Using LISP as an interactive database access language

Lukassen, R.J.P.B.

Award date:
1992

Department of Electrical Engineering
Eindhoven University of Technology
Design Automation Section (ES)

Using LISP as an Interactive Database Access Language.

by R.J.P.B. Lukassen

Master thesis.

Coach: ir. H. Fleurkens.
Under authority of prof. dr. ing. J. Jess.

November 1991 - June 1992

A

Abstract

ESCAPE is a tool, developed for fast prototyping of initial ideas at a high level of abstraction and developing a high level description of a design. It simulates the behaviour of circuits that are constructed with user-defined modules, which can communicate through nets.

There is a need for an additional tool that must provide programming capabilities as well as access to the ESCAPE data structures. With this new tool it must be possible to implement algorithms that operate on these data structures.

The tool developed is a LISP interpreter that uses a reference count mechanism to perform garbage collection in combination with a dynamic scope strategy. To access external data structures, such as the data of ESCAPE, the interpreter uses an interface that creates LISP objects of a special type. These special objects embed references to the external data. This interface is designed to be flexible, so that not only the data structures of ESCAPE can be accessed but also data structures of other programs, provided that they are written in **C**. The flexibility of the interface is created by using a publishing strategy. If the interpreter must access certain data, the definition of the type and a reference to the data structure must be published.

The interface also allows new functions to be added to the LISP interpreter, at run-time, to enhance the functionality of the interpreter. This enables the user to add functions that work on the interfaced data.

To test the interpreter, we have interfaced it with several programs. These test programs created data structures like binary trees and directed graphs, which were interfaced succesfully.

C

Contents

1. Introduction

1.1 Assignment 1
1.2 Outline of the tool 1
1.3 Choise and motivation 3

2. The LISP interpreter

2.1 Introduction 5

2.2 General structure

2.2.1 Read-Eval-Print loop 6
2.2.2 LISP-objects 7
2.2.2.1 The integer and real types 7
2.2.2.2 The string type 7
2.2.2.3 The symbol type 8
2.2.2.4 The array type 8
2.2.2.5 The bitvector type 8
2.2.2.6 The cons type, cons-cells and lists 9
2.2.2.7 The lambda- and predefined function type 10
2.2.2.8 The CObject type 10
2.2.2.9 The nil type 10
2.2.3 Reader module
2.2.3.1 Introduction 11
2.2.3.2 Preprocession of the input 12
2.2.4 Evaluator module
2.2.4.1 Introduction 13
2.2.4.2 Evaluation of symbol type objects 14
2.2.4.3 Evaluation of cons type objects 14
2.2.4.4 Evaluation of CObject type objects 16
2.2.5 Printer module 16

2.3 Scoping: symbol management

2.3.1 What is scope ? ... 18

2.3.2 Dynamic scoping versus static scoping ... 18

2.3.3 Implementation of dynamic scoping ... 21

2.4 Memory Management

2.4.1 Introduction ... 24

2.4.2 Methods for memory-management ... 26

2.4.2.1 Indirect management: stop-and-copy garbage collection ... 26
2.4.2.2 Direct management: reference-counting ... 27

2.4.3 Implementation of reference counting

2.4.3.1 Basic technique ... 29
2.4.3.2 Managing reference counters ... 30
2.4.3.3 Using local object-pointers ... 32
2.4.3.4 Allocating and storing of deallocated objects ... 32

2.5 Further reading ... 35

3. The interface

3.1 Introduction ... 37

3.2 Outline of the interface ... 37

3.3 Publishing types ... 40

3.3.1 Storing definitions of published types ... 40

3.3.2 Predefined publish types ... 41

3.3.3 User-define publish types ... 41

3.3.3.1 Publishing array-type definitions ... 41
3.3.3.2 Publishing array-type definitions ... 42

3.4 Publishing data ... 44

3.4.1 Storing published values ... 44

3.4.2 Publishing values as constants ... 44

3.4.3 Publishing values as pointers ... 45

3.5 Publishing functions ... 46

3.5.1 Storing published functions ... 46

3.5.2 Publishing functions ... 46

3.6 Accessing the published data 47

3.7 From read-access to write-access 48

4. Conclusions 51

5. Recommendations 53

Appendix A: an example of evaluating a user-function.

Appendix B: an example of an interfaced data-structure.

Appendix C: the interpreter in pseudo-code.

Appendix D: a list of publishmanager functions.

Bibliography

Glossary

F

List of figures

1.1 Integration of the tool 1
2.1 The main loop of the interpreter 6
2.2 An example of a cons-cell 9
2.3 An example of a list 9
2.4 The syntax accepted by the reader, in BNF 12
2.5 Evaluation rules for LISP objects 13
2.6 Scope in Pascal 18
2.7 Scope in dynamic scoped LISP 19
2.8 Definition of the `BINDINGSTACK` structure 22
2.9 Definition of the `BINDING` structure 22
2.10 Definition of the `LOCALSTACK` structure 23
2.11 Definition of the `OBJECT` structure 24
2.12a Example list (1 2 3) bound to x 25
2.12b The list after it is changed 25
2.13 Definition of the HEAPLIST structure 33
2.14a The freelist and heaplist just after allocation of a new heap 34
2.14a The freelist and heaplist just after deallocation of a used object 34

3.1 Place of the interface 38
3.2 The publication data base 39
3.3 Definition of the CTYPE structure 40
3.4 Definition of the CFIELD structure 42
3.5 Definition of the CPROP structure 43
3.6 Definition of the PUBLISH structure 44
3.7 Definition of the CFUNC structure 46
3.8 An interfaced binary tree 48

B.1 Definition of the node structure B.1
B.2 The published tree B.2

C.1 The main procedure, the read-eval-print loop C.1
C.2 The ReadForm function C.2
C.3 The EvaluateForm function C.3
C.4 The EvaluateCObject function C.3
C.5 The EvaluateCons function C.4
C.6 The EvaluateSymbol function C.4
C.7 The EvalLambdaFunction function C.5
C.8 The PrintForm function C.5

chapter 1

Introduction

1.1 Assignment.

The Design Automation Section of the Eindhoven University of Technology recently has developed a design entry program ESCAPE. ESCAPE allows fast prototyping of designs at a high level of abstraction and developing a high level description of a design. To go beyond designing and simulation, an extension of ESCAPE is designed and implemented. This extension provides programming capabilities as well as access to the ESCAPE data structures.

1.2 Outline of the tool.

The tool will be a programming language with an interface to external data structures, written in **C**. The tool will be integrated with another C-program (ESCAPE), shown in figure 1.1.

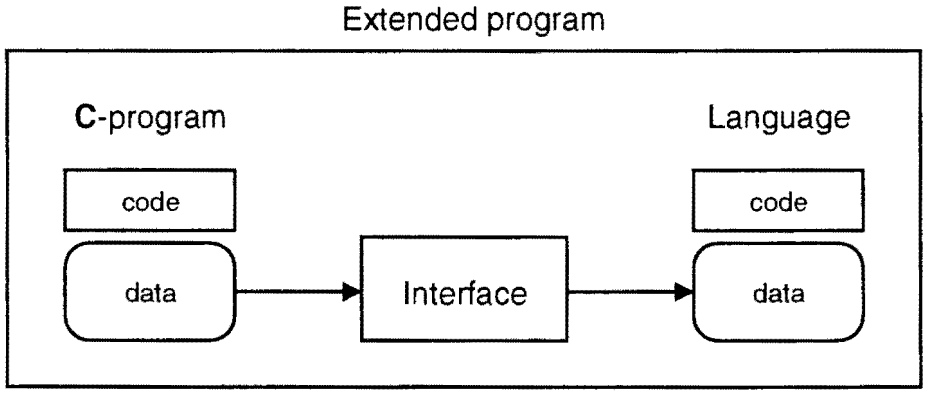


Figure 1.1: Integration of the tool.

An interface is necessary to translate the data structures of the C-program to data types of the tools' language. The complexity of this interface depends therefore on the language chosen for the tool.

We have looked at several programming languages that are eligible to be implemented, each with their own advantages and disadvantages:

C

ESCAPE is written in the programming language **C** which implies that the data structures of ESCAPE are **C** data types. If we would choose **C** then the interface between the tool and ESCAPE would be simple, since no *conversion* of data types would be necessary. Another advantage is that **C** is widely known which would make the tool easy to use.

On the other hand, the implementation of a **C** interpreter is certainly not easy. Besides that, **C** is not really suited for interactive use. Choosing **C** would also imply that the user would have to work with two different languages since ESCAPE uses a LISP-like language to define the behaviours of modules.

LISP

LISP is used by ESCAPE to define behaviours of modules, so the user will be confronted with only one language-syntax when working with ESCAPE.

The implementation of a LISP dialect is not difficult and extra LISP functions can be added dynamically to enhance the functionality of the tool. Furthermore, because LISP is based on functional programming, it is well suited for interactive use.

Because the **C** data types are different from the LISP data types, the interface is not trivial.

Custom-script language

A script language can be customised completely because it does not have to implement some other, already existing, language.

The interface is again not trivial because the data types of **C** are likely to be different from data types of the new language.

As with, **C** two languages need to be learned by the user, the build-in LISP interpreter for the description of the behaviours of the user-defined modules and the script-language. Also, because it is a completely new language, it has the steepest learning curve of all.

1.3 Choice and motivation.

Because of the constant interaction between the user and the tool, we have chosen to design and implement a LISP *interpreter* with an interface between the interpreter and ESCAPE. LISP is well suited for interactive use and is easy to implement and expand.

The interface between ESCAPE and the interpreter will be flexible and not be limited to data structures used by ESCAPE. This way, the tool can also be integrated with other programs that have other data structures. The tool has no functions that operate on the interfaced data structures. It is therefore necessary that special functions that the user has written to operate on its own data structures can be added to the interpreter without much difficulty.

chapter 2

LISP interpreter

2.1 Introduction.

LISP is a computer programming language that was developed by John MacCarthy as a language for symbolic computation. The combination of power and simplicity provided by LISP made it especially popular with computer scientists and it is widely used in the field of artificial intelligence. LISP is based on three fundamental ideas:

- Functional programming.
  In LISP, programmers do not write a program in the conventional way but define functions. Instead of executing a program, they "evaluate an expression", which means applying a defined function to some arguments.

- Simplified syntax.
  LISP has a rather exotic way of writing expressions. It uses a prefix notation for writing expressions. This eliminates problems like operator precedence.

- Recursion is the central control structure.
  LISP code rarely uses iteration or even assignment; almost everything is done with recursion.

There are many implementations of LISP, each with their own peculiarities. This is caused by the lack of a generally accepted standard for LISP in the past. In an effort to establish such a standard, **Common LISP** was introduced, which has become the de facto standard.

2.2 General structure.

2.2.1 The Read-Eval-Print loop.

All LISP interpreters use a *Read-Eval-Print* loop like the one shown in figure 1.1 as their main body. With each iteration of this loop, the interpreter *reads* an expression, *evaluates* it and *prints* the result. This loop continues until the user enters a special command that stops the loop (`quit`).

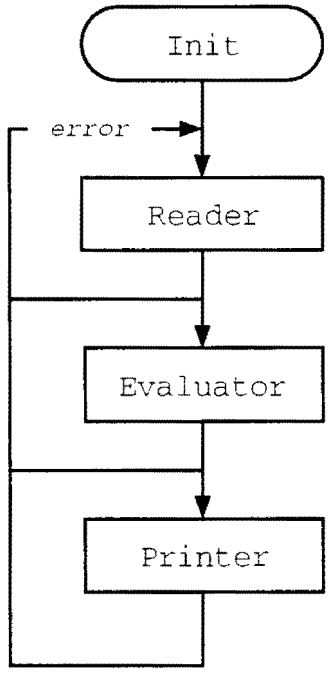


Figure 2.1: The main loop of the interpreter.

As can be seen from figure 2.1, the loop starts again when an object is printed or when some error occurs while reading or evaluating. We will discuss the reader, evaluator and printer in detail in sections 2.2.3, 2.2.4 and 2.2.5.

2.2.2 LISP objects.

During the read-phase, the interpreter translates a *textual* expression into a LISP object, which is the internal representation of the expression. LISP objects are data records that contain a type indicator and a value. All objects in our dialect of LISP have one of the following types:

- Integer
- Symbol
- Cons
- Predefined function
- Real
- Array
- CObject
- Nil
- String
- Bitvector
- Lambda function

2.2.2.1 The integer and real types.

The integer- and real-type form together the numerical types of the interpreter. Unlike **Common LISP**, our dialect of LISP doesn't support infinitely precise integer arithmetic ("bignums") and it also doesn't support complex numbers or complex arithmetic. It does however support floating-point arithmetic.

The integer used by our LISP interpreter is based on the `int` type of **C**, which is the language that the interpreter is written in. This means that this type depends on the compiler used to compile the interpreter source-code. The same applies to the real type, which is based on the `double` type of **C**.

2.2.2.2 The string type.

An object of type string is used to store textual information, like documentation of functions.

The string type is based on the standard **C** string. This means that only a pointer to a sequence of characters (`char *`), ending with a null-character is stored in the LISP object.

2.2.2.3 The symbol type.

The symbol-type is used to create *bindings* between names and objects. When a binding exists between a name and an object, the symbol with the same name is said to be bound to that object. Binding names to objects is described in detail in section 2.3.

A LISP object of the symbol type (sometimes also called just 'symbol') contains a name which is possibly bound to an object. The actual binding is not done by the symbol, but by an entry in a bindingtable. The name stored in the LISP object is a string, based on the standard **C** string, just like LISP objects of type string.

2.2.2.4 The array type.

Arrays are used to store multiple objects in a simple and fast way. Arrays can be one-dimensional, in which case they are also called vectors, but they can also be multi-dimensional.

The information stored in the LISP object is a pointer to a block of memory allocated to accommodate the items of the array, and a pointer to a special structure that contains information about the dimension of the array.

2.2.2.5 The bitvector type.

The bitvector type is included because it reduces the overhead which occurs when a user implements bitvectors with one-dimensional arrays. Bitvectors are indeed much like one-dimensional arrays, but may only contain the integers 0 or 1.

The information stored in the LISP object is the size of the bitvector in bits, and a pointer to a block of memory allocated to store the bits of the bitvector. This block of memory is allocated as a one-dimensional array of integers. The number of bits that each integer contains depends on the compiler used to compile the interpreter.

2.2.2.6 The cons type, cons-cells and lists.

An object of type cons is also called a cons-cell. A cons-cell is used to combine two other objects into one single object. This is primarily used to construct linked lists, which is probably the most used data-type in LISP.

A cons-cell has two fields that are historically called **car** and **cdr**. These fields are stored in the LISP object as pointers to other LISP objects. In figure 2.2, a cons-cell is shown.

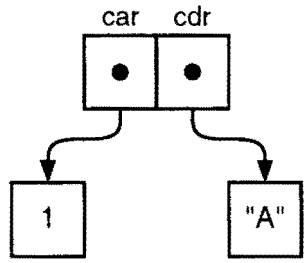


Figure 2.2: A cons-cell.

When creating a linked list, then by convention, the cdr-pointer of the cons-cells in the list is used to point to the next cons-cell (a LISP object of the cons type) in the list and the car-pointer is used to point to the element of the list. Of course, one must be able to determine when a list ends. This is done by pointing the cdr-pointer of the last cons-cell in the list to a special value, `NULL`. To the user of the interpreter it seems as if the cdr-field of the cons-cell is pointing to the special symbol `nil`. This is shown in figure 2.3.

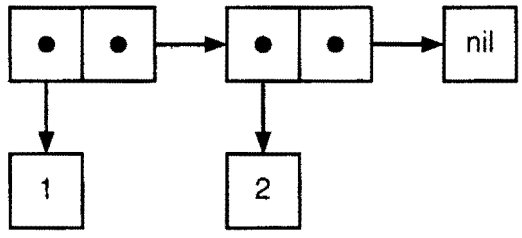


Figure 2.3: An example of a list.

A list can also be ended by having the cdr-field of the last cons-cell in the list pointing to something else then `NULL`. This is sometimes useful (for example, to implement complex numbers or so-called assoc lists).

2.2.2.7 The lambda- and predefined function types.

Objects of type lambda-function (user-defined) or predefined-function contain information that is used by the evaluator to apply a function to a list of arguments.

The information stored in the LISP object is a pointer to a string, containing the documentation of the function, and a pointer to a structure, containing more information about the function. This includes the number of required and optional arguments and in the case of a lambda-function, a list of arguments and a pointer to a list of LISP objects, which represents the body for the lambda-function. In case of a predefined-function, its structure contains besides the number of required and optional arguments a pointer to the **C** function that implements the predefined function.

2.2.2.8 The CObject type.

This type is not a standard LISP-type. CObjects are created by a special part of the interpreter that controls the interface between the LISP-interpreter and an arbitrary **C**-program. This LISP-interpreter is capable to access external data structures that must be made public using functions of the module `publishmanager`.

The information stored in the LISP object is a generic pointer (`void *`) which points to a data-structure that was published, and a ctype indicator, an integer that specifies the type of the embedded generic pointer.

CObjects and the `publishmanager` (the interface between the LISP interpreter and an arbitrary **C**-program) are discussed in detail in chapter 3.

2.2.2.9 The nil type.

This type of LISP object is used to indicate the empty list and as a return value for functions that can't return anything useful. It is also used to indicate the boolean value `FALSE`. There is one object of this type that is bound to the name `nil`. Another object of type symbol is bound to the name `t`. That object is used to indicate `TRUE`.

2.2.3 Reader module.

2.2.3.1 Introduction.

Now that we have introduced the basic types of the interpreter, we can look at the first module involved in the main loop of the interpreter, the reader.

The reader reads text from either a keyboard or a file, and if possible, translates this text into a LISP object. If this translation is not possible, it generates an error.

The reader automatically converts integers to reals if the integers' magnitude is too large. It also converts reals to integers when the fractional part of the real is exactly 0, and the magnitude of the real is not too large.

In figure 2.4, we present the syntax that is accepted by the reader. The text that is given to the reader is pre-processed to eliminate comments. The syntax of integer and real numbers depends on the compiler used to compile the source-code of the interpreter, since the standard **C**-function `strtod` is used to convert the text into a double. The result of reading the number is an integer, if the result is representable by an integer, otherwise the result is a real.

The reader can read only LISP objects of type integer, real, cons, string, symbol, nil and bitvector. Objects of type array, lambda- and predefined function and CObject must be created using special functions.

Start symbol: `form` Terminal symbols are given in **bold** style, non-terminal symbols in `normal` style.	
`form`	→ `space* • (list | quote | number | string | symbol | bitvector)`
`list`	→ `( • form • (space • form)* • [ space+ • . • space • form • space*] • )`
`quote`	→ `' • form`
`number`	→ `integer | real`
`string`	→ `" • ( ¬ " )* • "`
`symbol`	→ `symchar • ( ¬ " )*`
`bitvector`	→ **`#`** `• space* • ( **0** | **1** )* • space* • )`
`space`	→ `' '`
`integer`	→ `[ + | - ] • digit+`
`real`	→ `[ + | - ] • ( digit+ • [ . • digit* ] | digit* • . • digit+ ) • exponent`
`exponent`	→ `[ ( **e** | **E** ) • [ + | - ] • digit+ ]`
`symchar`	→ `¬ ( ( | ' | " | # | ( + | - ) • digit | . )`
`digit`	→ **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Figure 2.4: The syntax accepted by the reader, given in BNF.

2.2.3.2 Preprocessing of the input.

The pre-processing of the reader includes reading the text from a file or string, deleting comments and control characters other than the end-of-line character.

Comments must be preceded by a semi-colon (;). When a semi-colon is read outside a string, the rest of the text on the same line is ignored, including the end-of-line character.

When the pre-processor reads an end-of-line character and the parenthesis (and) match, the pre-processed text is returned to the actual reader which checks the syntax of the text and translates it to a LISP object when possible.

If a tilde (~) is read by the pre-processor, the previously read text is ignored completely. This is useful for cancelling a long command.

2.2.4 Evaluator module.

2.2.4.1 Introduction.

The evaluator is the most important part of the interpreter. It evaluates a LISP object according to the evaluation rules which are quite simple. The rules are given in figure 2.3.

There are two kinds of LISP objects:

- Self-evaluating objects.
- Non-self-evaluating objects.

The self-evaluating objects are simple to evaluate because they evaluate to themselves. The evaluation function just returns the pointer it has received as its argument.

Self-evaluating object types		Non-self-evaluating object types	
Integer	- return itself	Symbol	- return bound object
Real	- return itself	Cons	- evaluate as function call
String	- return itself	CObject	- try to translate to LISP object
Array	- return itself		
BitVector	- return itself		
Lambda function	- return itself		
Predefined function	- return itself		
Nil	- return itself		

Figure 2.5: Evaluation rules for LISP objects.

Non-self-evaluating objects are passed to a for each type special evaluation function that can evaluate the object. These special functions that evaluate symbol, cons and CObject type objects will be described in the next sections.

2.2.4.2 Evaluating symbol type objects.

LISP objects of type symbol, also called symbols, are explained in section 2.2.2.3. In that section it is explained that symbols contain a name that can be bound to another LISP object. Although not correct it is common practice to call the symbol a variable and the object it is bound to, its value.

When a symbol object is evaluated, the name that it contains (see 2.2.2.3) is used to search in the bindingtable to find a binding with the same name. It then returns the object that was bound to that name.

The value associated to the name is found by calling the function `FindBinding` with that name as argument. It returns a pointer to the bound object or it returns the `NULL` pointer when no object was bound to that name.

Further details about storing and management of bindings in the bindingtable can be found in section 2.3, which also describes the way the interpreter handles the symbol management when a user-function is called.

2.2.4.3 Evaluating cons type objects.

Objects of type cons are assumed to be function calls. The cons-cell passed to the evaluator is considered to be the first cell of a list. The first member of that list must specify the function being called. Consequently, this object must evaluate to either a lambda-function or a predefined-function. The rest, or tail of the list is considered to be the list of arguments of that function.

```
(function argument1 argument2 ... argumentn)
```

Before the evaluation is continued, the number of arguments is checked. The minimal and maximal number of arguments are specified by the function that is being called. If too many or too few arguments are specified, an error is generated and the evaluation is stopped. If the number of arguments is valid for the called function, the evaluation continues according to the type of the function.

Lambda (user-defined) function

If the function is a lambda-function then a special function `EvalLambdaFunction` is called to evaluate this function. This function first evaluates all the arguments (in an order that corresponds to the order in which they appear in the function call, from left to right) and binds the parameters to the result of these evaluations.

Three types of parameters can be defined for a user-function:

- Required parameters.
  Each required parameter must be specified by an argument. This implies that the number of required parameters defines the minimal number of arguments that must be specified.

- Optional parameters.
  Optional parameters need not be specified by an argument. Optional parameters that are not specified by an argument are initialised to `nil` or another value that can be specified by the user.

- Rest parameter.
  To be able to define user-functions that accept an unlimited number of arguments, a rest-parameter may be specified. If such a rest-parameter is indeed specified, then all arguments that are not used to initialise required or optional parameters are evaluated and the results are placed on a list. This list is then bound to the rest-parameter.

We see that the lower bound for the number of arguments is given by the number of required parameters. If no rest-parameter is specified, the upper bound is given by the sum of the number of required parameters and the number of optional parameters. If however the rest-parameter is specified, then there is no upper bound for the number of arguments.

When the parameters are bound, the body of the function is evaluated. This body may consist of several LISP objects. The result of the evaluation of the last object of the body is stored as the result of the evaluation of the lambda-function. This value will be returned after the evaluation is completed.

Before the result can be returned, the bindings that were created during the evaluation of the user-function must be removed from the bindingtables. This includes the bindings that are created for the parameters of the function and the bindings that were created by the evaluation of the body of the function. When the bindings are removed, the result is returned and the evaluation is completed.

The implementation of how the bindings are created, stored and removed is explained in detail in section 2.3.

Predefined function

This type of function is implemented as a **C**-function. The evaluator calls this **C**-function passing the list of arguments. The **C**-function must return a LISP object which is then returned as the result of evaluating the object of type cons.

The way the arguments are evaluated (if they are evaluated) is specified only by the function being called.

2.2.4.4 Evaluation of CObject type objects.

CObjects usually embed pointers to structures or array's. These CObjects evaluate to themselves because there is nothing else they can evaluate to meaningfully.

In some cases, a CObject contains a pointer to either an integer, double or string, instead of a pointer to a structure or an array. In that case the evaluator returns an object of type integer, real or string respectively. The *value* of that LISP object is equal to the value that the pointer embedded in the CObject is pointing to.

When the CObject embeds a pointer to a string (that is, a pointer of type `char **`), the string is copied and the pointer to the first character of the string is changed to point to the first character of the new string. The old string is left untouched.

For more information on CObjects see chapter 3 which deals with the `PublishManager`.

2.2.5 Printer module.

The printer module is used to produce a textual representation of LISP objects, and print this to the screen or file. All objects are printed using the standard **C** printing function `fprintf`, to print to an output stream (which is either the standard output stream `stdout` or a file).

We have now discussed the three basic modules reader, evaluator and printer. Appendix C presents the interpreter in pseudo-code that describes these modules more formally.

2.3 Scoping: symbol management.

2.3.1 What is scope ?

In most programming languages it is possible to use an identical name for global variables and parameters of functions, with the last taking precedence. The problem of determining the *declaration* of a variable that is associated with the *use* of the variable is called scoping. Figure 2.6 shows a quick view of scoping in **Pascal**.

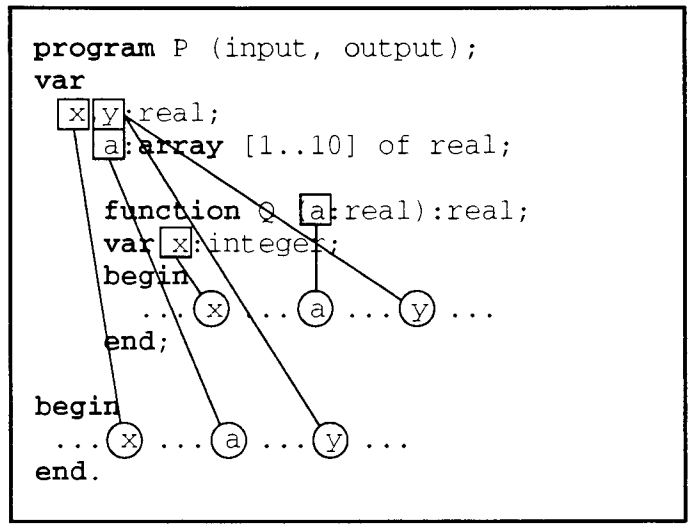


Figure 2.6: Scope in Pascal.

The example shows the use of global variables and local variables with conflicting names. In the figure, each use of a variable (circled) is connected with the associated declaration (boxed).

2.3.2 Dynamic scoping versus static scoping.

In **Pascal, Common LISP** and other languages scope is lexical or static. This means that the connections between uses and declarations of variable names can be made statically, based on the text of the program.

In most languages, declarations of variables can only be made in a *block*. We consider the declarations for the global variables to be made in a block that surrounds all other blocks. If blocks are defined inside other blocks, then static scope states that the declaration associated with a variable is the declaration made in the first block that declares the variable and surrounds the use of the variable.

In figure 2.6 we see two blocks (the procedure P and the function Q). Each block surrounds the use of x inside the function Q. Although x is declared within procedure P, this is not the declaration that is associated with the use of x within function Q because x is declared within Q and this block is 'closer' to the use of x than block P.

In most LISP dialects, a different kind of scoping is used. This scoping is based on the time when a name is bound to an object. This type of scoping is called dynamic scoping. The last binding in time is considered to be the effective binding of a name. This does not mean that the earlier bindings are completely lost when a new binding of a value with the same name is made. The old bindings are just ignored until the new binding is removed. Dynamic scope in LISP is shown in figure 2.5.

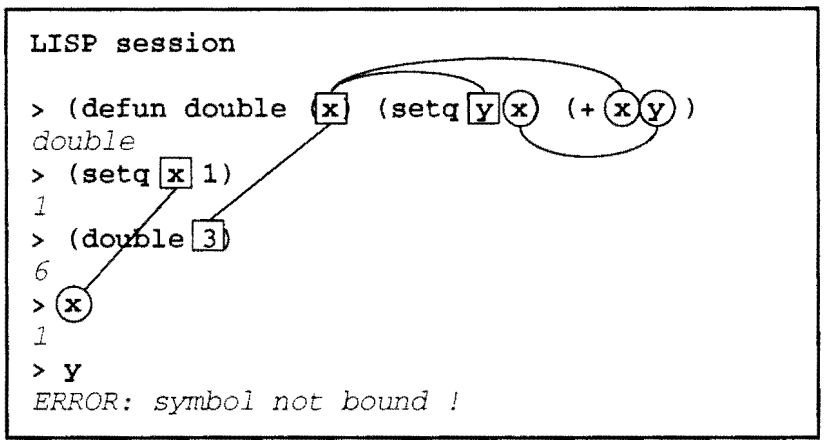


Figure 2.7: Scope in dynamic scoped LISP.

Static scope is superior to dynamic scope since static scope abides to the so-called *α-conversion*. This means that the result of an evaluation does not change when the names of local variables (the parameters of a user-function) are changed. Static scope also allows functions to be created that have truly local variables which value is stored even when the function is not called (like the static variables of C functions).

Static scope uses closures, which are lists of bindings between names and values and which are associated with function calls. Whenever a function is called, a closure is used to establish the valid bindings of so-called free variables. Free variables of a function are variables that are used in the body of the function and that were not bound by that function or globally when the function was defined. Using closures costs memory, since every function called must have its own closure and time, because searching for bindings in closures is slower than searching in a global bindingtable. Also, the implementation of static scope is more difficult than dynamic scope.

Although static scope is nowadays more preferred (see section 4.7 of [Kamin, 90]) and is used in **Common LISP**, our interpreter uses dynamical scoping because it is efficient, easy to implement and there is really no need for the extra possibilities static scope offers (real λ-calculus for example). This eliminates the use of closures and simplifies the way functions are handled. It does however place an amount of responsibility to the user because dynamic scoping can create name-conflicts when global variables are used inside functions that are called from other functions in which a parameter has the same name as a used global variable. The problem illustrated here can be avoided by using unique names for local variables.

2.3.3 Implementation of dynamical scoping.

The rule for dynamic scoping is that the value associated with a binding of a name is the value of the most recent, still valid binding. Normally, when a binding is created using the LISP functions `setq` or `defun`, it overrides the earlier binding, which is replaced by the new binding.

Multiple bindings of a name can only be defined by evaluating a user-function. It is clear however, that only one of these bindings can be considered to be the effective one. Dynamic scope demands that the effective binding is the binding that was made most recently and that is still valid.

As in other languages, calling functions is a LIFO-process, which means that if a function calls another function, the calling function can only continue when the called function returns. The creation and removing of multiple bindings of a name is therefore also a LIFO-process. This implies that bindings of a certain name can be stored on a stack, with the most recent binding on top of the stack. This is also the effective binding of that particular name.

These stacks of bindings can be joined to form a linked list of stacks. This list is called a bindingtable. Each stack contains all the bindings for a certain name and the effective binding is placed on the top of the stack. Finding the effective binding of a name is then reduced to finding the stack that contains the bindings for that name.

Because it is likely that many names are going to be bound, we distribute names over a number of bindingtables. As a result of this, searching for the effective binding of a name is faster. To determine the bindingtable for a binding we use a simple hash-function that is based on the name that is bound.

The bindingtables are implemented as double-linked lists of structures called `BINDINGSTACK` that is shown in figure 2.8. Each `BINDINGSTACK` structure contains the name that is bound by the bindings on that stack, two `BINDINGSTACK` pointers `prev` and `next` to maintain the double-linked list of stacks and a `BINDING` pointer `top` to the top of the stack of bindings.

```
typedef struct bindingstack {
  STRING       name;
  BINDINGSTACK *next;
  BINDINGSTACK *prev;
  BINDING      *top;
};
```

Figure 2.8: Definition of the BINDINGSTACK structure.

As mentioned before, the bindings for a certain name are placed on a stack. This stack is a single-linked list of BINDING structures shown in figure 2.9. Each BINDING structure contains a OBJECT pointer binding, which points to the object that is bound to the name. It also contains a field flags, which can indicate if the binding can be overwritten. For instance, all predefined functions can not be overwritten by default. The field next is used to maintain the single-linked list. Finally, the stackptr-field is a pointer to a stack that is created for each evaluation of a user-function. It is used to determine at which 'level' the binding was created.

```
typedef struct binding {
  int         *flags;
  OBJECT      *binding;
  BINDING     *next;
  LOCALSTACK *stackptr;
};
```

Figure 2.9: Definition of the BINDING structure.

When the interpreter evaluates a user-function it must remove all bindings that were created during that evaluation before the result is returned. Due to the LIFO-nature of the evaluation process, those bindings are always the most recently made still valid bindings of a name. This implies that they appear at the top of their stack. The bindings can be removed quickly when pointers to the stacks involved are known. For that purpose, pointers to these stacks are placed on a stack that is local to the evaluation of the user-function. This localstack is implemented as a single-linked list of LOCALSTACK structures.

The LOCALSTACK structure is shown in figure 2.10. The stack-field points to a BINDINGSTACK that has a binding on top of the stack that must be removed when the function is completely evaluated. The next-field points to the next entry on the localstack.

```
typedef struct localstack {
  BINDINGSTACK *stack;
  LOCALSTACK   *next;
};
```

Figure 2.10: Definition of the LOCALSTACK structure.

The localstack is emptied when a user-function has been evaluated completely. The pointers that pop of this stack are used to remove the bindings that were created during that evaluation.

To simplify the bindingtable management functions a *sentinel* is used. This is an empty `BINDINGSTACK` structure that is located at the front of the bindingtable that makes the double-linked list a circular list and eliminates the need for boundary checking. The `BINDINGSTACK` structures in a bindingtable are sorted alphabetically to make the search for the stack that contains the bindings for a particular name faster.

Of course other types of bindingtables could be implemented. Especially a 2-3 tree might speed up the searching process when many bindings are created. Using a linear search in a bindingtable takes a linear amount of time O(n), but searching in a 2-3 tree takes O(log n) time. However, insertion and deletion in a 2-3 tree are not as simple as in a double-linked list. As explained before, when a binding has to be removed, it must be at the top of its stack. Since a pointer to this stack is known, the binding can be deleted in constant time, in contrast to the deletion of entries in 2-3 trees which requires O(logn) time in the worst-case, even when a pointer to the node is known (see section 5.4 of [Aho,82]).

Appendix A shows an example of how the bindingtables change during the evaluation of a user-function.

2.4 Memory management.

2.4.1 Introduction.

The LISP interpreter often allocates (and deallocates) LISP objects from the memory-heap. These objects are C-structures of type OBJECT shown in figure 2.11.

```
typedef struct object {
  int        type;
  int        refcount;
  PROPERTY *properties;
  union {
    int       a_integer;
    double    a_real;
    STRING    a_string;
    STRING    a_symbol;
    struct {
      int vectorsize;
      int *vector;
    } a_bitvector;
    struct {
      OBJECT    **items;
      DIMENSION *dim;
    } a_array;
    struct {
      STRING documentation;
      union {
        LAMBDA    *lambda;
        PRED      *pred;
      } function;
    } a_function;
    struct {
      OBJECT *car;
      OBJECT *cdr;
    } a_cons
    struct {
      int        ctype;
      void      *cpointer;
    } a_cobject;
  } value;
};
```

Figure 2.11: Definition of the OBJECT structure.

This OBJECT structure is the main data-object that the interpreter uses to store values. For example: the text that the user enters is translated by the reader to an object and the values that are returned by the predefined functions are objects.

There are always references to an object. The object may be part of a list, in which case the previous cons-cell in the list points to the object through its cdr-field. It may be bound to a name, in which case the object is pointed to by the binding-field of the associated `SYMBOL` structure or an internal variable of type `OBJECT *` of the interpreter may point to it.

When the last reference to an object is removed, this object becomes inaccessible. This can happen when some special LISP function is used (like `rplaca` and `rplacd`), the binding of a name is changed or when some internal pointer is changed. This object should be returned to the memory-heap so that it can be used again. An example of an object that becomes inaccessible is shown in figure 2.12a and 2.12b. Figure 2.12a shows a list, (1 2 3) which is bound to the symbol x.

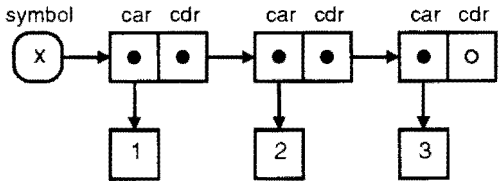


Figure 2.12a: Example list (1 2 3) bound to x.

Now, the car-field of the list is changed using the replaca function:

```
(replaca x (car (cdr x)))
```

This leaves the integer 1 with no references, which is shown in figure 2.12b. This integer-object can now be deallocated or reused to store other values.

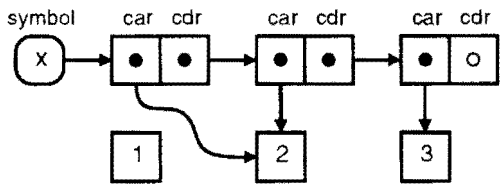


Figure 2.12b: The list after it is changed.

2.4.2 Methods of memory-management.

Basically, there are two different techniques for reclaiming objects that are free. It is possible to do it indirect, like the stop-and-copy garbage collection or direct, as in reference-counting. We will look at both techniques in the next two sections.

2.4.2.1 Indirect management: stop-and-copy garbage collection.

Although there are more indirect memory-managers, we look at the stop-and-copy garbage collection method as an example of indirect management. This method uses two heaps of memory. One heap is the heap that is used to allocate objects and is called the *allocation* heap. The other heap, called the *free* heap, is equal in size to the allocation heap but is not used until all the objects of the allocation heap have been allocated.

The allocation heap is divided into two parts. One part containing allocated (and possibly already freed) objects, and one part containing objects that are available for allocation. Thus, new objects are allocated from the second part of the allocation heap until it runs out. When the available memory in the allocation heap is used up, we switch to the other heap, copying only the accessible objects from the allocation heap to the free heap. Now, the roles of the heaps switch, the allocation heap becomes the free heap and the free heap becomes the allocation heap.

Finding accessible objects is done using a depth-first search on all the objects that were bound to symbols (note that the garbage collection can therefore be done only at top-level). Moving accessible objects to another place in memory implies that the pointers that link objects have to be changed to the new addresses of the objects (it is possible to store the new address of an object in a special field inside the object at the old address. When an object points to the old object, that pointer can then be updated using the value stored in that field).

The moved objects are placed in the first part of the free heap, so that the second part of the free heap can be used to allocate objects.

Although easy to use and to implement, it is clear that the stop-and-copy garbage collection causes the interpreter to be halted for the time needed to perform the garbage collection. For a detailed implementation of stop-and-copy garbage collection see section 10.4 of [Kamin,90].

2.4.2.2 Direct management: reference counting.

There is another technique for memory-management that operates run-time. This technique uses a reference count field to count the number of references made to a certain object.

The reference counter of an object is increased each time a new reference to that object is made, and it is decreased when a reference is removed. When the counter reaches 0, there are no more references to the object and it may be deallocated.

It is obvious that managing the reference counter of LISP objects must be done while the interpreter is evaluating expressions. This spreads the time needed for the garbage collection through the computation and there is no need to interrupt the interpreter.

Reference counting garbage collection is usually rejected by textbook authors as "probably not worth doing so" (section 12.2 of [Aho,82]). In section 10.5 of [Kamin, 90], the author lists three arguments against reference-counting:

- Efficiency

  "Maintaining reference counts is an on-going cost of computations, which can be a considerable percentage of total cost when the code is compiled into an otherwise efficient form. On the other hand, garbage collection methods have their own efficiency problems, so this disadvantage is not easily assessed."

- Generality

  "The major problem with reference counting is that it cannot deal with circular data structures, such as those that can be created by rplaca and rplacd. By contrast, the garbage collection methods we have studied have no trouble with circular data structures."

- Simplicity

  "Though simple in concept, reference counting can be very tricky to implement. It is dangerous to draw any general conclusions from the complexity of our implementation, but what can be said about reference-counting in general is that it requires pervasive changes in the evaluation process. Garbage collection, as we have seen, can be very cleanly separated from the evaluation *per se*."

Kamin shows two important points about reference counting. One argument states that reference counting is not able to cope with circular lists. This is not quite true, although the performance of the reference count technique is undoubtedly inferior to the stop-and-copy garbage collection, because it needs to check for circularity of lists in some situations. That is probably why no LISP implementation has used reference counting up to now (chapter 10, [Kamin,90]). On the other hand, we have no need for circular lists, and we have decided not to use them. Kamins' second point is that reference counting is difficult to implement and that it is not possible to implement reference counting without altering the evaluation process drastically. As we will see, this is not the case in out implementation.

Although reference counting is not recommended by textbook authors like Kamin and Aho for implementing a full LISP version, we have decided to implement the interpreter based on reference counting garbage collection. This limits the interpreter because of the inability to handle circular lists, but makes the interpreter more *friendly* to the user. The interpreter is not stopped to perform garbage collection.

Steele claims in [Steele,1975] that the time spend for garbage collection is typically 10 to 40 per cent of the total computation time. There no reason to believe that reference counting is more time-efficient then stop-and-copy or mark-and-sweep garbage collection. The main advantage of reference count garbage collection is that the time needed for garbage collection is spread evenly through the computation time. This prevents the interpreter to be interrupted to collect garbage.

2.4.3 Implementation of reference counting.

2.4.3.1 Basic technique.

To implement the reference counting garbage collection, each LISP object has a reference counter. This is the `refcount`-field of the `OBJECT` structure. The `refcount` of an object must always be equal to the number of references to the object. This way, if the reference counter of an object is decremented from 1 to 0, the last reference to that object is removed and the object may be deallocated or reused.

The implementation of reference counting is based on the following rules:

- References can only be made by a **C** pointer of type `OBJECT *`.
- The `refcount` of an object must always be equal to the number of references to that object. The initial value for `refcount` is therefore 0.
- If the `refcount` of an object changes from 1 to 0, the object can be deallocated or reused, since the last reference to it is removed, making it inaccessible. This rule is not valid for objects that are to be returned by functions. It must be possible to return objects with a reference count of 0, without deallocating them.

Since the reference counter of an object can only be changed when a reference is added or removed, we need to know when such a change is made. There are basically only two such situations:

- Assignment of an object-pointer.
- An object-pointer becomes inoperative.

The latter occurs when local variables of type `OBJECT *` of a function were assigned to some value and the function terminates, thus eliminating the local variables, or when another LISP object which contains pointers to LISP objects is deallocated (cons-cells and arrays).

2.4.3.2 Managing reference counters.

The previous rules are implemented using one function and six macros. The user must use only the function and two macros.

- `FreeObject(objectptr)`

  The function `FreeObject` accepts one argument that must be an object-pointer. The `refcount` for this object is decreased by one and when it reaches 0, the object is deallocated. The function is used when a pointer is going to be inoperative, either because the pointer is a local variable of a function that is terminating or because an object containing the pointer is deallocated.

- `ASSIGNOBJ(objectptr1,objectptr2)` ≡

```
(assignmentaux = (objectptr2), INCREFCOUNT(assignmentaux),
  FreeObject(objectptr1), (objectptr1) = assignmentaux )
```

  `ASSIGNOBJ` uses two arguments which both must be an object-pointer. It assigns the value of `objectptr2` to `objectptr1`, after it has adjusted the `refcounts` for the objects pointed to by `objectptr1` and `objectptr2`. The macro `INCREFCOUNT` is used to increase the refcount of the object pointed to by `objectptr2` by one, since a new reference is made to this object. The function `FreeObject` is called with `objectptr1` as argument, since a reference to the object it is pointing to is going to be removed.

- `RETURNOBJ(objectptr)` ≡

```
return(((objectptr) != NULL)?(--((objectptr)->refcount), objectptr)
:(NULL))
```

  `RETURNOBJ` uses one argument that must be an object-pointer. The refcount of the object is decreased, without checking if it reaches 0. It then returns from a function with the value `objectptr`.

The following macros are not to be used by the user, but are used by the interpreter itself.

- `SETREFCOUNT(objectptr,value) ≡`

```
(((objectptr) != NULL)?((objectptr)->refcount = (value)):(0L))
```

`SETREFCOUNT` uses two arguments. The first argument is an object-pointer that specifies the object to use. The second argument is an integer that specifies the new value for the `refcount`. The `refcount` of the indicated object is set to the new value. This macro is mainly used to set the initial value of the `refcount` of a newly allocated object to 0.

- `REFCOUNT(objectptr) ≡`

```
(((objectptr) != NULL)?((objectptr)->refcount):(0L))
```

`REFCOUNT` uses one argument that must be an object-pointer. It returns the value of the `refcount` of this object as an integer.

- `INCREFCOUNT(objectptr) ≡`

```
(((objectptr) != NULL)?(++(objectptr)->refcount):(0L))
```

`INCREFCOUNT` uses one argument that must be an object-pointer. It increases the value of the `refcount` of this object by one.

- `DECREFCOUNT(objectptr) ≡`

```
(((objectptr) != NULL)?(--(objectptr)->refcount):(0L))
```

`DECREFCOUNT` uses one argument that must be an object-pointer. It decreases the value of the `refcount` of this object by one. It does not check if the object can be deallocated or reused.

2.4.3.3 Using local object-pointers.

If a function uses local variables of type `OBJECT *`, the references that are created by these pointers are removed when the function terminates, since the variables themselves are no longer accessible. It is therefore necessary that before the function is terminated, all the references by local object-pointers are removed. This is done by passing all the local object-pointers to `FreeObject` before using `return` or `RETURNOBJ` to return from the function.

If however one of these local object-pointers contains the value that is to be returned by the function, this local variable should not be passed to `FreeObject`, because that could mean that the object it is pointing to is deallocated. Instead, this value must be returned using `RETURNOBJ`, which decreases the `refcount` of the object so that even objects with a reference count of 0 can be returned.

It is extremely important that all the pointers to `OBJECT`s are initialised to `NULL`, because the macros and functions involved in the reference counting cannot distinguish a non-initialised pointer from a valid pointer, and therefore consider every pointer that is not `NULL` to be a valid pointer.

2.4.3.4 Allocating and storing of deallocated objects.

Although allocation and deallocation of objects can easily be implemented using the standard **C**-functions `calloc` and `free`, it is usually profitable to allocate more objects together and not freeing them at all, but placing them on a freelist, at least on systems that don't return deallocated memory to the system memory until the process that allocated it is terminated, such as UNIX.

HeapList

Objects are allocated as an array of `OBJECT`s. This enables the allocation of a large number of objects at the same time, but eliminates the possibility of deallocating them individually.

The arrays of OBJECTs are linked to form a list of all the allocated memory for objects. Each entry is a structure HEAPLIST, shown in figure 2.13. This structure has a pointer to the next entry and a pointer to an array of objects. The first entry of the list is pointed to by a global variable, memorylist.

```
typedef struct heaplist {
  OBJECT    *heapptr;
  HEAPLIST  *next;
};
```

Figure 2.13: Definition of the HEAPLIST structure.

The linked list of heaps is used to return the memory that was allocated for the heaps. The list of free objects (deallocated or just allocated and not yet used) is called the freelist.

FreeList

The freelist is a linked-list of LISP objects that are not used. A global variable freelist points to the first LISP object that is free. To link the objects, the car-field of the objects is used to form a single-linked list of free objects.

If the last free LISP object is allocated from the freelist, a new heap is allocated. The function that allocates the heap already creates a linked list for all the objects in the heap. The freelist is then updated by assigning the pointer freelist to the address of the first object in the heap.

If an object is deallocated, it is placed at the head of the freelist. This is done by pointing the car-field of the object to the object that freelist is pointing to. Then freelist itself is changed to point to the deallocated object.

Figures 2.14a and 2.14b show how the freelist and heaplist interact. Figure 2.14a shows the situation just after a new heap of objects is allocated (in this example, a heap contains 4 objects). The other heap that is shown is completely used.

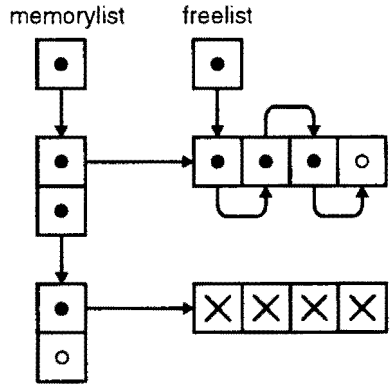


Figure 2.14a: The freelist and heaplist just after allocation of a new heap.

Figure 2.14b shows the situation after a used object was freed. The deallocated object is placed at the head of the freelist.

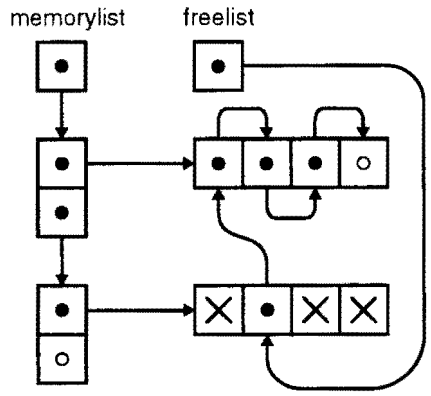


Figure 2.14b: The freelist and heaplist just after deallocation of a used object.

The same technique can be used with other frequently allocated and deallocated structures like `BINDING`, `BINDINGSTACK` and `LOCALSTACK`.

2.5 Further reading.

The first publications about LISP were by John MacCarthy [MacCarthy,60] and [MacCarthy,62]. More modern treatments of LISP and LISP-like langauges can be found in [Friedman,74], [Touretzky,84], [Wilensky,86] and [Winston,84]. For more information about Common LISP, see the Common LISP manual by Steele [Steele,84].

The book of Abelson and Sussman [Abelson,85] is a good introductory book on Scheme, a LISP-like language using lexical scope. It also provides a detailed discussion about scope and many examples and problems. More about scoping can be found in [MacLennan,87], [Marcotty,86] and [Sethi,88].

Methods for garbage collection are descibed in [Knuth,73], [Standish,80] and [Cohen,81]. The book by Kamin [Kamin,1990] descibes many techniques for functional programming languages, as well as topics as scope and memory management.

chapter 3

Interface

3.1 Introduction.

The interpreter is designed to be integrated with another **C**-program. Although the interpreter and the **C**-program share the same *core* (on Unix systems) this does not imply that the interpreter can access the data-structures of the **C**-program directly, since these data structures are not LISP types and therefore not usable by the interpreter.

Because the interpreter itself can't access the data structures of the **C**-program, an interface must provide a way to do this. This interface must translate the external data structures to a LISP object.

The interface must also be capable of expanding the functionality of the LISP interpreter by dynamically adding new functions to it.

3.2 Outline of the interface.

To translate the external data structures into LISP data, we have created the special CObject LISP object type. LISP objects of that type are also called CObjects.

CObjects contain a generic **C**-pointer (`void *`) and a type indicator. The **C**-pointer is used to *refer* to the external data-structure and the type indicator specifies the type of the data-structure that the pointer is pointing to.

Referring to the external data structures has the advantage of not having to copy the complete data. It also enables the interpreter to change the data in place.

The interface must be able to translate the basic data structures of **C**, which are:

- structures
- pointers
- doubles
- arrays
- integers
- strings `(char *)`

To perform the translation, the data structures must be made known to the interface. This is what we call *publishing* a data-structure. The interface manages these publications and is therefore also called the *publishmanager*.

In figure 3.1 the place of the interface between the **C**-program and the interpreter is shown.

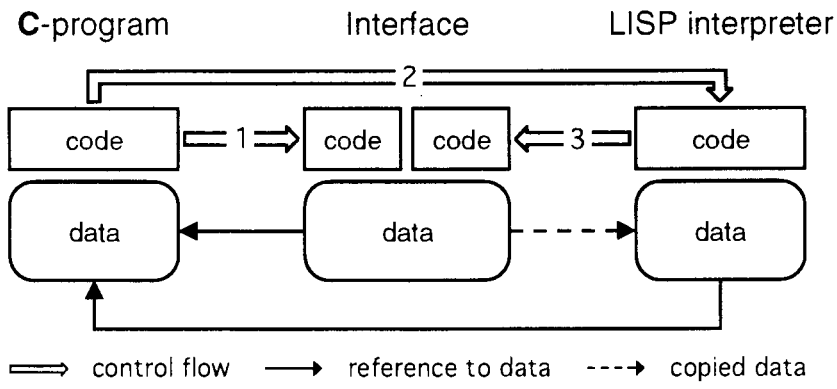


Figure 3.1: Place of the interface.

The LISP interpreter is invoked by the **C**-program in three steps shown in figure 3.1:

1) The C-program publishes the type-definitions, functions and values.

2) The C-program calls the `LispInterpreter` function, which initialises the interpreter.

3) After initialisation of the LISP interpreter, the CObjects containing references to the published data, are created and bound to user-defined names. During this process, the published functions are also added to the interpreter.

The publication of the data structures that must be interfaced is done in two steps:

- Publication of types and functions:
  The **C**-program publishes functions and definitions of the data structures that must be made known to the interpreter by calling appropriate interface functions. This has to be done only once, because these definitions remain valid until the **C**-program is terminated.

- Publication of values:
  After the publication of the types, values may be published. These values must be pointers to data structures which type is published. In addition to that, constants may be published. These published constants are not translated to CObjects, as are the published pointers, but to LISP objects of the corresponding type. The interface allows integers, doubles and strings to be publihsed as constants. The publicated values remain valid until the C-program terminates.

Publication of type-definitions, functions and values results in the creation of an intermediate data base with *references* to the data of the **C**-program. This data base, shown in figure 3.2, is used to construct the CObjects when the LISP interpreter is called from the **C**-program.

Appendix D shows a brief summary of the functions of the publishmanager that are to be used to publish data structures and to use the translated data structures from LISP.

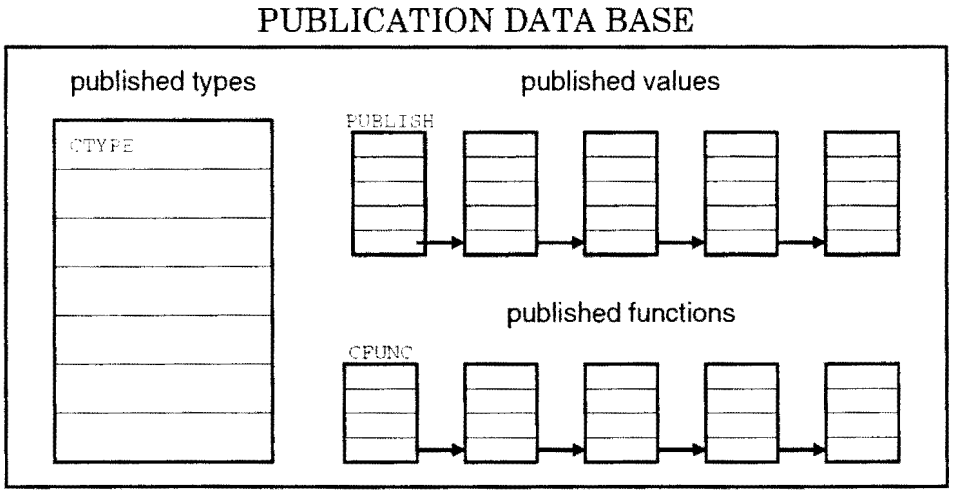


Figure 3.2: The publication data base.

3.3 Publishing types.

3.3.1 Storing definitions of published types.

The publication data base contains an array of CTYPE structures which are used to store the definitions of the publish types. Each CTYPE structure, shown in figure 3.3 contains the definition of one published type. Internally, the interface uses the *index* of a CTYPE structure that defines a certain published type to refer to that type. The type indicator of the CObjects of a certain published type uses this index to specify the type of data structure that the generic pointer is pointing to. The index of a CTYPE structure that defines a certain publish type is also called the **C**-type number of that type.

User-defined publish types are either definitions of a **C**-structure or a **C**-array. The CTYPE structure is used to store both kinds of type-definitions, although they differ slightly. The part of the definitions that are specific to each kind are stored in two structures that are joined in by the union value. The interpretation of this union depends on the kind of the definition, which is specified by the type field.

```
typedef struct ctype {
  int    type;
  int    size;
  int    flags;
  STRING name;
  union {
    struct {
      CFIELD *fields;
      int     propoffset;
    } a_cstruct;
    struct {
      int arraytype;
      int arraylimit;
      int ptrcnt;
    } a_carray;
  } a_cthing;
};
```

Figure 3.3: The definition of the CTYPE structure.

3.3.2 Predefined publish types.

The interface has three publish types that are predefined:

- Integer
- Double
- String

These types use the first three `CTYPE` structures of the **C**-type array. Neither of them actually need a definition. The `CTYPE` structure is merely used to store their names and to give them a **C**-type number so that they can be referrred to internally. Because these **C**-type numbers are based on the index of the `CTYPE`, the **C**-type numbers of the integer, double and string type are 0, 1 and 2 respectively.

3.3.3 User-defined publish types.

Besides using the predefined publish types, the user can define new publish types. These types must be based on a **C**-structure or a **C**-array.

3.3.3.1 Publishing array-type definitions.

The specific information for an array is stored in the structure `a_carray` of the union `a_cthing`, shown in figure 3.3. This information includes:

- the type of the items, which must be a published or predefined type.

- the size of the array

In **C** it is possible to define pointers to pointers. A variable of type double ** is a pointer to a pointer to a double. To get the value of the double it is pointing to, this variable must be dereferenced twice. The type of that variable could be seen as a combination of the basic type double and the number of dereferences needed to get the value of that basic type, 2.

Since the items of an array may also be pointers, we specify the type of the items as a combination of a **C**-type number (specifying the basic publish type) and a number indicating how many dereferences are needed to get the value of the basic publish type.

The `arraytype` field of the structure contains the **C**-type number and the `ptrcnt` field is used to store the number of dereferences needed to get the value of the basic publish type. The `arraylimit` field contains the size of the array. This is used to check for the array-boundaries at run-time.

For example, an array that has 100 items that are pointers to pointers to a double has 1 stored in the `arraytype` field (1 is the **C**-type number for the predefined publish type double), 2 stored in the `ptrcnt` field and 100 stored in the `arraylimit` field.

3.3.3.2 Publishing structure-type definitions.

The specific information for a structure is stored in the structure `a_cstruct` of the union `a_cthing`, shown in figure 3.3. This information includes:

- the fields of the structure.

- the offset of a CPROPERTY pointer field, if it is specified for the structure.

The structure `a_cstruct` contains two fields. The first field is a pointer to the first member of a linked-list of `CFIELD` structures. A `CFIELD` structure contains the information about a published fields for the structure.

```
typedef struct cfield {
  STRING name;
  int    offset;
  int    type;
  int    flags;
  int    ptrcnt;
  CFIELD *next;
};
```

Figure 3.4: The definition of the CFIELD structure.

A field definition includes a name for the field, the offset in bytes from the base-address of the structure for which the field is defined and the type of the field. This type is specified in the same way as the type of the items of a published array-type.

The last field is the pointer to the next field that is defined for this structure.

The second field is the offset in bytes of the **C**-property field, if it is defined in this structure. The **C**-property field is a pointer to the first member of a linked-list of `CPROP` structures. **C**-properties are attached to the structure using special LISP functions. The list of **C**-properties is **not** removed when the interpreter returns control to the interfaced **C**-program. This enables adding information to the interfaced data structures that can be examined from the **C**-program.

```
typedef struct cproperty {
  STRING     name;
  int        type;
  union {
    int      cinteger;
    double   cdouble;
    STRING   cstring;
    void    *cptr;
    } value;
  CPROPERTY *next;
};
```

Figure 3.5: The definition of the CPROP structure.

C-properties are identified by their name, stored in the `name` field. The **C**-type of the property is stored in the `type` field.

The union `value` contains room for storing an integer, a double, a string or a pointer. The value is to be accessed according to the type-field.

The last field of the `CPROP` structure is the link to the next **C**-property.

3.4 Publishing data.

3.4.1 Storing published values.

The published values are stored in a linked-list of PUBLISH structures. Each PUBLISH structure, shown in figure 3.6, contains one published value.

```
typedef struct publish {
  STRING     name;
  int        type;
  int        flags;
  union {
    int      cinteger;
    double   cdouble;
    STRING   cstring;
    void    *cptr;
    } value;
  PUBLISH *next;
};
```

Figure 3.6: Definition of the PUBLISH structure.

The cptr field in the value union contains the reference to the published data structure. This pointer will be embedded in a CObject that will be bound to the name that is specified in the name-field.

The flags field can be used to indicate that this published data structure is to be published as a constant. Publishing an integer, a double or a string as a constant means that instead of a CObject, the appropriate LISP object is created and bound to the name when the LISP interpreter is called.

3.4.2 Publishing values as constants.

Integers, doubles and strings can be published as constants. This means that they are not translated into a LISP object of type CObject when the interface translates its data base to LISP objects, but to a LISP object of the appropriate type.

The published value is stored in the appropriate field of the union `value`. For this sort of publication, the `flags` field contains the flag `PUBLISH_CONSTANT`.

3.4.3 Publishing values as pointers.

This is the normal way for publishing values. A pointer to the data structure that is published is stored in the `cptr` field of the `PUBLISH` structure created for this publication. When the interface translates its data base into LISP objects, it creates a LISP object of type CObject which embeds the pointer. The pointer that is published must be a pointer to an integer, a double, a string, a structure or an array which type is published. The `flags` field will have the flag `PUBLISH_VARIABLE` set.

Integers, doubles and strings can be published as constants and as pointers. When published as a pointer, it is possible to change its value. Note however that when publishing a string as a pointer, what is actually published is a pointer to a pointer to char (`char **`). The value that is changed here is the value of the pointer to char (`char *`). When the value of that pointer is changed, there may be no way to deallocate the memory that was allocated to store the string.

3.5 Publishing functions.

3.5.1 Storing published functions.

Besides publishing data, the user must be able to publish functions. Those functions must be added to the interpreter when the interpreter is called. The interface has a special structure CFUNC, shown in figure 3.7, for storing information about those functions.

The CFUNC structures form a linked-list, so there is no limit to the number of functions that the user can publish.

```
typedef struct cfunction {
  STRING name;
  PRED   function;
  STRING doc;
  CFUNC *next;
};
```

Figure 3.7: The definition of the CFUNC structure.

The first field of the structure is the name of the function. The function will be bound to this name when the interface translates its data base to LISP objects.

The second field is a structure that contains information about a predefined function, such as the pointer to the function, the number of required arguments, the number of optional arguments and if an upper limit is set for the arguments. The third field contains a pointer to the documentation-string for this function.

The fourth field is a pointer to the next published function.

3.5.2 Publishing functions.

The functions that are published are added to the interpreter as normal predefined functions.

Appendix B provides an example of an interfaced data-structure.

3.6 Accessing the published data.

To access the published data, several special LISP functions are available. These include functions to get fields of structures, items from arrays, a test for NULL-pointers embedded in CObjects, functions to show the published types and published fields and of course the published functions.

The published values are added to the predefined bindings of the interpreter and they can be used like any other binding.

Changing data

Embedded pointers to integers, doubles or strings can be used to changed the value that they are pointing to. This is done using the function `setc`. The reader should note that published pointers to strings are really pointers to pointers to char (char **) and that what is really changed is the value of the pointer, not the data it is pointing to. The string that it is pointing to is left untouched.

Accessing fields of structures and items of arrays.

In section 3.3.3 we have discusssed how structures and arrays are published. This also explained how fields of these structures are published. To access a published field of a particular structure, the LISP function `requestfield` is used. The function returns an appropriate LISP object. This can be a CObject but also an integer, a real or a string if the field was published to be constant. The function `requestarray` is used to get an item of a published array. The function also returns the item as a CObject, integer, real or string.

If these functions return a CObject, this CObject embeds a pointer to a basic published type. If a field of type `double ***` was published, the result of getting it using `requestfield` is a CObject which embeds a pointer of type `double *` and not, as might be expected a pointer of type `double ***`. This is also true for items of arrays.

If a field is published as a basic publish type, say an integer, the result of getting the field with requestfield is a CObject that embeds a pointer of type `int *`. The pointer contains the address of the field inside the structure.

3.7 From read-access to write-access.

Our current interface is created mainly for read-access of external data structures. In the previous sections, we have seen that only published data of type integer, double or string could be changed using the LISP function `setc`. Changing this type of data is possible because the *size* and *format* of the type is known.

Although it is not possible to change other published data structures with `setc`, it is possible to publish functions that could do that. In most cases, this leads to difficulties. Consider the binary tree of figure 3.8. This tree is published by publishing the `node` structure and by publishing the pointer `root`.

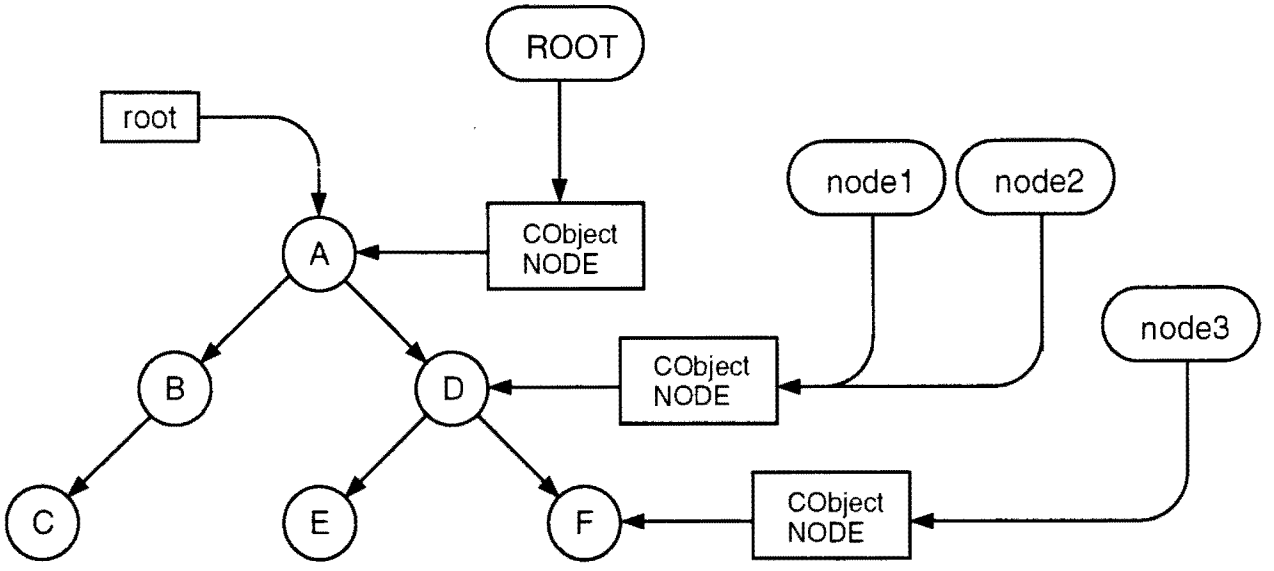


Figure 3.8: An interfaced binary tree.

Suppose that `node1` and `node2` were to be bound to another LISP object. Due to our reference count mechanism, this would cause the deallocation of the CObject pointing to node D. If a CObject refers to a node that is not a part of the interfaced tree then this node must also be deallocated. So, if it is possible to create nodes that are not part of the interfaced binary tree (*loose* nodes), it must be possible to distinguish nodes that are loose and nodes that are not. The best way to deal with this problem is to rely on the user not to publish functions that are able to create loose nodes.

Creating back-links

Suppose that the user has published a function to delete sub-trees and uses this function to delete node D. Since nodes E and F are part of the subtree rooted by node D, these nodes are also deleted. This leaves the CObject bound to the name `node3` pointing to a deallocated node. To avoid this, and avoid creating loose nodes, the pointer embedded in this CObject must be set to `NULL`.

The problem can be solved by adding a LISP object pointer to the node structure that is published. This pointer must point back to the CObject that embeds a pointer to the node. This back-link can be used to find the CObject that embeds a pointer to the node when that node is deleted. The reader should note that using this strategy, it is not possible to create two CObjects that refer to the same data structure, since the back-link can only point to one CObject. This is not a problem, really, since we can bind a CObject several times to different names. We only need one CObject per interfaced data structure.

If a node is requested using requestfield, the interface must look at the back-link of that node to find the CObject that embeds a pointer to the node. If this back-link is `NULL`, such a CObject does not yet exist an must be created. Otherwise, the existing CObject can be used.

Conclusion

We have discussed write-access using a binary tree as the interfaced data structure. We have seen that some problems occur that are solvable by adding a LISP object pointer to the external data structures. If the interface is to be used for write-access, the function `requestfield` must also be changed, so that it returns the CObject pointed to by the back-link if it exists.

chapter 4

Conclusions

An LISP interpreter is designed and implemented in **C**. This interpreter has an interface which allows to access external data structures of other programs, written in **C**. This combination of interpreter and interface can be used to define algorithms on such data structures.

The interpreter uses dynamic scope instead of static scope. This limits the interpreter in some ways, but this does not devaluate the use of the interpreter as a tool for interfacing data. The interpreter uses a rather unusual technique for garbage collection, by counting the references to LISP objects instead of other more usual methods. This reference count mechanism is especially suited for use with our interpreter since it spreads the time needed for the garbage collection. Other methods halt the interpreter while performing garbage collection which make them less friendly to the user.

The interface is capable of interfacing with virtually every data-structure in written in **C**. Only unions are not supported. Because of the flexiblity of the interface, it is not only possible to interface ESCAPE, but other programs as well, provided they are written in **C**. It is also possible to expand the interpreters functionality by publishing functions that are written in **C**. This feature can be used to make predefined LISP functions that can operate on specific data structures.

The interpreter has been succesfully interfaced with several test programs, however not yet with ESCAPE.

chapter 5

Recommendations

Although the tool has been tested with other programs, it is designed to be integrated with ESCAPE. Testing the tool with ESCAPE should be our first goal.

In the near future, the publication of structures should be extended with the possibility to specify the offset of an `OBJECT` pointer field which can be used as the back-link to the CObject that embeds the pointer to the structure. Along with this change, the function `RequestField` should be changed to use the back-link. This change would make write-access possible.

Also in the near future, the `BINDING`, `BINDINGSTACK` and `LOCALSTACK` structures should be placed on their own freelist, thus making the interpreter a bit faster again.

The interpreter should be extended with macros and possibly keywords.

The replacement of dynamic scope with static scope should be inverstigated.

As a final point, we suggest that a graphical user-interface is written for the interpreter, as replacement for the simple line-based text interface. The interpreter might then also support functions to open windows, and to print to and to draw in these windows.

Appendix A

Example of evaluating an user-function.

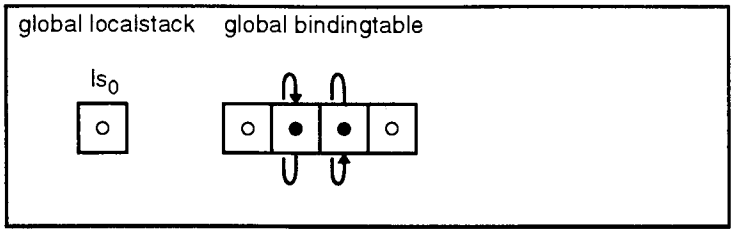
To illustrate how the bindingtables change, let us consider the following function `sos`, which calculates the sum of the squares of two arguments, `x` and `y`. To calculate the square of a number, another function is defined, `square`, which has only one argument, `x`. We will now look at a LISP session that evaluates a function call using `sos`. To simplify things, we ignore that the bindingtables are hashed and consider only one bindingtable. Also the predefined bindings are left out for clarity.

Remember that the localstacks are local to the function `EvalLambdaFunction`, and that only the localstack that is local to the current call to `EvalLambdaFunction` is visible from other functions, because the global variable `activestack` points to the top of that stack.

In the following example, user commands are shown in **bold** style, the interpreters results in *italic* style and the interpreters actions in indented, normal style.

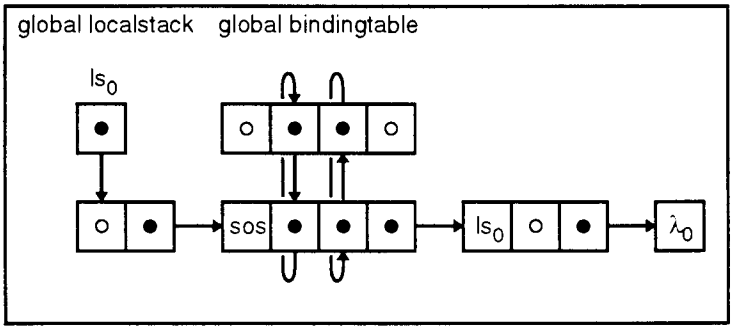
The example assumes that no bindings yet exist, so the initial state shows an empty bindingtable (except for the sentinel) and an empty global localstack.

Initial state:



> (defun sos (x y) (+ (square x) (square y)))

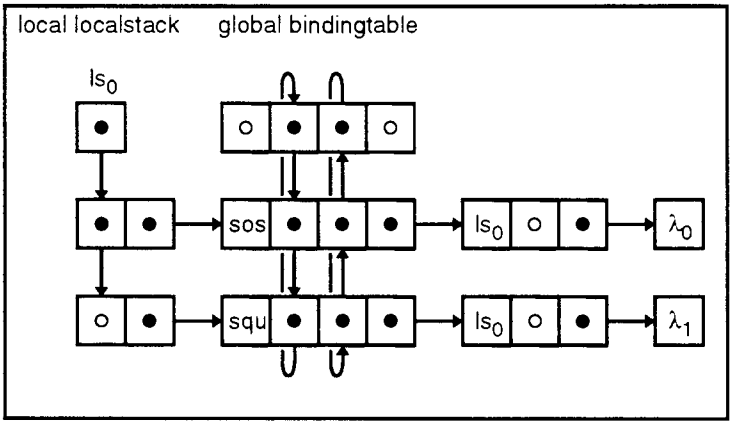
Create binding: sos → λ_0=(λ (x y) (+ (square x)(square y)))



sos

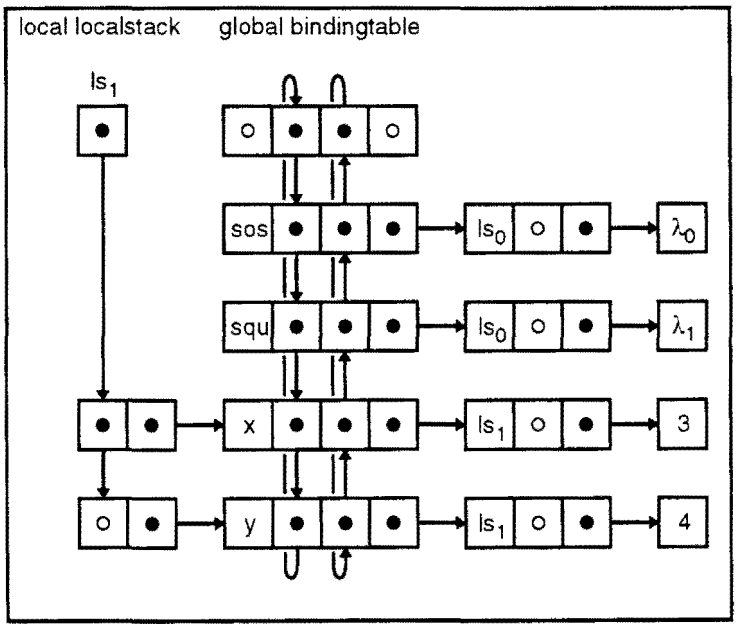
> (defun square (x) (* x x))

Create binding: square → λ_1=(λ (x) (* x x))



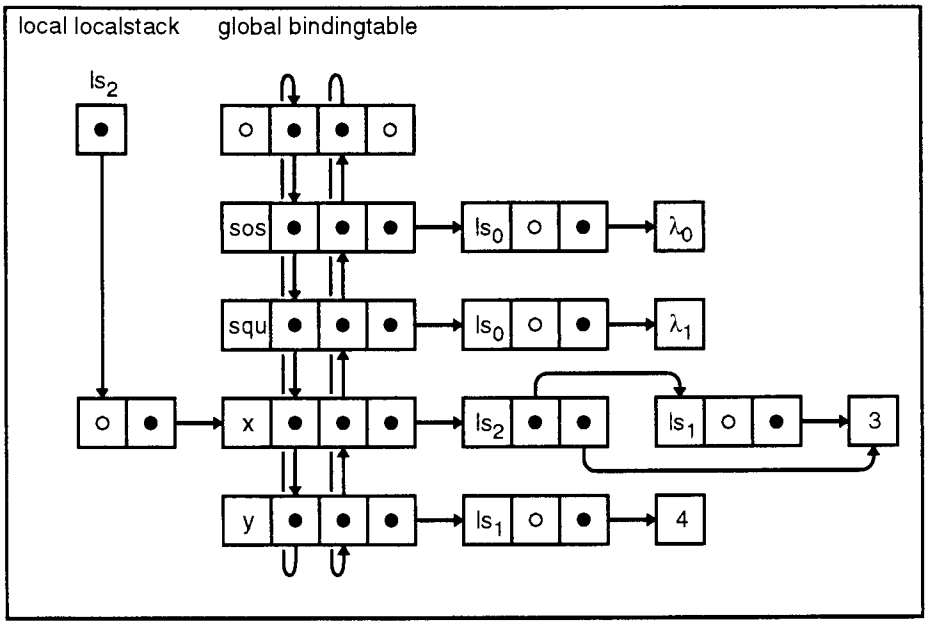
square

```
> (sos 3 4)

   Evaluate lambda-function: sos (λ0)
   parameterlist     : (x y)
   argumentlist      : (3 4)
   evaluate argument: 3
   result            : 3
   evaluate argument: 4
   result            : 4
   create bindings   : x → 3, y → 4
```



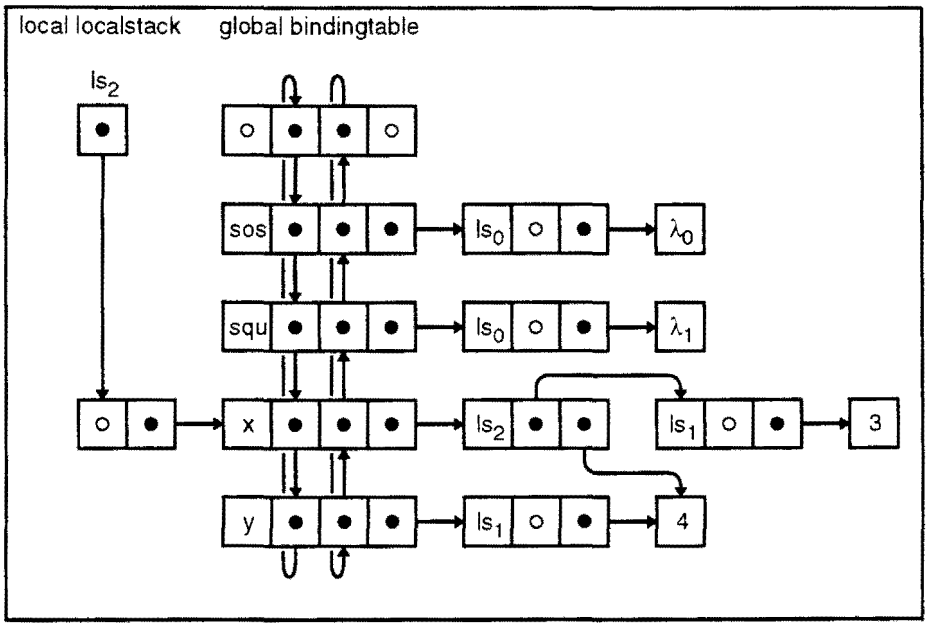

```
   evaluate body     : (+ (square x) (square y))

         Evaluate predefined function: +
         argument list     : ((square x) (square y))
         evaluate argument: (square x)
```

```
Evaluate lambda-function: square (λ1)
parameterlist     : (x)
argumentlist      : (x)
evaluate argument: x
result            : 3
create binding    : x → 3
```




```
evaluate body     : (* x x)

      Evaluate predefined function: *
      argument list     : (x x)
      evaluate argument: x
      result            : 3
      evaluate argument: x
      result            : 3
      Return: (* 3 3) = 9

result            : 9
remove local bindings: x
Return: 9

result            : 9
evaluate argument: (square y)
```

```
Evaluate lambda-function: square (λ1)
parameterlist     : (x)
argumentlist      : (y)
evaluate argument: y
result            : 4
create binding    : x → 4
```




```
evaluate body     : (* x x)

      Evaluate predefined function: *
      argument list     : (x x)
      evaluate argument: x
      result            : 4
      evaluate argument: x
      result            : 4
      Return: (* 4 4) = 16

result            : 16
remove local bindings: x
Return: 16

result            : 16
Return: (+ 9 16) = 25

result            : 25
remove local bindings: x, y
Return            : 25
```

25

Appendix B

An example of interfacing a data-structure.

In this appendix we present an example of an interfaced data-structure. Although this data-structure is by far not as complex as the data structures of ESCAPE, the reader should get a good idea of how data is interfaced.

The example describes how a binary-tree structure is published. The C-definition for the structure used to create binary trees is shown in figure B.1.

```
struct node {
  CPROPERTY   *properties;
  char        *info;
  char        *private;
  struct node *leftchild;
  struct node *rightchild;
};
```

Figure B.1: Definition of the node structure.

The tree is published by publishing the node-structure type and publishing the pointer `root`, which points to the root of the binary tree that is to be published. This tree is shown in figure B.2.

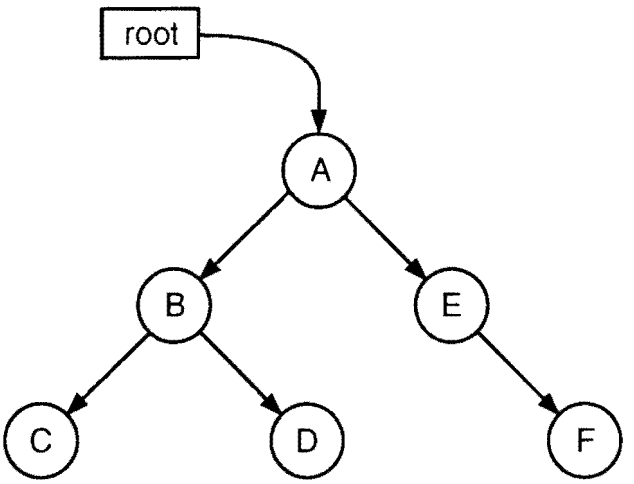


Figure B.2: The published tree.

The interfaced program publishes the type and the pointer to the tree:

```
PublishInit();

PublishStruct("NODE",sizeof(struct node),
FIELD_OFFSET(struct node,properties));

PublishField("node-info","NODE","STRING",
FIELD_OFFSET(struct node,info), CONSTANT);

PublishField("node-left","NODE","NODE*",
FIELD_OFFSET(struct node,leftchild), VARIABLE);

PublishField("node-right","NODE","NODE*",
FIELD_OFFSET(struct node,rightchild), VARIABLE);

PublishVariable("ROOT","NODE",(void *) root);
```

The LISP interpreter is then called with as argument one filename:

```
filename = "bintree.lsp";
LispInterpreter(1,&filename);
```

The file contains definitions for functions that operate on the published tree. The functions that are used for accessing published data are shown in **bold** style:

```
(defun depth (bintree) "(depth <NODE>)"
 (if (cnull bintree) 0
  (++ (max (depth (requestfield bintree "node-left"))
           (depth (requestfield bintree "node-right"))
      )
  )
 )
)

(defun numofleaves (bintree) "(numofleaves <NODE>)"
 (if (cnull bintree) 0
  (if (= (setq templeaves
           (+ (numofleaves (requestfield bintree "node-left"))
              (numofleaves (requestfield bintree "node-right"))
           )
         )
         0
       )
       1
     templeaves
  )
 )
)
```

```
(defun numofnodes (bintree) "(numofnodes <NODE>)"
 (if (cnull bintree)
     0
     (+ 1 (numofnodes (requestfield bintree "node-left"))
          (numofnodes (requestfield bintree "node-right"))
     )
 )
)

(defun preorder (bintree) "(preorder <NODE>)"
 (if (cnull bintree)
     nil
     (progn
      (print (requestfield bintree "node-info"))
      (preorder (requestfield bintree "node-left"))
       (preorder (requestfield bintree "node-right"))
     )
 )
)

(defun postorder (bintree) "(postorder <NODE>)"
  (if (cnull bintree)
      nil
      (progn
        (postorder (requestfield bintree "node-left"))
        (postorder (requestfield bintree "node-right"))
        (print (requestfield bintree "node-info"))
       )
 )
)
```

```
(defun inorder (bintree) "(inorder <NODE>)"
 (if (cnull bintree)
     nil
     (progn
      (inorder (requestfield bintree "node-left"))
      (print (requestfield bintree "node-info"))
      (inorder (requestfield bintree "node-right"))
     )
 )
)

(printn "depth of binary tree :" (depth ROOT))
(printn "number of leaves     :" (numofleaves ROOT))
(printn "number of nodes      :" (numofnodes ROOT))
(printn "preorder traversal   :" (preorder ROOT))
(printn "postorder traversal  :" (postorder ROOT))
(printn "inorder traversal    :" (inorder ROOT))
```

This produces the following output:

```
depth of binary tree : 3
number of leaves     : 3
number of nodes      : 6
preorder traversal   : A B C D E F
postorder traversal  : C D B F E A
inorder traversal    : C B D A E F
```

Appendix C

The interpreter in pseudo-code.

The figures C.1 to C.8 describe the LISP interpreter in pseudo-code. Code in *italic* style is not explained further. It is not the intention to specify the interpreter completely. The appendix merely gives the reader a quit view on the basic structure of the interpreter.

Figure C.1 describes the main loop which is also called the Read-Eval-Print loop.

```
proc main =
|[var readobject,evalobject:LISP OBJECT
| Initialise interpreter;
  while QUITFLAG not set
  do
    while QUITFLAG and RESTARTFLAG not set
    do
      readobject := ReadForm;
      evalobject := EvaluateForm(readobject);
      PrintObject(evalobject);
    od
    if RESTARTFLAG is set
      Initialise interpreter;
    fi
  od
  Uninitialise interpreter;
]|
```

Figure C.1: The main procedure, the read-eval-print loop.

```
proc ReadForm = (out returnobj:LISP OBJECT
| |[var character:char
  |
    character := Get next character;
    case character of
    '(': do
           returnobj := Read a list;
         od
    '\'': do
            returnobj := Read a quoted form;
          od
    '-','+': do
               if Next character is not a digit
                 returnobj := Read a symbol;
               else
                 returnobj := Read a number;
               fi
             od
    digit: do
             returnobj := Read a number;
           od
    '"': do
           returnobj := Read a string;
         od
    '.': do
           if Next character is a digit
             returnobj := Read a number;
           else
             Report a misplaced-dot error;
           fi
         od
    '{': do
           returnobj := Read a bitvector;
         od
    ')': do
           Report a misplaced-dot error;
         od
    otherwise: do
                 returnobj := Read a symbol;
               od
    esac
  ]|
)
```

Figure C.2: The ReadForm function.

```
proc EvaluateForm = (out returnobj:LISP OBJECT;
in object:LISP OBJECT
| |[case LISP type of object of
    cons: do
            returnobj := EvaluateCons(object);
          od
    symbol: do
              returnobj := EvaluateSymbol(object);
            od
    CObject: do
               returnobj := EvaluateCObject(object);
             od
    otherwise: do
                 returnobj := object;
               od
    esac
  ]|
)
```

Figure C.3: The EvaluateForm function.

```
proc EvaluateCObject = (out returnobj:LISP OBJECT;
in object:LISP OBJECT
| |[case C-type of pointer in object of
    C_INTEGER: do
                 returnobj := Create integer LISP object;
               od
    C_DOUBLE: do
                returnobj := Create real LISP object;
              od
    C_STRING: do
                returnobj := Create string LISP object;
              od
    otherwise: do
                 returnobj := object;
               od
    esac
  ]|
)
```

Figure C.4: The EvaluateCObject function.

```
proc EvaluateCons = (out returnobj:LISP OBJECT; in object:LISP OBJECT
| |[var function,args:LISP OBJECT
  | args := CDR of object;
    function := EvaluateForm(CAR of object);
    if Too few or too many arguments in args for function
      Report too-many-or-too-few-arguments error;
    else
      case LISP type of function of
      lambda function:
        do
          returnobj := EvalLambdaFunction(function,args);
        od
      predefined function: do
                             returnobj := Call the C-code;
                           od
      otherwise: do
                 Report a void-function-value error;
                 od
      esac
    fi
  ]|
)
```

Figure C.5: The EvaluateCons function.

```
proc EvaluateSymbol = (out returnobj:LISP OBJECT;
in object:LISP OBJECT
| |[var binding:BINDING;
  | binding := Find the binding for the name of object;
    if binding is NULL
      Report an unbound-symbol error;
    else
      returnobj := LISP object bound by binding;
    fi
  ]|
)
```

Figure C.6: The EvaluateSymbol function.

```
proc EvalLambdaFunction = (out returnobj:LISP OBJECT;
in function:LISP OBJECT, args:LISP OBJECT;
| |[var localstack:LOCALSTACK,body,evalargs:LISP OBJECT,nargs:int;
  | evalargs := Evaluate arguments args;
    nargs := Number of arguments;
    localstack := activestack;
    activestack := New LOCALSTACK structure;
    Create new bindings by binding the required parameters to
    the result of evaluating the argument;
    nargs := nargs - Number of required parameters;
    while nargs > 0 and optional parameters not bound
    do
      Create binding for a optional parameter by binding it to the
      next argument-evaluation result;
      nargs := nargs - 1;
    od
    while optional parameters not bound
    do
      Bind the next optional parameters to nil or to another
      initial value that is determined by the user;
    od
    if rest parameter specified
      if nargs > 0
        put the rest of the argument-evaluation result in a list
        and bind this list to the rest parameter;
      else
        bind the rest parameter to nil;
      fi
    fi
    body := List of expressions that form the body of the function;
    while body is not NULL
    do
      returnobj := EvaluateForm(CAR of body);
      body := CDR of body;
    od
    Empty activestack and remove the bindings on it;
    activestack := localstack;
  ]|
)
```

Figure C.7: The EvalLambdaFunction function.

```
proc PrintForm = (in object:LISP OBJECT
| |[Print a textual representation of object;
  ]|
)
```

Figure C.8: The PrintForm function.

Appendix D

D.1 Publishmanagers' functions.

Type definition functions:
`PublishStruct(STRING typename, int typesize, int propoffset)`
`PublishArray(STRING typename,STRING itemtype,int arraysize,int flags)`

Data publication functions:
`PublishVariable(STRING valuename, STRING typename, void *valueptr)`
`PublishInteger(STRING valuename, int value)`
`PublishDouble(STRING valuename, double value)`
`PublishString(STRING valuename, STRING value)`

Function publication function:
`PublishFunction(STRING functionname,STRING docstring, int nrequired, int noptional, BOOLEAN rest, OBJECT *(*functionptr)())`

Publication undo functions:
`UnPublishValue(STRING valuename)`
`UnPublishType(int ctypeid)`
`UnPublishField(STRING fieldname, STRING typename)`
`UnPublishFunction(STRING functionname)`

Additional functions:

```
PublishProcess()
PublishInit()
PublishClear()
PublishReset()

PublishNumberType(int ctypeid)
PublishGetTypeId(STRING typename)

RequestArray(OBJECT *cobject, int arrayitemindex)
RequestField(OBJECT *cobject, CFIELD *cfield)
```

D.2 LISP functions that operate on published data.

Predicates:

```
(cnull <cobject>)
(cobjectp <cobject>)
(ctype <cobject>)
```

CProperty functions:

```
(getcprop <cobject> <string>)
(listcprop <cobject>)
(remcprop <cobject> <string>)
(setcprop <cobject> <string> <form>)
```

Field and item request functions:

```
(requestarray <cobject> <integer>)
(requestfield <cobject> <string>|<integer>)
```

Value change function:
(setc {<cobject> <number>|<string>}*)

Auxiliary functions:
(getctypeid <string>|<cobject>)
(getctypename <integer>)
(getfieldid <string> <string>|<integer>|<cobject>)
(getfieldtype <string> <string>|<integer>|<cobject>)
(resubscribe)
(showctypes)
(showfields)

B

Bibliography

[Abelson,85] H. Abelson, G.J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass., 1985.

[Aho,82] A. V. Aho, *Data structures and algorithms*, Addison-Wesley, Reading, Mass., 1982.

[Aho,86] A. V. Aho, R.Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addision-Wesley, Reading, Mass., 1986.

[Barrett,86] W. Barrett, R. Bates, D. Gustafson, J. Couch, *Compiler Construction: Theory and Practice*, 2ne Edition, SRA, Chicago, Illinois, 1986.

[Cohen,81] J. Cohen, "Garbage collection of linked data structures", *ACM Computing Surveys*, 13(2), 1981, pp. 341-368.

[Friedman,74] D. P. Friedman, *The Little LISPer*, SRA, Chicago, Illinois, 1974.

[Hoppcroft,79] J. Hopcroft, J. Ullman, I*ntroduction to Automata Theory, Langauges, and Computation*, Addision-Wesley, Reading, Mass., 1979.

[Kamin,90] S. N. Kamin, *Programming Languages*, Addison-Wesley, Reading, Mass., 1990.

[Knuth,73] D. E. Knuth, *The art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, 2nd Edition, Addison-Wesley, Reading, Mass., 1973.

[MacCarthy,60] J. MacCarthy, "Recursive functions of symbolic expressions and their computation by machine", *Comm. ACM* 3(4), April 1960, pp. 184-195.

[MacCarthy,62] J. MacCarthy, *LISP 1.5 Programmer's Manual*, The MIT Press, Cambridge, Mass., 1962 (reprinted in Horowitz [1983]).

[MacLennan,87] B. MacLennan, *Principles of Programming Languages*, 2nd Edition, Holt, Rinehart and Winston, New York, 1982.

[Marcotty,86] M. Marcotty, H.F. Ledgard, *Programming Language Landscape*, 2nd Edition, SRA, Chicago, Illinois, 1986.

[Sethi,88] R. Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading, Mass., 1988.

[Standish,80] T. A. Standish, *Data Structure Techniques*, Addison-Wesley, Reading, Mass., 1980.

[Steele,84] G. L. Steele, Jr., "Multiprocessing compactifying garbage collection", *Comm. ACM* 18, 9, Sept. 1975, pp. 495-508.

[Steele,84] G. L. Steele, Jr., *Common LISP: The Language*, Digital Press, Burlington, Mass., 1984.

[Touretzky,84] D. Touretzky, *LISP: A Gentle Introduction to Symbolic Computation*, Harper-Row, New York, 1984.

[Wilensky,86] R. Wilensky, *Common LISPcraft*, Norton, New York, 1986.

[Winston,84] P. H. Winston, B. K. Horn, *LISP*, 2nd Edition, Addision-Wesley, Reading, Mass., 1984.

G

Glossary

α-conversion
This is the name that is used by theorists to indicate that the meaning of a certain function does not change when the names of the local variables (and parameters) are changed. Static scoped languages abide by this α-conversion, dynamical scoped languages don't.

Binding
A binding is a 'connection' of a name and a LISP object. A name can be bound to many different objects. Only one binding is considered to be the *effective* binding of that name. The LISP object that is bound to a name by the effective binding of that name can be considered to be the value of the variable of that name.

Bitvector
A bitvector is a sequence of bits. The length of the bitvector is defined to be the number of bits that form the sequence. The bits of a bitvector are numbered from left-to-right and the numbering starts with 0.

BNF
Backus-Naur-Form, a method of describing the syntax of programming languages. References [Aho,86], [Barrett,86] and [Hopcroft,79].

C
A programming language developed to provide high level control structures together with low level features such as the ability to assign hard addresses to pointer variables.

Car

This is an abbreviation for "contents of address register" which refers to an implementation of LISP on a IBM 704, in machine language. It is used to specify one of the two pointers that are contained by a cons-cell. When a cons-cell is part of a list, the car-pointer is used to point to the member of the list.

Cdr

This is an abbreviation for "contents of decrement register" which refers to an implementation of LISP on a IBM 704, in machine language. It is used to specify one of the two pointers that are contained by a cons-cell. When a cons-cell is part of a list, the cdr-pointer is used to point to the next cons-cell of the list. The list terminates when the cdr-pointer of a cons-cell points to something else as a cons-cell. The normal way to terminate a list is by having the cdr-pointer point to `NULL`.

Closure

A closure is a set of bindings together with a user-function. The bindings are created to preserve the value of so-called free variables when a function is defined. Free variables are variables that do not occur in the list of parameters for the function and are also not global variables. This technique is only used by static scoped LISP and not by dynamic scoped LISP. Closures premit full λ-calculus, but use memory and extra time searching for the effective binding of a name.

CObject

This is a special type of LISP object, used to interface external data. The CObject contains a generic **C**-pointer (void *) which is used to point to an external data-structure. The type of this external data is specified by another field, the type-indicator.

Cons-cell

A LISP object of type cons. This type of object contains two pointers to other LISP objects. By linking cons-cells, lists are formed. Because of the unusual type of memory management, circular lists are not allowed.

Dynamic scope
A name for the scope method traditionally used by LISp interpreters and that is also used by our interpreter. The effective binding of a name is considered to be the most recently made, still surviving binding. This definition mentions surviving because bindings can be deleted when user-functions are completely evaluated.

ESCAPE
A tool, developed by the Department of Design Automation, for fast prototyping of initial ideas at a high level of abstraction and developing a high level description of a design.

Evaluator
The evaluator is a module that contains functions to evaluate LISP objects.

Functional programming
A style of programming, where the user defines functions instead of writing programs, and applies these functions to arguments instead of running those programs.

Lexical or statical scope
Lexical scope is a name for the way that Common LISP, Scheme and Pascal detremine the effective binding of a name, based on the text of the program. The effective binding of a name is that binding that was created by an expression that is 'nearest' to the use of the name as a variable, based on the text of the program.

LIFO
This stands for last-in, first-out. This term is used to describe the time-order of events. The evaluation of user-functions in LISP is a LIFO process.

LISP
A programming language, developed by John MacCarthy [MacCarthy,60] for symbolic computation.

LISP object
LISP objects are records with a field that indicates the type of the object and a field that contains the value of the object. In this implementation the last field is implemented uing the union type of **C**.

Nil

`Nil` is a special type of LISP. There is only one object of this type. This object is used to indicate that a function could not return anything useful or that the result is a logical false. `Nil` is also used to terminate lists, at least from the users point of view. Internally this representation of `nil` ending a list is implemented by having the last cdr-pointer of that list point to `NULL`.

Printer

The printer is a module that contains functions that produce a textual representation of a LISP object.

PublishManager

The `publishmanager` is the name given for a collection of functions and data that together form the interface between a **C**-program an the interpreter.

Reader

The reader is a module that contains functions that read a textual expression from an input source and translate this text into a LISP object.

Read-Eval-Print loop

This is the main loop of the interpreter, which uses the reader module, the evaluator module and the printer module to read an expression, translate it to a LISP object, to evaluate that object and to print the result.

Sentinel

A sentinel is an empty entry in a double-linked list which makes the list circular. This is done by replacing all references to NULL by references to the sentinel. This way, there is no need for 'boundary' checking which makes dealing with these lists easier.