A **C** Portable LISP Interpreter and Compiler [1]

E. Karabudak, G. Üçoluk [2] , T. Yılmaz [3]

August 1990 [4]

[1] This project was supported by the Revolving Fund Institution of METU, Grant:AFP 88-01-05-02

[2] Permanent address: Dept. of Computer Engineering, METU, Ankara, Turkey

[3] Permanent address: Dept. of Physics, METU, Ankara, Turkey

[4] This project was completed in a time of about three years.

The authors believe they have participated equally.

Contents

Chapter 1

About the Distribution

This software can be freely distributed and used provided the conditions described in the accompanying LICENSE.txt file are followed.

The authors can be contacted at:

Dr. Tugrul Yilmaz,
Dept. of Physics,
Middle East Technical Univ.
ODTU, Ankara, Turkiye
`tugrul@trmetu.bitnet`

Dr. Göktürk Üçoluk
Dept. of Computer Engineering
Middle East Technical Univ.
ODTU, Ankara, Turkiye
`ucoluk@trmetu.bitnet`

Ersin Karabudak
LOGO Coorp.
P.K. 322 Kadıköy,
Istanbul, Turkiye

Chapter 2

Introduction

This document intends to serve as a guide of a LISP interpreter and compiler coded in **C** . It is in no means a LISP language manual or tutorial.

LISP is the oldest and most used language in the world of AI programming and symbolic manipulation . Although PROLOG with its powerful first order predicate logic functional design and built-in backtracking mechanism becomes more and more popular, it is for certain that LISP will remain the most promising computer language in the fields mentioned, for many more years. It is worth to mention that there exists hardware architectures specific to LISP and Texas instruments brought a LISP dedicated microchip into the market, in the year 1986.

Parallel to the improvements in LISP software, an academic language that was supported by Bell laboratories gained more and more interest in the professional system programming world. With the similarly developed UNIX operating system this language, namely **C** , started to become an indispensable standard. Nowadays there exists hundred of manufactured computers, different in capacity and size, many of which offer the purchaser a UNIX environment and even a broader class offers a **C** language compiler, anyway. So **C** has became a standard language of serious project implementation. This fact was the reason why this project has been realized: A LISP that was coded in **C** would be totally portable. To get a working LISP on a completely new hardware would only require a **C** compiler and a couple of man-hour work of a non-expert.

In addition to the interpreter a LISP ⇒ **C** compiler which would take a LISP source and produce a **C** code would complete the picture. If this would be done then any project that had to be developed in an LISP could be realized on any computer with our system, provided that this computer supports a **C** compiler. After the development phase is completed the final LISP code could be compiled to a **C** program and then this **C** code could be moved around, ported to any computer and **C**-compiled there. So this work would not only enable a portability of LISP but also a versatility and portability of **any** LISP coded software.

In this document we describe the implementation of such an interpreter and compiler. The software introduced here is available from the authors.

It is something to regret that such a powerful language like LISP has not a standard. There has been effort to standardize this language, but presumably due to trading reasons and vicious academic quarrels still we are far from such a standard. Of course another reason is that LISP itself provides a unique convenience for modifying any **X**-LISP to become a **Y**-LISP. As known, LISP permits the redefinition of even the internally defined functions.

One of the attempts for a standardization was the standard LISP proposal of the Utah group. The proposal is described in "*Standard Lisp Report*"[2]. This LISP was used for the implementation of some LISP based SAM systems like REDUCE. It was named STD-LISP, later Utah-STD-LISP. **The LISP standard chosen for our implementation is this version**.

So, this is an implementation of Utah-STD-LISP. **It is not a COMMON LISP**. Furthermore the authors have strong feelings that COMMON LISP is still a wrong choice for standardization and another mistake is to promote it to be "COMMON".

2.1 Choice of Design Preferences

It is for sure that writing an interpreter and compiler has not to be done in a unique methodology. There exists parts of such a software where one has to choose among several tradeoffs. Each of these alternatives have their assets and liabilities. In this section the aim is to put and answer some global questions of this kind.

2.1.1 Parameter Passing

This is **not** a stack oriented LISP. If it would be so, the LISP arguments, pre-evaluated or not, would be pushed onto a stack and the relevant function would be called to carry out the evaluation. It is the right and duty of the called function to pop-off its arguments from stack, do the evaluation and finally push the result onto the stack. An implementation of this kind would be nice, cute and short in code. But the hardware world does not favor stack implementations. Although for a great amount of computers, there exist stack oriented microcode conveniences, they are expensive in time and do not correspond to a special hardware architecture, actually. But the inconveniences present in the case of stack, vanish if the implementation is shifted toward intensive register usage. This is unpleasant but truth. All architectures provide a certain number of memory locations (usually internal) –so called registers– that possess special hardware features especially related to fast data transfer and manipulation. Therefore the choice of argument passing is made in favor of register-based evaluation. This is also the choice for the compiler made by A.C. Hearn (*See the section : Compiler*)

2.1.2 Why Off-Line Compilation ?

Unlike some other LISP implementations there does not exist a function like `(COMPILE <fnname1> <fnname2> ...)` in the normal interpreter part of this system. Instead of this there exists a separate LISP interpreter with the compiler present (and usually used only for compilation purpose). If this version (which is of course bigger in size) is called and ordered to compile a LISP source, the products are four files which contain the **C** code and explained in the section : The Compiler. This generated **C** code along with the **C** code for the interpreter will be **C** compiled using a MAKE utility. The choice made for the internal data structure is such that it disables partial compilation and linking (e.g. the symbol table for the build in functions is hold as a static array, therefore it is not expandable). These choices are done only for efficiency reasons. We believe the interpreter is sufficient for the development phase of a project and the very last move is the compilation, The code generated this way will be in most integrity.

2.1.3 Why not an Integrated Development Environment ?

It is an undeniable fact that integrated development environments (IDE) become more and more popular, so why did we choose not to do so? The answer is in fact simple. First of all, the feature that the system developed should be fully **C** portable was vital. Just the fact that an IDE would require a build in editor would mean a necessity for a graphics or screen kernel which of course would be system dependent. Secondly, our task was to build a professional compiler for heavy work and not a toy environment for novice training. Furthermore a good IDE would be large in code size, and therefore, it might have caused problems on small memory systems.

2.2 Source Organization & Implementation

Much care has been taken to ensure the source code is self-documented and easy to understand. To achieve readability and consistency a number of conventions and organizations have been used in formulating the code. These are as follows.

- All the LISP source codes are in files with extension **L** .
- These files (files with extension **.l**) are not ready to compile. For details see the chapter on compilation.
- Macro-defined constants are appearing in upper case only.
- LISP data structures are all conventionally named to start with a capital letter **X**.
- `typedef` keywords of LISP data structures are all appearing in upper case only. Throughout the source code these keywords are used mostly and not the ones with **X**.

File organization and description of the files are given below.

big-n.l contains the arbitrary precision integer arithmetic support functions.

errors.l Contains the error messages of the LISP interpreter.

flags.l System dependent flag macros are defined in this file.

fnames.l Contains the names, internal names, number of arguments, and types of the STD-LISP functions of the interpreter.

hd.l Contains all the global static declarations and initializations of LISP interpreter.

lisp-fn.l Contains the implementations of the STD-LISP functions of the interpreter.

lisp-zfn.l Contains the support functions. Here the function names always begin with the letter **z**.

sysid.l List of predefined identifiers of STD-LISP, their flags, print names,and values. Used when BITF flag is zero.

sysids.l Contains the list of predefined identifiers of LISP, their attributes, print names,and values. Used when BITF flag is one.

type.l, types.l Contain the macro definitions, structure definitions, and type definitions of the implementation.

yylex.l Contains the parser support functions for LISP.

zfnames.l Contains the names of functions that are defined in **lisp-zfn.l**

Following utility programs are used to prepare the LISP file system.

cr1.c Hashtable of the predefined LISP identifiers is generated, and identifiers are declared statically.

cr2.c Prepares support functions for compilation.

crc.c Prepares compiler generated functions for compilation.

cri.c Prepares the initialiation file.

size.c Program which calculates `PAGESIZE`

In addition to these files you will find some other files related to the LISP ⇒ **C** compiler. They are grouped under the subdirectory **compiler** in distribution. Here you will find the following files:

compiler.lsp The LISP code of A.C. Hearn's Portable LISP compiler[4].

lap.lsp The LISP code that is the front end for the compiler. The job of this code is to convert the output of Hearn's compiler (which are LISP expresions corresponding to macro calls of a lisp-machine) to C function definitions and an initialization environment (we name this environment *urwelt* in rememberance of W.Heisenberg, meaning whatever existed before history started).

comp* These files are the outcome of the compilation of the compiler itself. How they are used is explained in details in the **"Compiler"** chapter.

2.2.1 LISP Options On The Command Line

The generic command-line format is

```
lisp [option option ...]
```

Each command line option is preceded by a dash (-), and separated from the other options by at least one space. There are five options available. Description of these options are given below.

-S# Sets the dynamic SEXPR stack size. Default size is 1024 cells and has been defined in **hd.l**. This option is valid if LISP has been compiled with non-zero DSTACK (dynamic stack) option macro.

-A# Sets `alist` stack size. Default size is 512 cells and has been defined in **hd.l**.

-P# Sets identifiers print name space size. Default size is 4000 bytes and has been defined in **hd.l**.

-T# Sets string space size. Default size is 3500 bytes and has been defined in **hd.l**.

-M# Sets minimum pair space size before garbage collection.

-G# Sets minimum number of pair space pages before garbage collection.

-E*environment_variable* Introduces to lisp *(or to the compiled lisp program)* the operating system environment variable which is set to the initialization file name. If not supplied the default variable name **LISPINI** is assumed and searched in the environment. Please note that it is the name of the environment variable that will follow the **-E** option, and not the file name itself.

For this option only: It is allowed to leave one or no blank between **-E** and the name of the envionment variable.

For example given the following command line

```
lisp -s2000 -a100 -P1000 -E LINIT
```

lisp will run with a stack of 2000 cells, alist stack of 100 elements and 1000 character long identifier print name space. An assignment of the environment variable LINIT variable should be made prior to the execution of the lisp execution. For example in UNIX BOURNE shell this would be some lines as:

```
LINIT=/usr/local/lisp/MYLISPINI
export LINIT
```

if the specific initialization file for this program lives in the **/usr/local/lisp** directory under the name **MYLISPINI** .

Chapter 3

Syntax and Data Structure

LISP objects (S-expressions or so called SEXPR's) are divided into two classes: atoms and dotted-pairs. Atoms are further classified as identifiers, integers, big integers, floating point numbers, strings, vectors and function pointers. Dotted-pairs are composite objects constructed by the LISP function `cons` (where `cons[a,b]` is denoted by (a . b)). The main universe of discourse for LISP is the closure of the set of all atoms under the operation `cons`. Of course the trivial operations among numbers are also included.

The syntax of these LISP objects and their corresponding internal data structure are explained in the following subsections. But before going into details we would like to make a brief summary of the static structure of this LISP. There exists 11 different data structures 7 of which corresponds to the above stated LISP objects. The remaining 4 are for internal use of which the user has an indirect discernment (like the error objects). These 11 data structures are all conventionally named to start with a capital letter **X**, here they are:

- Xpair
- Xid
- Xstring
- Xinteger
- Xbig
- Xfloating
- Xvector
- Xfpointer
- Xerrmsg
- Xstrelmnt
- Xforwardadr

There exist also a numbering of these objects for identification purpose. As a coding convention the variable '`tp`' is used all through the source for a variable that will hold one of these object-identification number.

• Tpair	0
• Tid	1
• Tstring	2
• Tinteger	3
• Tbig	4
• Tfloating	5
• Tvector	6
• Tfpointer	7
• Terrmsg	8
• Tpname	9
• Tsname	10
• Tforwardadr	11

These structures are associated to the following capital letter typedef keywords.

[throughout the next section these keywords will be used and not the ones with **X**].

PAIR	Xpair
ID	Xid
STRING	Xstring
INTEGER	Xinteger
BIG	Xbig
FLOATING	Xfloating
VECTOR	Xvector
FPOINTER	Xfpointer
ERRMSG	Xerrmsg
STRELEMENT	Xstrelmnt

To unify all pointer to these different data structures under a single name a continuous type casting is made: a hypothetical type is assumed, which is now the `int` type and a pointer type is defined as

```
typedef int *PSEXP;
```

and all LISP object data structure pointers are explicitly converted by type casting to this hypothetical pointer.

As will be explained in details in the section: Memory Management and GC, the data items are hold in linked memory pages each of which are of a constant size. This size is a multiple of the least-common-factor of the sizes of possible data structures that corresponds to a LISP object. The memory manager, upon demand for such a specific data type, allocates a page of this size, fills it with empty objects of that type, and then put it in a link with the pages of similar kind. So there exist `NTYPES`-many linked page lists where `NTYPES` is the count of data types (in this implementa tion 7). The start of these pages are hold in a pointer array.

Each data type has its first byte of common structure. This byte is named as the `Xtype` field and serves to identify what this object is. It holds the type identification number, described above that has a mnemonic define-name that starts with the letter **T** *(eg. like* `Tpair`*).*

Furthermore, for convenience, the sizes of each type is stored in an array '`sz[NTYPES]`' where `NTYPES` is the total count of above explained data types.

In the following sections, for each data object, first a syntactical explanation is given then the internal data structure is described.

3.1 Dotted Pairs

Usual dot and list notations for dotted pairs may be used in any combination on input. On output, they are written in the **shortest** possible form. For example

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

is printed as

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

but may be written in both forms by the user. Also the expression `(quote x)` may be abbreviated as `'x` where `x` is any S-expression.

3.1.1 Implementation

```
struct Xpair { char Xtype;
               PSEXP Xcar;
               PSEXP Xcdr; } ;
```

char	**PSEXP**	**PSEXP**
Xtype	Xcar	Xcdr

3.2 Identifiers

Are composed of letters, and non initial digits. Also, any other character may be included in an identifier by preceding (escaping) it with a `!` (exclamation mark). Identifiers are unlimited in length. Some examples of identifiers are :

```
a
AVeryLongIdentifier
U238
emsg!*
!!!&x
!2
```

But `2` and `u:238` are not identifiers. As an exception, the LISP reader treats an unescaped special character, other than those that are meaningful to it (i.e. parentheses, dot, quotes etc.) as a single character identifier. So `u:238` is read as a sequence of three atoms: identifier `u` , identifier `!:` and integer `238` . In fact the LISP reader will make S-expressions from anything you invent for input.

3.2.1 Implementation

Depending on a compiler flag `BITF` setting, one of the below given definition is performed. They differ only in the second field. In the first one the field `Xattr` is masked explicitly with some selecting bytes to fetch out the attribute bits. In the second this is done using the bit slicing property of the **C** language. The alternatives exist, since, some compilers generate inefficient codes for bit slicing.

An identifier can have several attributes, like being a global or fluid, used as a name of interpretively defined function etc. Some of the bits in the second field of this structure is deviced for this purpose, the remaining bits are left unused.

As known in LISP each identifier has an associated:

printname which is the way the identifier is recognized in input and output.

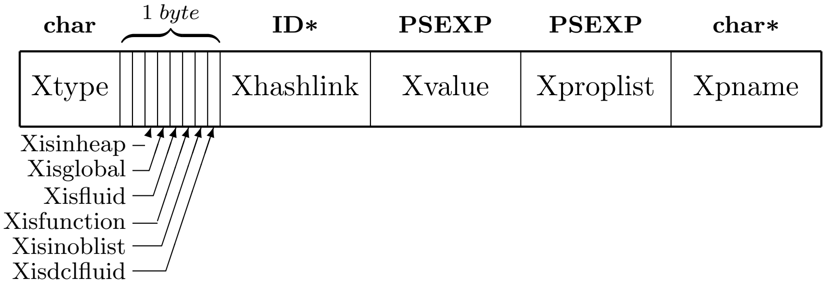
value cell where you can assign any SEXPR or can be left unassigned.

property-list where you can store several properties under indicators, or flag the identifier with to possess a property. This is a list of SEXPRs.

Therefore it is trivial that the internal data structure will have fields to accommodate these associated quantities. You will realize that a field named `Xhashlink` exists. In our implementation identifiers are chained together through this field, provided they belong to the same hash-bucket. The hash-number of an identifier is determined by summing up the ascii codes of its printname characters and then taking this number modulo 128. So there are 128 number of hash-buckets. A new created identifier, upon the decision of his hash-number is chained into the link of the identifiers with the same hash-number (we call this chain a hash-bucket) via this `Xhashlink` field. The entry points of these buckets are kept in an array named `hashtab`.

```
struct Xid { char Xtype;
             unsigned Xisinheap : 1;
             unsigned Xisglobal : 1;
             unsigned Xisfluid  : 1;
             unsigned Xisfunction : 1;
             unsigned Xisinoblist : 1;
             unsigned Xisdclfluid : 1;
             struct Xid *Xhashlink;
             PSEXP Xvalue;
             PSEXP Xproplist;
             char *Xpname; } ;
```

```
struct Xid { char Xtype;
             char Xattr;
             struct Xid *Xhashlink;
             PSEXP Xvalue;
             PSEXP Xproplist;
             char *Xpname; } ;
```



3.3 Strings

Are enclosed between double quotes. Any character may be used in a string. A double quote should be written as two consecutive double quotes. The following are examples of valid strings.

```
"abcdefghijklmnopqrstuvwxyz"
"that's"
"blabla!"
"He said ""LISP"""
```

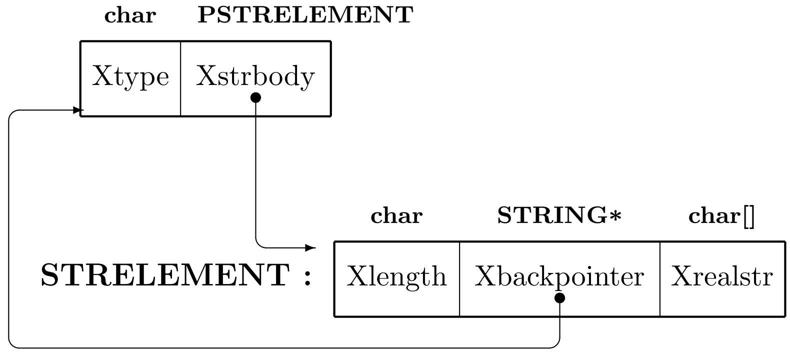
3.3.1 Implementation

There is nothing much to say about string implementation. The corresponding structure holds a pointer to a string-element. The string-elements are allocated from a pre-allocated (either static or dynamic, depending on what the compiler flag `DSTACK` says) byte array. It is something very common that due to garbage collection some gaps may occur in this array. So the famous problem of fragmentation arises.To solve this problem a Compactification is performed on this byte array, when it is necessary. The back-pointer that one observes in the string-element serves for this purpose. The last field of a string-element is declared as you see to be a character array of size 254, which is of course nothing real. It will never be the case that such a string-element is created using the **C** systems variable creation mechanisms. This structure declaration will always be used as type casting. So a good controlled pointer that points to the first free position in the string-byte array (where all of the string elements live) will be type casted to be a STRELEMENT and will be advanced such an amount that the actual string will exactly fit into the place just following the back-pointer. So it will also be possible to index the actual string bytes through the field identifier (which was declared to be an array) `Xrealstr`, cute, isn't it?. For the IBMPC implementation the string-byte elements are casted with a structure that we named `struct Xstrelmnt` . But as this code was ported to other computers this casting proved itself to be non-portable. The reason was that some (non - 80x86) processors (like the 680xx, or the SPARC) did not accept pointer indirections if the pointer is sitting in the memory at an address that is not zero (modulo 4). For these cases the string-byte array is not type casted but the fields are accessed (harshly) by movmem calls. Please note that the underlying data-representation is exactly the same in both cases. If you have a (80x86) processor then you may turn on the IBMPC flag, so the fields will be accessed by type-casting and (hopefully) a faster code will be generated. Below is the picture for the IBMPC case. For the non IBMPC situation consider the second picture.

IBMPC case

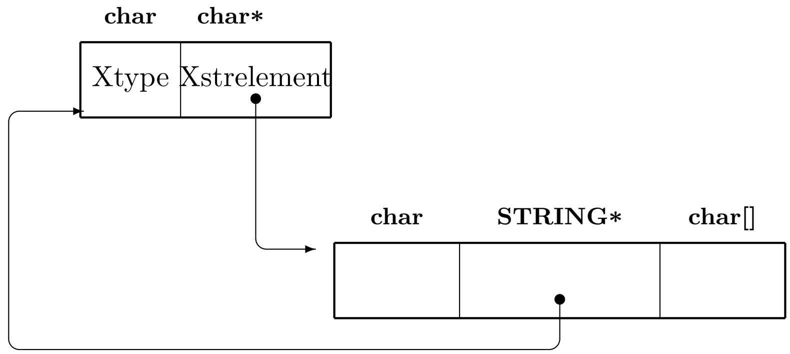
```
     struct Xstring { char Xtype;
                      struct Xstrelmnt *Xstrbody; } ;

     struct Xstrelmnt { char Xlength;
                        STRING* Xbackpointer;
char Xrealstr[254]; } ;
```



non - IBMPC case

```
struct Xstring { char Xtype;
                 char *Xstrelement; } ;
```



3.4 Integers and Big Integers

Have the usual regular gramer syntax: $[+ -]$**digit**$+$. Normal integers (or short LISP integers) are implemented as `long int` types of **C**. This implementation enables arbitrary size integers and their arithmetic. The system is completely aware of all the arithmetic and the internal type change from integers to big integers (which are implemented as a kind of link list) is transparent to the user. All this type of internal type conversions are done automatically.

3.4.1 Implementation

```
struct Xinteger { char Xtype;
                  long int Xintval; } ;

struct Xbig { char Xtype;
              long int Xintval;
              PSEXP Xcdr; } ;
```

char	**long int**
Xtype	Xintval

char	**long int**	**PSEXP**
Xtype	Xintval	Xcdr

3.5 Floating Point Numbers

Are implemented as doubles of **C**. In syntax they contain a decimal point, optional sign and exponent.

```
12.
3.1415
-.5
+12.3e-123
5.E7
```

are floating point numbers, but not `2e3` (2.e3 should be used).

3.5.1 Implementation

```
struct Xfloating { char Xtype;
                   double Xfloval; } ;
```

char	**double**
Xtype	Xfloval

3.6 Vectors

Are one dimensional arrays of S-expressions. Although they are certainly composite objects, they are considered to be atoms in LISP. Every vector has a fixed nonnegative upper bound, and its indices range through 0 to this upper bound (inclusive). They are written between brackets, where the elements are separated by commas, like

```
[elt0,let1,...,eltn]
```

3.6.1 Implementation

```
struct Xvector { char Xtype;
                 int Xupbv;
                 PSEXP *Xvectelts; } ;
```

char	**int**	**PSEXP∗**
Xtype	Xupbv	Xvectelts

3.7 Function Pointers

Machine code routines, either produced after compilation, or parts of the interpreter itself are considered to be LISP objects. More precisely, the interpreter uses function pointers to access these dynamically. These can be part of an S-expression, or can be printed. They are denoted by the octal address of the function enclosed between number signs #.

3.8 Functions and Function Definitions

Lisp functions are divided into two main classes. One includes functions defined as **lambda expressions**, which are interpreted, the other includes compiled functions and functions that are defined in the LISP system, which are accessed via function pointers. Arguments are passed to interpreted functions by lambda convention (saving the current value of the lambda dummy-argument-identifier onto an internal stack (the 'alist' stack) and temporarily assigning the current argument to the dummy-argument-identifier, as value), to compiled functions the arguments are passed through the **registers** (internal defined static memory locations). Functions are further classified by the argument passing mechanism peculiar to them:

spread functions have a fixed number of arguments (with a maximum of 16) which are bound to lambda parameters on one-to-one basis. Giving incorrect number of arguments to these results in a mismatch error.

nospread functions may have a variable number of arguments,but have a single formal parameter, which is bound to the list of the actual arguments. (Obvious examples are the LISP functions as `and, or, cond, list`

eval type functions receive their arguments pre-evaluated in contrast to

noeval type functions which receive their arguments unevaluated (as in the case of `and, cond, quote`)

All of the possible eight combinations are implemented as follows:

	spread	**nospread**
eval	expr subr	lexpr lsubr
noeval	nexpr nsubr	fexpr fsubr

expr, fexpr, nexpr, and lexpr's are lambda expressions, subr, fsubr, nsubr, and lsubr's are function pointers.

In addition to these there is one more type: 'macro'. A function of type macro has a single formal parameter which is bound to the entire expression invoking it, and the value returned by it is reevaluated.

3.8.1 Implementation

The `Xfnc` field is a function pointer which contains the entry point of the compiled function. Since the arguments may vary in number, somehow the system must know this. The second field `Xargno` serves for this purpose.

```
struct Xfpointer { char Xtype;
                   char Xargno;
                   int (*Xfnc)(); } ;
```

char	**char**	**int (*)()**
Xtype	Xargno	Xfnc

3.9 The Lexical Analyzer

The lexical analyzer is not a DFA. At the development phase the **lex** program of the UNIX system was used, but later, the code this system has generated was found to be large in size. Therefore, a hand coded functional equivalent has been substituted. This is the reason why most of the functions have names that are used by the **lex+yacc** utilities. The hand coded lexical analyzer, is not as fast as a DFA, but the difference in timing is very minor.

The design philosophy of the lexical analyzer is consulting a series of functions each of which are sensible to a certain lexical pattern type. For example, the function `isid()` is sensible to identifiers. If the current parsing pointer `curpos` points to a position where an identifier starts, then upon a call to `isid()` three actions will be taken by this function:

1. The input string is scanned until the end of the identifier is found.
2. `curpos` is advanced to the first unused character.
3. A return value of **1** is passed back to the caller.

If the pattern that was currently pointed by `curpos` was not an identifier then a call to `isid()` would not alter the value of `curpos` and the return-value would be **0**.

The main function of the lexical analyzing process is `yylex()`, when this function is called, functions like `isstring()`, `isid()`, `isfpointer()`, etc. are called until one of them succeeds (that means return a **1**). When this happens the matched pattern is copied into the array `yytex` by a call to the function `storetext()`. If the parsed pattern is a delimiter then a global variable `delimflag` is set to **1**.

Chapter 4

The Dynamic Structure

4.1 Memory Management

page is the main storage unit of this LISP implementation. LISP data items (SEXPR) are hold in linked memory pages each of which are of a constant size. This size PAGESIZE is a multiple of the least-common-factor of the sizes of possible data structures (sizes of which are stored in the sz array) that corresponds to a LISP object. Pages are allocated dynamically from the system memory by using malloc. Each page is a structure of three elements (defined in **type.l**):

```
     struct page { struct page *nextpage;
                   char *free;
   char pagebd[PAGESIZE]; } ;

     typedef struct page PAGE, *PPAGE;
```

where

nextpage Pointer to the next page of the same page type.

free Free cell pointer of the link list of the free cells of the page.

pagebd Space for LISP data types (SEXPR).

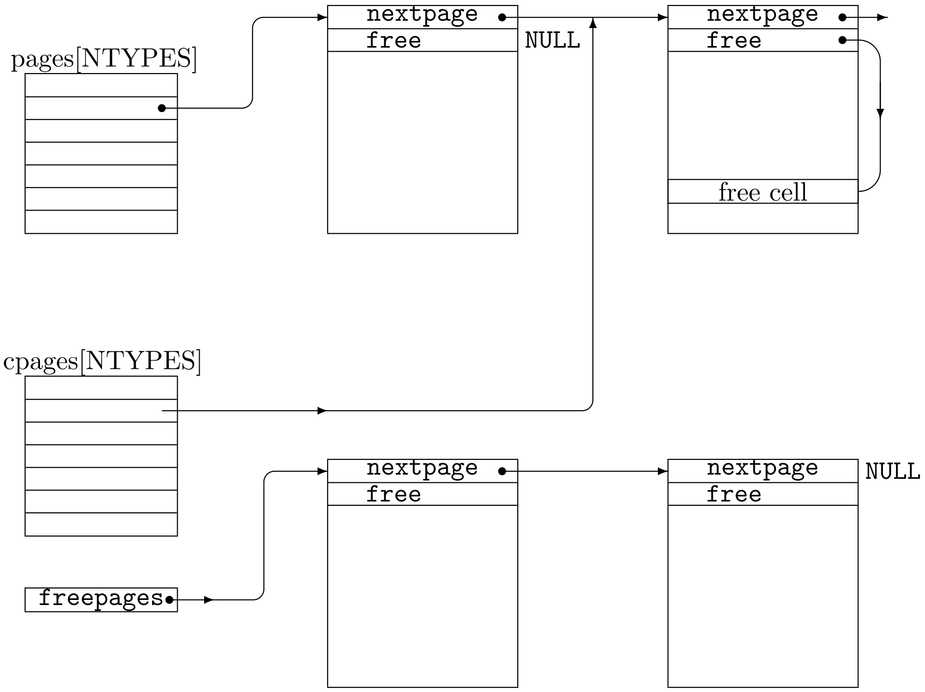


Figure 4.1: **Typical page organization.**

PAGESIZE Size of the page.

PAGE Page structure type.

PPAGE Pointer type to a page structure type.

Following variables are also used in memory management:

```
PPAGE pages[NTYPES], cpages[NTYPES];
PPAGE freepages;

unsigned sz[NTYPES] = { sizeof(PAIR),
                        sizeof(ID),
                        sizeof(STRING),
                        sizeof(INTEGER),
                        sizeof(BIG),
                        sizeof(FLOATING),
                        sizeof(VECTOR) };
```

NTYPES Number of dynamically generated cell types (macro). It is 7 in this implementation.

pages Array of pointers to the first pages of different types. Used by garbage collection.

cpages Array of pointers to the active pages of different types. Allocation of new items are done from these active pages.

sz Array which contains the sizes of cells.

freepages Pointer to a page. It points to the first page of the link list of completely empty pages. These pages are generated and linked by the garbage collector if all the cells in page are unbounded cells. Free pages are untyped pages.

Allocation of LISP data items are done by two functions, `zalloc` and `zgetpage` in file **lisp-zfn.l**:

zgetpage: `PPAGE zgetpage(unsigned tp);`

`zgetpage` is the function for allocating page of given type `tp`. Each time it is called it returns a pointer to an empty page which contains the linked list of `PAGESIZE/sz[tp]` number of free cells of type `tp`.

Linking is done by `zgetpage` and first free cell address is put into free pointer of the page. nextpage pointer of the page is always set to NULL. `zgetpage` returns a page from the free pages list, if there are any pages available (`freepages` non-NULL). Otherwise it allocates new page from the system sources if possible and otherwise it returns NULL.

zalloc: `PSEXP zalloc(unsigned tp);`

`zalloc` is the lowest-level function for allocating cell of given type `tp`. Each time it is called it returns a pointer to a fresh cell. It returns a fresh cell, if

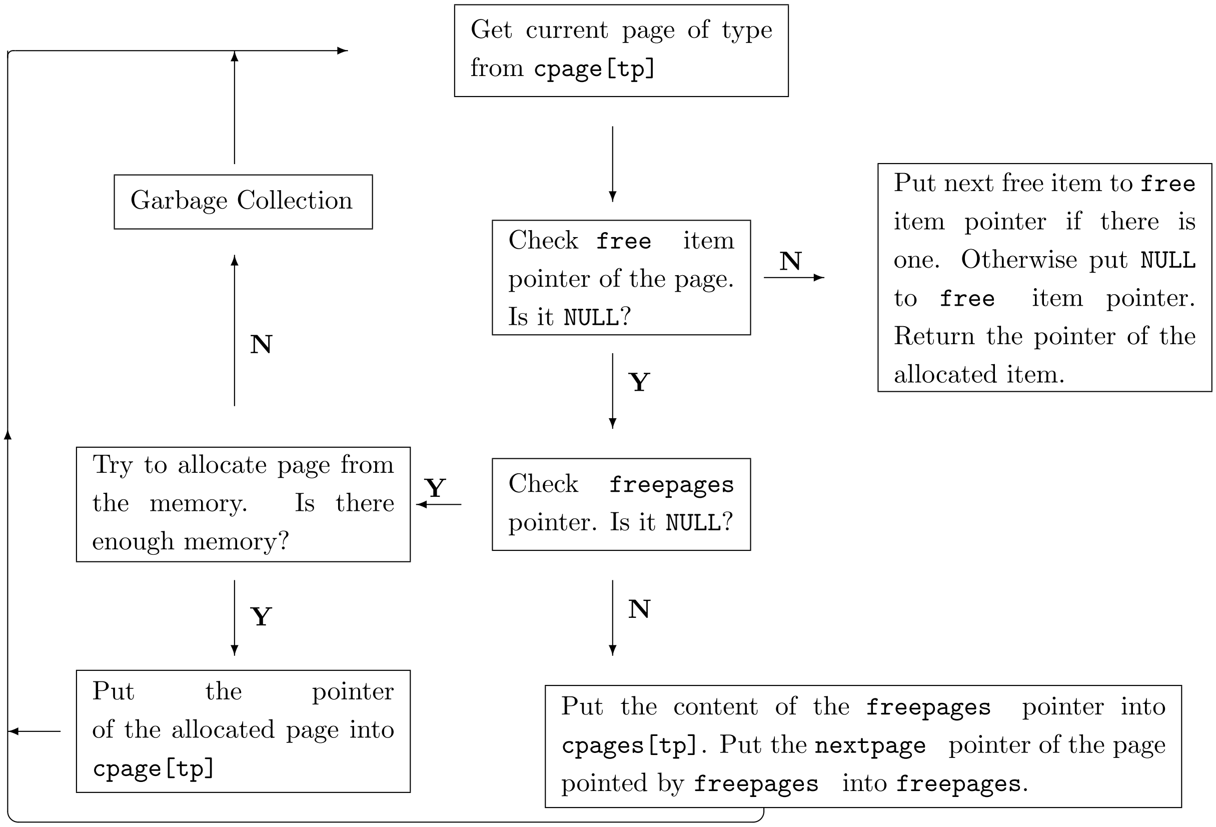

Get current page of type from cpage[tp]
Check free item pointer of the page. Is it NULL?
N
Put next free item to free item pointer if there is one. Otherwise put NULL to free item pointer. Return the pointer of the allocated item.
Y
Check freepages pointer. Is it NULL?
Y
Try to allocate page from the memory. Is there enough memory?
N
Garbage Collection
Y
Put the pointer of the allocated page into cpage[tp]
N
Put the content of the freepages pointer into cpages[tp]. Put the nextpage pointer of the page pointed by freepages into freepages.

Figure 4.2: **Item allocation algorithm of type 'tp' (roughly).**

there are any cells available on active page (free pointer of page non-NULL). Otherwise, it attempts to allocate a new page by calling `zgetpage` and if `zgetpage` returns a non-NULL pointer to a fresh page it returns the first cell in that page else it performs garbage collection which constructs linked lists of reusable cells for each type.

Other allocation functions are `zcalloc` and `zsalloc`:

zcalloc: `PCHAR zcalloc(unsigned n);`

Performs space allocation for identifier print name of size '`n`' characters from print name space. Returns pointer to the allocated space.

zsalloc: `PCHAR zsalloc(unsigned n);`
Allocates space in string space for a string of '`n`' character long.

4.2 The Stacks

As it was explained in the introduction, this implementation is not a stack oriented one. That means the argument passing in evaluations are not performed via a stack but through some fixed memory locations (registers). This fact, should in no means be interpreted as no stack is used at all. Indeed, there exist basically three different stacks in this implementation.

The System Stack This stack is provided by the **C** compiler and is the place where all the **C** local variables —or so called **auto**'s— are created, arguments are passed on, the functions return addresses are pushed on. The system is designed not to interfere with this stack. We tried to refrain passing LISP objects as arguments on this stack, and if we have done so, it was in those cases where we were absolutely sure that no garbage collection marking would be necessary for those entities.

The SEXPR Stack In fact this is not a stack where the S-expressions are pushed on but merely a stack where **pointers to S-expressions** are pushed on. A language like LISP, that provides recursion must definitely have a stack where domain entities of it can be pushed. It is implemented as an array of `PSEXP`'s and is has its start pointed by the variable `zstackptr` and its current top pointed by the variable `zstacktop` . There exist define macros that serve to load to/from this stack from/to the registers and macros to move the top-pointer: (`kalloc(), ksetn(), kloads(), ksets()`). They will not be explained here in details, the code is self-explanatory. It is for sure that a garbage collection marking is performed from this stack. The array that is used for this purpose is either allocated as a static array at compile time or is allocated dynamically from heap at the initialization of LISP. The way chosen depends on the the state the compiler flag `#DSTACK` is set to, at compile time of the LISP system. If it is to be allocated dynamically then the default size can be overridden by a command line option.

The ALIST stack This stack is used for the **lambda** and **prog** bindings. Especially in the interpretive environment the user defined lambda expressions and prog's are of intense use. Every time a identifier is used as a lambda variable or a local variable for a prog body the LISP system has to save the old value of that identifier, in order to be able to restore it upon the exit. The implementation of the stack is realized as an array of data structures (named `ALISTENT`) of the following form.

ID∗	**PSEXP**
alistid	alistval

Similar to the SEXPR stack it is possible to allocate this stack in the static area or dynamically depending on the same compiler flag. The start of this stack is pointed by the variable `alist` and the top is pointed by `alisttop`.

4.3 Garbage Collection

During the course of a computation, contents of cells which were taken from free space list often becomes unnecessary. They are garbage. In general it is difficult to know exactly which cells are garbage. The responsibility for reclamation is therefore undertaken by the LISP system itself. The fundamen tal assumption of garbage collection is

> *at any point in a LISP computation, all cells which contain parts of the computation are reachable from a fixed set of known cells or registers.*

Garbage collection consist of two phases: *mark* and *sweep*. The first phase of the garbage collector, called the marking phase, marks all of the LISP data structure which is currently active. By definition, a cell (any LISP data object) is active if it is reachable from the registers, stack, or identifier table. An identifier which is not reachable in this way may still be active if it has a non-NULL `value` or a non-NIL `proplist`.

In terms of our implementation, we mark from the registers, from the identifier list (its values and properties), from the SEXPR stack, from the `urwelt` ar ray and from the ALIST stack. During the mark phase each cell which is active is marked (by using `putmark` macro) by marking its `type` field. The following macros (file **type.l**) are used in marking and sweep phases:

```
#define putmark(x)  (type(x) := 0x80)
#define clrmark(x)  (type(x) &= 0x7f)
#define marked(x)   (type(x) &  0x80)
#define gtype(x)    (type(x) &  0x7f)
```

A structure might be referenced several times in the marking process, since we allow multiple reference to structure. We must take this into account since, though naive marking of an already marked structure is at wasteful, it is fatal if the structure is self-referential. Marking of structures are done by `zmark` (file **lisp-zfn.l**). Once all active structure has been marked, any unmarked cell can be treated as garbage and may be returned to the free storage for reuse. This is done during sweep, the second phase of the garbage collection.

The sweep phase proceeds linearly through pages, collecting all those cells which have **not** been marked. These unmarked cells are chained together and `free` pointer of the page is set to this chain. At this stage, unmarking of the marked cells are done also (by `zcollect`). If all the cells in some page are unmarked then this page is removed from the linked list of pages of a specific type and chained to the `freepages` list.

There are five garbage collection functions in the file **lisp-zfn.l** and are described below.

zmark: `void zmark(PSEXP x);`

`zmark` is responsible for marking cells during garbage collection. It takes an `PSEXP` as an argument and it marks according to its type. If the SEXPR pointed at has already been marked, `zmark` simply returns, since we are assured that any cells further down the structure have already been marked. If object is a vector, it marks the vector itself and all objects to which the entries in the vector point. If it is a dotted pair, it simply marks all the cells pointed by the car and cdr pointers. Any other object is assumed to be elementary (i.e., contains one cell only) and is marked by marking its cell. An identifiers is marked only if it is in a dynamically allocated page. Compile time identifiers are allocated separately in static locations and therefore they are not marked.

zcollect: `int zcollect(PPAGE p, int tp);`

The sweep phase routine `zcollect` is responsible for collecting all those cells which have **not** been marked in the page pointed by '`p`'. These unmarked cells are chained together and the `free` pointer of the page is set to this chain. Unmarking of the marked cells are done also. Returns *one* if all the cells in the page are unmarked. Otherwise returns *zero*.

zcompactatom: `void zcompactatom();`

Identifiers print name space `(char *)startpns` compactification algorithm.

zcompactstring: `void zcompactstring();`

String space `(char *)startstr` compactification algorithm.

zgarbage `void zgarbage(unsigned tp);`

Main algorithm of the garbage collection. Mark phase is done first. If necessary, compactification is performed on identifiers print name space `startpns` and string space `startstr` after the mark phase. Then the sweep phase is performed. If `!*gcflag` is non-NIL, some statistical information about the garbage collection is given to the user.

4.3.1 Pair space compactification

A compactifying garbage collector is under development. The code distributed includes this garbage collector as well. The way to turn it on is to issue the command

```
(setq !*gcflag 2)
```

Please note:

For the time being the compactification phase is **NOT** bug free. But, the normal (non-compactifying) garbage collector, which is in use as default, works error-free.

Chapter 5

Compilation

5.1 General

In this section our intention will be to describe two actually different subjects together. One is the prescription, or better to say, a case dependent set of descriptions

- to perform an actual creation of a LISP system,
- to start with some LISP program and create a compiled version of it.

The other is about the internal implementation of this compilation. So, those who don't have a burning desire to understand every bit of what is going on can simply avoid reading that part. But frankly speaking, it would be much better if one understands the underlying scheme of implementation, since the process of compilation, for the sake of portability, is not very straightforward. This detailed, “why is it so” type of explanations are put in italic, below.

5.2 Bootstrapping LISP

“Bootstraping LISP” means “Creating a pure STD-LISP interpreter from scratch”. Here is how to do it:

1. Put all the files with extension ”**.l**” into the same directory. Put also the **C** programs **cr1.c** , **cr2.c** , **crc.c** , **cri.c** , **size.c** , **crfile.h** into the same

directory.

Each of these **C** *programs will create a part of the* **C** *source, those which will later be compiled by a* **C** *compiler and presumably will be linked to get the actual executable program.*

cr1 *Will create* **lisp1.c** *, the part of the LISP system that contains the static declarations.*

cr2 *Will create* **lisp2.c** *.* **lisp2.c** *is the source of all the LISP system functions, the memory manager, the garbage collector, the I/O routines etc. Unless these definitions are not changed you do not have to run* **cr2** *more than once even if you later perform a LISP ⇒* **C***compilation of any LISP program. (Please note that this is not so for* **lisp1.c** *, when you later perform a LISP code compilation than* **cr1** *has to be fired again to reproduce* **lisp1.c** *.*

crc *Actually this program and the below described program will only be used if you will compile a LISP program that you have written. Details will be explained in the section entitled "Compiling a LISP code". For the time being just know that this program will attack the output of the LISP ⇒* **C** *compiler and produce a true* **C** *source that will be C compilable.*

cri *Just as* **crc** *this program is also used only if a user written LISP code is compiled. It produces a LISP initialization file, in which the non-compilable, user defined LISP expressions are stored. (e.g. quoted lists, strings, atom names).*

2. Change the directory information in the **crfile.h** file.

   We hear you groaning, but this was necessary, again, to keep true portability, since systems with different operating system have different directory and file access strings. (The trivial example is UNIX versus DOS, one has a slash, the other has a backslash for directories)

3. If necessary edit the file **flags.l** . In this file you have various compiler flags for the **C** compilation. They are explained below:

TRACEABLE If you perform a traceable and/or redefinable code generation (see the relevant section on this topic) for user defined functions, this **C** flag if zero, tells the **C** compiler to omit the traceability or redefinability. So, the value of this flag is only important if you perform a LISP ⇒ **C** compilation.

DSTACK Of course any LISP has to have a stack, that is for sure. The DSTACK **C** flag stands for 'Dynamic-STACK' depending on which the stack is created in the static data area or alternatively allocated at run time as a dynamic memory.

BITF If this flag is non-zero then the code generated will make use of the bit-sub field concept of **C**. If the flag is set to zero no such code will be generated.

The bit fields come into account in the tag of a dotted pair, where various informations have to be hold about that dotted pair (e.g. one bit is for garbage collection marking). It is cute and elegant to use in this case bit-fields, but unfortunately some compilers generate a terrible code for bit-field accesses. (e.g. some of them perform a number of shift right operations to get the desired bit to the least significant bit position and then test it, which is awful!). But another alternative is to use just a byte and mask the bits of it. Then setting will correspond to a bitwise OR, and the testing will correspond to bitwise AND operation among the tested byte containing the tested bit and the selection mask byte.

TURBOC If you are using Borland's Turbo **C** compiler this file will serve for better code generation.

IBMPC If you are working on an IBM-PC or compatible, then the higher ascii codes are set in a specific manner, if this flag is non-zero the generated **C** code of the LISP interpreter will make use of these ascii codes.

Furthermore the IBM-PC is using the 80x86 μP-chip which allows pointers to sit at addresses that are not zero (modulo 4). The code makes use of this (non-portable) feature if the flag IBMPC is set to 1. If you don't have an 80x86 or if you get errors like (bus-error, segmentation fault) turn this flag to zero.

4. Compile all the **C** files, that you have copied.

5. Execute the **size** program you have just compiled. This program will propose a value for the constant **PAGESIZE** . That constant lives in the file **types.l** . Set in this file `#define PAGESIZE` to an integer multiple of the proposed value.

   data structures may have different sizes on different **C** *compilers, because of the underlying hardware. Therefore it is necessary to determine how your compiler acts. We need this information because this LISP implementation does not simply use* `malloc` *or* `calloc` *when it has to create a LISP data object, for further details see the chapter on "The Dynamic Structure".*

6. Execute **cr1** and **cr2** in order to generate **lisp1.c** and **lisp2.c** , respectively.

7. Compile **lisp1.c** and **lisp2.c** with the **C** compiler.

8. Link them, with the appropriate libraries in order to obtain an executable program which is your STD-LISP interpreter.

   what the libraries will be is something system dependent, so since you got your hands on a **C** *compiler, we have to assume that you are fully capable of creating an executable code after a* **C** *compilation.*

5.3 Creating LISP with Compiler

The word "Compiler" in the name of this section refers to the LISP ⇒ **C** compiler. This compiler is actually a LISP program. It constitutes of two functional parts. One is the part that is proposed as *A Portable LISP Compiler*, by M. L. Griss and A.C. Hearn. This part generates a compiled code in terms of some macros that correspond to certain actions of the underlying LISP system (e.g. calling a compiled function, pushing some LISP object on stack, accessing the stack, etc.) It is actually portable, because what is needed is only a secondary process, done again in LISP, that will take this macro involving code and looking at it produce a code in the target language (in our case **C**). One has to point out that the *A Portable LISP*

Compiler assumes an underlying register implementation. This is another aspect which biased our decision towards a register implementation rather than a stack one. We name the second part, that converts the macros to the **C** code, as **lap** . In our implementation **lap** performs some code refinements too. The lisp source of **lap** lives in the file **lap.lsp** distributed. So, a full bootstrapping of the compiler including LISP is nothing else then performing a LISP code compilation. But how to get the first working compiler? It is trivial that this ... *Egg*⇒*Chicken*... *Egg*⇒*Chi cken* ... chain has to start somewhere. One answer is: You get a LISP interpreter compiled (the previous section) you load the LISP code for the compiler, namely the file **compiler.lsp** , you load the LISP code for the lap, namely the file **lap.lsp** and you get an interactive working compiler, you supply this interactive compiler the same LISP sources of the compiler+lap, by a **compilefile** LISP function call (see below for the arguments) and then you wait for ages to get the first "*Chicken*" out. We did this process. For your benefit we have not thrown away the **C** code that came out to produce this "*Chicken*". In the source supplied, you will find a set of files collected under a subdirectory named **compiler** . The files included in this set are the outcome code of the compilation of the compiler (those having names that start with the letters **comp** and have no file extensions. In order to get a working version of LISP, having the compiled compiler present, follow the steps:

1. Copy the set of files, in the subdirectory **compiler** , to the same directory you will/have put all the files with extension "**.l**" of the previous section.

2. Regardless of whether you had done it already or not, run **cr1** in order to create **lisp1.c** .

   We do have to recreate **lisp1.c** *because this file will contain some static data which will be created looking at some of the files of the* **compiler** *package*

3. Perform those steps of the prescription of the previous section, which you have not done by now up to the 6 th step, including it.

4. Run the **crc** program. This will create the necessary **C** sources related to the compiler part.

5. Compile all the **C** sources which are created by **crc** . (attention: **crc** will create more than one **C** source)

   If you have not missed anything, now you must have to hand the object codes for **lisp1** *,* **lisp2** *and those objects which are the result of the compilation of the* **crc** *produced* **C** *sources.*

6. Simply link all of them to obtain the LISP version that has the compiler present.

   UNIX users: You can do all the compilation and linking of **lisp1.c** , **lisp2.c** , and the **lispc#.c** files just in one line, namely:

   ```
   cc lisp*.c -lm
   ```

   -lm *links in the standard math library*

7. Run the **cri** program. This will create an initialization file, with the name you supplied as the second command line argument.

   If you omit this the default name **LISP-INI** *will be given. If you decide to use the second command line argument don't forget to submit the first one which is:* `comp` *for the compiler compilation, and some other generic name you once gave (see the section "Compiling a LISP Code") otherwise*

   At run time, an environment variable should be set to the name of the initialization file (including the full path), and furthermore the LISP that you are created shall be informed at run-time about this enviroment variable by a command line option **-E** *(for details see relevant explanation in this manual). If you do not use the* **-E** *option then the LISP you compiled will look for an environment variable with the name* **LISPINI** *.*

   If you forget to set the environment variable to point to the initialization file, and the initializiation file is not in the current directory then the LISP that you have created, will abort, at run-time, with an error indicating that the initialization file is absent

5.4 Compiling a LISP Code

When you have developed a LISP package, usually it is desirable to get it compiled. This has advantages commonly known.

- Your program will speed up by a factor of 10-20.
- The code will no more be portable and understandable. (Yes, indeed. This actually is mostly the case for commercially customized applications!)
- The size will eventually shrink.

What follows is the prescription if you are at this stage. (i.e. you are happy, your program works, and its time to finalize the job)

1. Execute the LISP that has the compiler present. How to create such a LISP was the topic of the previous section. We suggest that you run it at least with the following parameters:

   ```
   LISP -P20000 -S10000
   ```

   Actually, you can do this job interpretatively, by loading the LISP code for the compiler (that is also present in the software supplied). But this will be painfully slow, therefore it is not very logical, except one case: Due to some reason you cannot perform the compilation explained in the previous section "Creating LISP with Compiler"

2. Type-in and evaluate the following LISP expression in order to start the compilation.

   `(compilefile "`*lispfile*`" '`*genname*`)`

   Here *lispfile* is the name of the file that contains your LISP program (or the starting part of it). *genname* is a generic atom which will serve in naming of the output files. All outcome files of this compilation will have names starting with *genname*.

   So the "genname" for the files that you have used in the creation of the compiler (in the previous section) was `comp` *. The output files are:*

gennameU **:** *This file contains all the constants of the LISP program you have compiled.*

gennameE **:** *This file contains all the non compilable, LISP s-expressions of your program. Later on* **cri** *will attack this file and generate the "initialization file".*

gennameNn **:** *Here the last* n *is an integer. All these files contain the procedure names of the LISP program that was compiled.*

gennameCn **:** *Here the last* n *is an integer. These files contain the C source of the compiled LISP procedures.*

gennameXn **:** *Here the last* n *is an integer. These files contain the names of externally called procedures. So, normally, the union set of the contents of* gennameXn *shall be a subset of the union set of the contents of* gennameNn *files. That means no procedure shall be used without having it defined somewhere. But, this compiler is cleaver to accept those undefined procedures by just giving a warning and generating such a code that this procedure is definable at run-time, cute eh?. Best to understand this feature is to test it.*

3. For the successive inputing of other LISP source files just give their names as strings (included in double quotes) or as quoted atoms (if they have no extensions of course). To declare the end of the compilation enter `end` to finish the compilation (as any bright man will immediately recognize, it is therefore only possible to get a lisp source that lives in a file named **end** , compiled, if one surrounds the **end** with (double quotes)).

4. Perform all the steps of the previous section starting with the 2. step, with one change: Executions of **cr1** , **crc** , **cri** shall have command line arguments if the *genname* you used is different then **comp** . You have to supply this *genname* as command line argument to the mentioned **cr** programs. An example:

   If your *genname* was **rlisp** , for example, call **crc** as

   ```
   crc rlisp
   ```

Furthermore if you want that **cri** names the initialization file that it is creating other than **LISP-INI** , then supply this name as the **second** command line argument.

```
cri rlisp MY-LISP-INI
```

If you have forgotten to do this and **cri** has created the initialization file with the default name **LISP-INI** , just you can rename it as you like.

5.5 Traceable/Redefinable Code Generation

This compiler has a feature which is not found in many LISP compilers. It is possible to order the compiler to generate a code for the user defined and compiled LISP functions, such that they (or a subset of them) will be redefinable at run-time. Furthermore it is also possible to have a code generation that enables tracing of compiled user defined functions.

If you are performing a LISP ⇒ **C** compilation then you have the choice to set the flag `!*traceable` to a non-nil value (by default it is nil) before you start the compilation with the `compilefile` function. If `!*traceable` is set then the **C** code generation of all user defined functions is altered to have an overhead code that performs a check on the global environment (which is set by the LISP function `traceable` , and described below), and if this environment says at that moment a trace is desired a trace-print out will be done. The code change that is done when `!*traceable` is non-nil, is performed by an insertion of a **C** macro. Now if `TRACEABLE`, a **C** define macro in the **flags.l** file, is set to zero then the **C** compiler will omit this inserted overhead. But if `TRACEABLE` is set to 1 then the compilation will be done so that the user defined functions are traceable. Therefore, it is always clever to perform the LISP ⇒ **C** compilation with `!*traceable` set to non-nil and then use or not this code alternation by controlling the **C** define macro `TRACEABLE` in the **flags.l** file.

Similar to `!*traceable` you can set the flag `!*rdable` which will generate a **C** code for user-defined functions that allows them to be redefined at run time in such a manner that even the compiled functions will use this new definition. (Normally, for most of the LISPs this is not the case, of course it is trivial that in any LISP you can redefined a compiled function, even you can replace a new definition for **car** , but this will only effect the interpretive calls to that function and not the calls inside the compiled code, that means if you have compiled two functions **A** and **B** and **A** is calling **B** , then you redefine **B** at runtime, from that moment on all interpretive calls to **B** will use the new definition but **A** will still work in the old fashion, making use of the old definition of **B**).

If you have set `!*traceable` this automatically implies all the user defined functions to be redefinable, so you do not have to set `!*rdable` in addition.

Although it is not possible to define a subset of functions to be traceable it is not so for 'redefinability'. It is possible to declare a set of functions to be redefinable. This is done by flagging their names with the atom `rdable`, prior to compilation (i.e. issuing the `compilefile` call).

At run time the control of tracing and redefinition is done by the function `traceable`. The usage follows:

- `(traceable nil)` Disables tracing. Furthermore Disables the usage of the redefinitions of the functions that were redefinable.

  Redefinable or traceable code has some computational overhead, that is for sure. But when the use of the redefinitions is enabled an extensive check and process is performed, it could be the case that although you have compiled with redefinition ability, you don't want to use it at that instance, or you do not want to trace, in those cases it is desirable to issue `(traceable nil)` *which will minimize the computational overhead (in this case on a conditional on a global variable will be processed).*

- `(traceable 0)` Same as above.

- `(traceable t)` Enables the trace of **all** traceable functions to any trace depth. If you set this option, regardless of the presence of any `'trace` flagged function, all of the traceables will be traced.

- `(traceable` *number*`)` Enables the trace of only `'trace` flagged functions to the call depth which is said to be *number*. *number* is an integer greater than or equal to 2. The calls of untraced functions will be counted, too.

- `(traceable 1)` Enables the use of the redefinitions for the redefinable functions. If this is not enabled, the code will execute much faster.

- `(flag` *list-of-fn-names* `'trace)` Is used in conjunction with `(traceable` *number*`)`. Defines a set of function names to be traced. Executing `(traceable nil))` will not remove those flags.

5.6 Altering the size of compiler generated C files

If you have gone through the compilation process of a LISP code, you must have observed that although you submitted a single LISP source file, the LISP ⇒ **C** compiler decides to create a number of files each of which is of a certain size. Why is this so?

Some **C** compilers cannot handle big sources, they overflow in symbol tables etc. So you must subdivide the code into smaller modules and get them all compiled. This is not a trivial task since the small modules export and import some **C** functions. The LISP ⇒ **C** compiler does this job for you.

The size of 'chopping' the generated **C** source is determined by the content of the LISP atom **max!-comp!-size** which is set to 2000 units. This atom is defined in **lap.lsp** which is distributed under the subdirectory **compiler** . Ofcourse you don't have do do a recompilation to set it to another value. You got two alternatives:

1. You set change it (for example to the value 7000) by a

   (setq max!-comp!-size 7000)

   prior to the **compilefile** call, in the run of the LISP that has the compiler present. Or

2. You simply edit the initialization file of the LISP with the compiler present. There in you will find a line:

   (setq max!-comp!-size 2000)

   Change it, as you like.

5.7 Compiler’s naming convention

Lisp functions are compiled (by the compiler) into **C** functions. A lisp function name is an identifier in which any printable character may occur, provided that the non-alphanumerics are prefixed with the escape character, the exclamation mark ‘!’. This, of course is not the case in **C**, so while mapping the lisp names to **C** names a conversion has to be made. Please note that this has nothing to do with the execution of the compiled code. Surely, as it is for all built-in functions of lisp, at the lisp level a named function is recognized through its name, which is an lisp identifier, actually a lisp atom having a function pointer sitting in its value field. This function pointer is generated by the **C**-compilation as to be a pointer to the related function, an entry point. So, after the **C**-compilation no trace remains about what the **C**-name of the compiled function was. But sometimes it is interesting to investigate the compiler generated **C**-code.

The naming rules in the lisp⇒**C** compilation of functions is as follows:

1. If the first character is alphabetic then it is capitalized.

2. If the first character is a digit (escaped) then it it is prefixed with an underscore ‘_’.

3. Any printable but non-alphanumeric is replaced by a pair of uppercase letters by the following convention:

`!␣` SP	`!!` XL	`!"` DQ	`!#` NB	`!$` DL	`!%` PS	`!&` AN
`!’` SQ	`!(` LP	`!)` RP	`!*` AS	`!+` PL	`!,` CM	`!-` MN
`!.` DT	`!/` SL	`!:` CL	`!;` SC	`!<` LT	`!=` EQ	`!>` GT
`!?` QS	`!@` AT	`![` LB	`!\` BS	`!]` RB	`!^` UP	`!‘` BQ
`!_` US	`!{` LC	`!|` OR	`!}` RC	`!~` TL		

4. If the name is flagged `special`, then a `1` (one) is appended at the end of the name. *(This is normally used for name clashes with the C library functions, some examples are open, read, write etc.)*

5. Automatic generated naming is provided, this is explained in details below:

There is no theoretical limit to the lenght of a function name. But this might not be so for a **C** compiler. Therefore a global variable is established to control the maximal length of a function name, which will be converted to a **C**-equivalent by the above stated conventions. Longer function names then the maximal length stored in this global variable '`!*maxfnamelen`' will be converted to automatic generated function names. Such names are cooked up by appending an incremented integer value (for each such new function) to the end of '`GenFun`'. By default '`!*maxfnamelen`' is set to `nil` which means no such automatic naming shall be applied.

If desired you may set it, (to 23 in the example below), before the '`compilefile`' call by an evaluation like:

```
(setq !*maxfnamelen 23)
```

5.8 Built-in atoms

There are a number of Built-in atoms, like `!*echo, !*raise` etc. They are defined as globals in the standard lisp report. As known, it is imposible to unglobal an atom. No such built-in function is existing. If one decides to change the way a built-in atom is defined, a **C**-compilation of lisp has to be performed. To make the necessary changes is quite easy. Built-in atoms are described in the file 'sysids.l'. In there following the name of the atoms you will see a sequence of 0/1's (actually 6 of them) they correspond to the fields:

Xisinheap; Xisglobal; Xisfluid; Xisfunction; Xisinoblist; Xisdclfluid

Now let us have decided to change the atom `!*echo` from being a global to be a fluid. Now, as you will recognize the second 0/1 field (that is the Xisglobal field) of the '`!*echo`' line in the file is 1 (so it is a global), to define it to be a fluid, reset this to 0 and set the following field (that is the Xisfluid field) to 1 i.e. set the pattern to be:

```
0 0 1 0 1 0
```

Bibliography

[1] J.R. Allen, *The Anatomy of LISP*. McGraw-Hill, New York, 1978

[2] J. B. Marti, A. C. Hearn, M. L.Griss, and C. Griss, "*Standard Lisp Report*" SIGPLAN Notices, ACM, New York, 14, No 10 (1979) 48–68

[3] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., New Jersey, 1978

[4] Martin L. Griss and A. C. Hearn,*A Portable LISP Compiler* Software-Practice and Experience, Vol. 11, 541-605 (1981)