g)
  `(get (pred ,g)
      (if (largs ,g)
       (nature (ultimate (car (largs ,g)) PCE)) 'def)))

(defmacro def_part (g)
  `(get (pred ,g) 'partial))

(defun nature (te)
  (cond
   ((var? te) 'def)
   ((null te) 'empty)
   ((atom te) 'atom)
   ((list? te) 'list)
   (t 'fonct)))

(defun getOutputVariables(c)
  (let* (  (fn (car c))
  (mods (cdr (get fn 'mod)))
  (args (cdr c)))
    (loop for x in args
  for m in mods
 when  (equal m '+) collect x)))


(defun mkCall(application)
  (let* (  (fn (car application))
   (mods (cdr (get fn  'mod)))
   (args (cdr application)))
    (cons fn
       (loop for x in args
  for m in mods
  for qx = (if (equal m '+)
       (list 'quote x)
       (list 'value x))
 collect qx))))

(defun mkFunCall(application)
```

```
  (let* (  (fn (car application))
   (mods (cdr (get fn  'mod)))
   (args (cdr application)))
    (cons fn
       (loop for x in args
  for m in mods
  when (equal m '-)
    collect (list 'value x)))))

(defun mkprologside(application)
    (let* (  (fn (car application))
   (mods (cdr (get fn  'mod))))
      (loop for x in (cdr application)
 for m in mods
 collect
   (if (equal m '+)
       (list (gensym (symbol-name x)))
       x))))

(defun denudeGlobalVars(s)
  (loop for x in s
     collect (if (consp x) (car x) x)))

(defun theOutputVariables(s)
  (loop for x in s when (consp x) collect (car x)))

(defun uniglobals(gs vs)
  (loop for g in gs
     for v in vs
     collect (list 'uni g v)))

(defparameter *defs* nil)



(defmacro notsafe? (x)
  `(and (not CP) (>= (cdr ,x) CL)))

(defmacro bind (x te)
  `(progn
     (if (or (and (> ,x BottomL) (< ,x BL))
     (<= ,x BG))
 (pushtrail ,x ,te))
     (vset Memory ,x ,te)))
```

```
(defmacro bindv (x y)
  `(if (< (cdr ,x) (cdr ,y))
       (bind (cdr ,y) ,x)
       (bind (cdr ,x) ,y)))

(defun unifnum (x y)
    (cond
     ((eql x y) t)
     ((var? y)
   (if (var? x)
 (if (= (cdr x) (cdr y)) t (bindv y x))
       (bindte (cdr y) x)))
     ((var? x)
      (bindte (cdr x) y))
     ((or (atom x) (atom y)) (throw 'impossible 'fail))
     (t (throw 'impossible 'fail))))


(defun generate_specialized_unification(i arg clauses)
  (declare (ignorable clauses))
  (if  (or (and (consp arg)
(eq (car (cadr arg)) 'N))
   (numberp arg))
       `(unifnum (svref xArgs ,i)
     (ultimate ,arg env))
     `(unif (svref xArgs ,i)
 (ultimate ,arg env))))

(defun generate_arg_unifications (nv args clauses)
  ;;(format t "nv= ~a, args= ~a~%" nv args)
  (if (eq nv 0) t
  (list 'catch (list 'quote 'impossible)
`(let ((env (push_Environment ,nv)))
   ,@(loop for i from 1 to nv
   for x in args
collect
  (generate_specialized_unification
   i (if (numberp x) x (list 'quote x))
   clauses))) )))

(defun specialized_choice (unifs paq)
  (let* ((resu  (shallow_backtrack paq)  )
```

```
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
    (if (funcall unifs) ;(eq (unify_with (largs (head c))
    ; (push_Environment (nvar c))) 'fail)
     (backtrack)
     (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c)))))))

(defun generate_choice (unifs paq)
  `(let* ((resu  (shallow_backtrack ,paq)  )
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
    (if ,unifs ;(eq (unify_with (largs (head c))
    ; (push_Environment (nvar c))) 'fail)
     (backtrack)
     (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c)))))))


(defun select-unif(s)
  (if (and (consp s) (eq (car s) 'N))
      'unifnum 'unif))

(defun provit (paq)

  (if (cdr paq)
      (let* ((caput (car (cadr paq)))
    ; (tc (tail paq))
     (args (largs (head caput)))
     (nv (length args)))
```

```
(declare (ignorable caput args nv))
    ;; (format t "unifs= ~a~%" unifs)
`(progn (push_choice)

;;(pr_choice  ,paq)
(let* ((resu  (shallow_backtrack ,paq)  )
      (c (car resu)) (r (cdr resu)))
  (cond ( (null r)
  (pop_choice)
  (if (eq
       (let ((unargs (largs (head c)))
     (e (push_Environment (nvar c))))
 (declare (ignorable e unargs))
  (catch 'impossible
  ,@(loop for i from 1 to nv
       collect
 (list (select-unif (pop args))
       (list 'svref 'xArgs i)
    '(ultimate (pop unargs) e))))
       ) 'fail)
     (backtrack)
     (when (tail c) (push_continuation)
(setq CP (tail c) CL *LocalPointer*)
      (max_Local (nvar c)))))
( (push_bpr r)
  (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c))) ))) ))
  ;;  (pr_choice ,paq)))
     (let* ((c (car paq))
    (tc (when (cdr paq) (tail paq)))
    (args (largs (head  c  ))  )
    (nv (length args))
    (unifs (generate_arg_unifications nv args paq)) )
`(if  (eq ,unifs 'fail)

      (if (zerop BL) ;; backtrack
(setq Duboulot nil)
(progn (setq *LocalPointer* BL
     *GPointer* BG Cut_pt (BL BL)
     CP (BCP *LocalPointer*)
     CL (BCL *LocalPointer*))
       (load_A2)
```

```
       (poptrail (TR BL))
       (pr_choice (BP *LocalPointer*))))

    (when ,tc (push_continuation)
      (setq CP ,tc CL *LocalPointer*)
      (max_Local ,nv)))  )))

(defun compile-args(pred ind)
  (let* ( (nargs (length (cdr (cadar (get pred ind)))))
  (definition (list 'quote (get pred ind)))
  (vs (loop for i from 1 to nargs collect
   `(vset xArgs ,i
 (let ((te (ultimate (pop largs) PCE)))
     (cond
( (atom te) te)
( (var? te)
  (if (notsafe? te)
    (genvar (cdr te)) te))
( (copy? te) te)
( (recopy te PCE))))  ))))
  `(lambda()
     (let ((largs (largs PC)))
     (declare (ignorable largs))
  (vset xArgs 0 ,nargs)
  ,@vs  )
     (if CP ,(provit definition)
     (progn
       (if (<= BL CL)
   (setq *LocalPointer* CL))
   (setq CP (CP CL) CL (CL CL))
   ,(provit definition))  ))))


;; The first compiler clause handles deterministic Prolog
;; The second clause makes deterministic statement
;; The third executes non-deterministic Prolog
(defun add_cl (pred c ind)
  (cond  (  (and (equal pred 'def))
    (let* ((nm (intern (string-downcase
    (symbol-name (caadr  c)))))
(args (cdr (cadr c)))
(prologside (mkprologside (cadr c)))
   (fn
    (if (checkFunMode (cdr c))
```

```
‘(lambda(,@(denudeGlobalVars prologside))
   (uni ,(car (car prologside))
,(mkFunCall (car (cdr c)))))
‘(lambda(,@(denudeGlobalVars prologside))
(multiple-value-bind ,args
   ,(mkCall  (car (cdr c))  ))
   (declare (ignorable ,@args))
   ,@(uniglobals
       (theOutputVariables prologside)
       (getOutputVariables (cadr c)  )   )) ))  ))
      (eval c)
      (setf (symbol-function nm)
    (eval fn))
      (setf (get nm ’evaluable) t)))
 (  (equal pred ’mdef)
  (eval c)  )
 ((equal ind ’def)
  (when (not (member pred *defs*))
       (push pred *defs*))
  (setf (get pred ind) (append (get pred ind) (list c)))
  (setf (get pred ’partial) t)
  (setf (symbol-function pred)
(eval (compile-args pred ind))))
 (t  (when (not (member pred *defs*))
       (push pred *defs*))
     (setf (get pred ind) (append (get pred ind) (list c))))))

(set-macro-character
 #\$
 #’(lambda (stream char)
     (declare (ignorable char))
     (let* ( (*standard-input* stream) (c (read_code_cl)))

       (add_cl (if (symbolp (car c)) (car c)
   (pred (head c ))) c ’def)
       (if (largs (head c))
   (let ((b (nature (car (largs (head c))))))
     (if (eq b ’def)
 (mapc
  #’ (lambda (x) (add_cl (pred (head c)) c x))
  ’(atom empty list fonct))
       (add_cl (pred (head c)) c b)))))
     (values)))
```

```
(defun yes/no()
  (princ "More : ")
  (finish-output)
  (member (rchnsep) '(#\o #\y #\s #\d #\j #\;)))

(defun answer ()
  (printvar)
  (if (zerop BL)
      (setq Duboulot nil)
    (if (yes/no)
(backtrack)
      (setq Duboulot nil))))

(defun printvar ()
  (if (null *lvar)
      (format t "Yes ~%")
    (let ((n -1))
      (mapc
       #' (lambda (x)
    (format t "~A = " x)
    (write1 (ult (+ (incf n)
    (Environment BottomL)))) (terpri))
       *lvar))))

;; mini-wam
;; unification
;;

(defun ult (m)
  (declare (fixnum m))
  (do* ( (n m (cdr te))  (te (svref Memory n) (svref Memory n)))
       ( (not (and (var? te) (/= (cdr te) n))) te)
      (declare (fixnum n))))

(defun ultimate (x e) (if (var? x) (ult (adr x e)) x))
(defun val (x) (if (var? x) (ult (cdr x)) x))


(defun bindte (xadr y)
  (declare (fixnum xadr))
  (if (or (atom y) (copy? y))
      (bind xadr y)
      (bind xadr (recopy y (Environment *LocalPointer*)))))
```

```
(defun genvar (x) (declare (fixnum x))
       (bind x (push_Global (cons 'V *GPointer*))))



(defun unify_with (largs e)
  (catch 'impossible
    (do ((i 1 (1+ i)))
((null largs))
(declare (fixnum i))
      (unif (svref xArgs   i)
    (ultimate (pop largs) e)))))

(defmacro mval(x) `(if (var? ,x) (ult (cdr ,x)) ,x))

(defun unif (x y)
    (cond
     ((eql x y) t)
     ((var? y)
   (if (var? x)
 (if (= (cdr x) (cdr y)) t (bindv y x))
       (bindte (cdr y) x)))
     ((var? x)
      (bindte (cdr x) y))
   ((or (atom x) (atom y)) (throw 'impossible 'fail))
   ((let ((b (copy? y)) (dx (pop x)) (dy (pop y)))
      (if (eq (functor dx) (functor dy))
  (do* ( (ax x (cdr ax))
 (vx (mval (car ax)) (mval (car ax)) )
 (vy (pop y) (pop y)))
       ((null ax))
      (unif vx
    (if b (mval vy)
(ultimate vy (Environment *LocalPointer*)))  ))
(throw 'impossible 'fail))))))


;; mini-wam
;; resolution
;;

(defun lispforward ()
  (do () ((null Duboulot))
      (cond ((null CP) (answer))
```

```
    ( (load_PC)
     (cond
       ((partial? PC)
(funcall (car PC)))
       ((user? PC)
(let ((d (def_of PC)))
  (if d (pr2 d) (backtrack))))
       ((builtin? PC)
(if (eq (apply (car PC) (cdr PC)) 'fail)
    (backtrack)
  (cont_eval)))
       ((backtrack)))))))

(defmacro valor (x)
  `(if (or (var? ,x) (atom ,x))
       (ultimate ,x PCE)
       (copy ,x PCE)))

(defun forward ()
  (do () ((null Duboulot) (format t "no More ~%"))
    (cond ((null CP) (answer))
    ( (load_PC)
      ;; PC contains the goal
     (cond
       ((partial? PC)
(funcall (car PC)))
       ((equal (car PC) '|call|)
(let ((fn (valor (cadr PC)))
      (xs   (cddr PC)))
  (setf PC (cons fn xs))
  (format t "call=~a~%" PC)
  (funcall (car PC))))
       ((user? PC)
(let ((d (def_of PC)))
  (if d (pr2 d) (backtrack))))
       ((builtin? PC)
(if (eq (apply (car PC) (cdr PC)) 'fail)
    (backtrack)
  (cont_eval)))
       ((backtrack)))))))

(defun load_PC ()
  (setq PC (pop CP) PCE (Environment CL) Cut_pt BL))
```

```
(defun cont_eval ()
  (unless CP
    (if (<= BL CL) (setq *LocalPointer* CL))
 (setq CP (CP CL) CL (CL CL))))

(defun pr_choice (paq)
  (let* ((resu  (shallow_backtrack paq)  )
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
    (if (eq (unify_with (largs (head c))
     (push_Environment (nvar c))) 'fail)
     (backtrack)
     (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c)))))))

(defun shallow_backtrack (paq)
  (if (and (cdr paq)
   (eq (unify_with
(largs (head (car paq)))
(push_Environment (nvar (car paq))))
       'fail))
      (progn
(poptrail (TR BL))
(setq *GPointer* BG)
(shallow_backtrack (cdr paq)))
    paq)  )

(defun backtrack ()
  (if (zerop BL)
      (setq Duboulot nil)
      (progn (setq *LocalPointer* BL *GPointer* BG Cut_pt (BL BL)
   CP (BCP *LocalPointer*) CL (BCL *LocalPointer*))
   (load_A2)
   (poptrail (TR BL))
   (pr_choice (BP *LocalPointer*)))))

(defun load_A2 ()
```

```
  (let ((deb (- *LocalPointer* (size_C *LocalPointer*))))
    (dotimes (i (AChoice *LocalPointer*) (vset xArgs 0 i))
      (declare (fixnum i))
(vset xArgs (+ i 1)
      (svref Memory (+ deb i))))))

(defun lisploop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (lispforward)))

(defun readProvePrintLoop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (forward))
  (cond ( *keep-going*
 (handler-case (readProvePrintLoop (read_prompt))
   (error(e) (format t "Error: ~a" e) (readProvePrintLoop (read_prompt)))))
(t (format t "Bye!~%")
   (setf *keep-going* t))))

;; mini-wam
;; default predicates
;;

(defvar Ob_Micro_Log
  '(|debugvar| |one| |nt| |write|
    |nl| |tab| |read| |get| |get0|
    |var| |nonvar| |atomic|
    |atom| |number| |clear|  |is|
    ! |fail| |true| |halt| |lisp|
    |div| |eqn| |gt| |ge|
    |divi| |mod| |plus| |call|
    |minus| |times| |le| |lt|
    |name| |consult| |sult|
    |abolish| |cputime| |statistics|))
```

```
(mapc #'(lambda (x) (setf (get x 'evaluable) t)) Ob_Micro_Log)

(defmacro value (x)
  `(if (or (var? ,x) (atom ,x))
       (ultimate ,x PCE)
       (copy ,x PCE)))
(defun uni (x y) (catch 'impossible (unif (value x) y)))


(defun |debugvar|(x)
  (format t " dvar ~a " x))

;;lisp/2
(defun |lisp|(res fn)
  (uni res
       (eval (cons (intern
    (string-upcase
       (symbol-name
(functor (functorDescription fn) ))))
 (mapcar (lambda(x) (value x)) (largs fn))) ) )

(defun evalfp(x)
  (cond ((numberp x) x)
( (and (consp x) (eq (car (functor x)) '|neg|))
 (- (evalfp (car (fargs x)))))
((functor x)
 (funcall (symbol-function (car (functor x)))
  (evalfp (car (fargs x)))
  (evalfp (cadr (fargs x))))) ))


;;write/1 (?term)
 (defun |write| (x) (write1 (value x)))
 (defun write1 (x)
      (cond
       ((null x) (format t "[]"))
       ((atom x) (format t "~A" x))
       ((var? x) (format t "X~A" (cdr x)))
       ((list? x) (format t "[")
(writesl (val (cadr x)) (val (caddr x)))
(format t "]"))
       ((writesf (functor (functorDescription x))
 (largs x)))))
```

```
(defun writesl (tete q)
      (write1 tete)
      (cond
       ((null q))
       ((var? q) (format t "|X~A" (cdr q)))
       (t (format t ",") (writesl (val (cadr q))
  (val (caddr q))))))

    (defun writesf (fct largs)
      (format t "~A(" fct)
      (write1 (val (car largs)))
      (mapc #'(lambda (x) (format t ",")
      (write1 (val x))) (cdr largs))
      (format t ")"))

;;nl/0
       (defun |nl| () (terpri))
;;tab/1 (+int)
(defun |tab| (x)
  (dotimes (i (value x)) (format t " ")))
;;read/1 (?term)
       (defun |read| (x)
 (let ((te (read_terme)))
   (catch 'impossible
     (unif (value x)
   (recopy (cdr te)
   (push_Environment (car te)))))))

(defun read_terme ()
  (let ((*lvar nil))
    (let ((te (read_term (rchnsep))))
      (rchnsep) (cons (length *lvar) te))))
;;get/1 (?car)
       (defun |get| (x) (uni x (char-int (rchnsep))))
;;get0/1 (?car)
       (defun |get0| (x) (uni x (char-int (read-char))))


;;var/1 (?term)
(defun |var| (x) (unless (var? (value x)) 'fail))
;;nonvar/1 (?term)
(defun |nonvar| (x) (if (var? (value x)) 'fail))
;;atomic/1 (?term)
(defun |atomic| (x) (if (listp (value x)) 'fail))
```

```
;;atom/1 (?term)
(defun |atom| (x) (unless (symbolp (value x)) 'fail))
;;number/1 (?term)
(defun |number| (x) (unless (numberp (value x)) 'fail))


;;fail/0
       (defun |fail| () 'fail)
;;true/0
       (defun |true| ())

;;divi/3 (+int,+int,?int)
(defun |divi| (z x y) (uni z (floor (value x) (value y))))
;;div/3
  (defun |div| (z x y) (uni z (/ (value x) (value y))))
;;mod/3 (+int,+int,?int)
       (defun |mod| (z x y) (uni z (rem (value x) (value y))))
;;plus/3 (+int,+int,?int)
       (defun |plus| (z x y) (uni z (+ (value x) (value y))))
;;minus/3 (+int,+int,?int)
       (defun |minus| (z x y) (uni z (- (value x) (value y))))
;;mult/3 (+int,+int,?int)
       (defun |times| (z x y) (uni z (* (value x) (value y))))
;;le/2 (+int,+int)
       (defun |le| (x y) (if (> (value x) (value y)) 'fail))
;;lt/2 (+int,+int)
       (defun |lt| (x y) (if (>= (value x) (value y)) 'fail))
;; eqn/2
(defun |eqn| (x y) (if (not (= (value x) (value y))) 'fail))
;;gt/2
(defun |gt| (x y) (if  (<= (value x) (value y)) 'fail))
;;ge/2
(defun |ge| (x y) (if (< (value x) (value y)) 'fail))
;;name/2 (?atom,?list)
(defun undo_l (x)
  (if (atom x)
       x
      (cons (undo_l (val (cadr x))) (undo_l (val (caddr x))))))

(defun |name| (x y)
  (let ((b (value x)))
     (if (var? b)
 (uni x (impl (undo_l (value y))))
 (uni y (do_l (expl b))))))
```

```
(defun impl (l) (intern (map 'string #'code-char l)))
(defun expl (at) (map 'list #'char-int (string at)))

(defun addit(c)
     (add_cl (if (symbolp (car c)) (car c)
   (pred (head c ))) c 'def)
       (if (largs (head c))
   (let ((b (nature (car (largs (head c))))))
     (if (eq b 'def)
 (mapc
  #' (lambda (x) (add_cl (pred (head c)) c x))
  '(atom empty list fonct))
       (add_cl (pred (head c)) c b)))))

(defun |tolisp| (f)
  (let ((nm (value f)))
    (setf (get nm 'partial) (get nm 'def))
    (setf (get nm 'def) nil)))

(defun |consult| (f)
  (let* ((filename (format nil "~a" (value f)))
 (code (rd_file filename)))
    ;;(format t "file= ~a~%" code)
    (loop for c in code
       do (addit c))))

;;consult/1 (+atom)
(defun |sult| (f) (format t "~A~%"
(load (format nil "~A" (value f))  )))

;; abolish/1
(defun |abolish| (p)
  (mapc  #'(lambda (x) (setf (get p x) nil))
 '(atom empty list fonct def)))

;; clear/0
(defun |clear| ()
  (mapc #'(lambda(x) (|abolish| x)) *defs*))

;; cputime/1
(defun |cputime| (x)
  (uni x (float (/ (get-internal-run-time)
   internal-time-units-per-second))))
```

```
;; statistics/0
(defun |statistics| ()
  (format t " local stack : ~A (~A used)~%"
  (- +LocalStackOverflow+ BottomL) (- *LocalPointer* BottomL))
  (format t " global stack : ~A (~A used)~%"
  BottomL (- *GPointer* BottomG))
  (format t " trail : ~A (~A used)~%"
  (- (array-dimension trailMem 0) BottomTR)
  (- TR BottomTR)))

(defun |halt| ()
  (setf *keep-going* nil))

(defvar *G* (the fixnum 0))
(defvar *L* (the fixnum 0))
(defvar *TR* (the fixnum 0))
(defvar *BG* (the fixnum 0))

(defun |one| ()
  (setq *G* *GPointer* *L* *LocalPointer* *TR* TR *BG* BG))

(defun |is|(x fx)
  (uni x (evalfp  (value fx ))))

(defun |nt|(n)
    (setq BL (Cut CL) BG (if (zerop BL) BottomG (BG BL))
  *LocalPointer* (+ CL 3 n))
    (when *TR* (setq TR *TR* *GPointer* *G* *G* nil *TR* nil)) )

;;cut/0
(defun ! (n)
  (setq TR (if (zerop BL) BottomTR (TR BL))
 BL (Cut CL)
BG (if (zerop BL) BottomG (BG BL))
       *LocalPointer* (+ CL 3 n)) )

(defun bye()
  (cl-user::exit))
```