Identifier Semantics: A Matter of References

Technical Report LIX RR 89 02, pp 67–80,
École Polytechnique, France, 1989.

Greg Nuyens[a]
ILOG

2, Avenue Gallieni
94025 Gentilly – France

Internet: nuyens@ilog.fr

Christian Queinnec[a]
Laboratoire d'Informatique Théorique de Programmation

4, Place Jussieu
75252 PARIS Cedex 05 – France

Internet: queinnec@inria.inria.fr

[a]partially funded by SERICS contract # 87 1 8 13 8 00

[a]partially funded by GRECO de Programmation

Abstract

We present a clear semantics for identifier use. Using precise terminology, this semantics accounts for: the creation, use, modification and visibility of bindings, as well as the mention of identifiers. It covers current practice in Lisp-like languages, while incorporating features usually found outside Lisp. This semantics is shown to be viable for both Lisp_1 (unified function and value cell Lisp) and Lisp_2. A syntactic device is included which allows concise expression without sacrificing generality. The ensemble has been demonstrated in a meta-circular version, while a production version for the next release of Le-Lisp[1] is in progress. The associated denotational semantics will figure in an annex of the full paper.

1 Introduction

Computing formalisms generally include the ability to bind an identifier to domain values and to remotely reference the binding with the identifier. The semantic and syntactic treatment of this binding has varied widely among computer languages. Because historically the expressive power of these languages has increased incrementally, binding and reference within multiple namespaces is not always consistent. With an eye to providing a general consistent mechanism for binding and reference, let's consider aspects of binding in a few example languages.

[1]Le-Lisp is a trademark of INRIA.

Aspect I. Static vs Dynamic Variable Binding

MacLisp

```
(defun foo () x)

(defun bar (x)
   (foo))

(setq x 4)

(bar 3) ;⇒ 3
```

Pascal

```
Program Main;
var x: integer;
    Function foo : integer;
    begin
        foo := x;
    end;

    Function bar (x :integer) : integer;
    begin
        bar := foo;
    end;

begin
x := 4;
bar(3)   { will return 4 }
end.
```

These examples demonstrate the difference between dynamic and lexical variable scoping. The difference is of course whether references are determined at the binding declaration site, or determined in the binding execution site. The important point is that even though the references to **x** in MacLisp [8] and Pascal [6] have different meanings, the reference is unambigous, each language has a single scoping rule for variables.

Aspect II. Separation of Name Spaces

APL

```
        bar ← 4

    ∇   r ← bar x
 [1]    r ← 3
    ∇

        bar ⇒ 3
```

MacLisp

```
(setq bar 4)

(defun bar (bar) bar)

(funcall 'bar 3) ;⇒ 3
bar ;⇒ 4
```

In APL [5] niladic function application and variable reference are syntactically indistinguishable, and thus share the namespace. In MacLisp [8] **funcall**, they coincide, but we see that functions and variables have separate namespaces.

Aspect III. Single Variable Name Space

Common Lisp

```
 (defun bar (foo)
   (equal
     (locally (declare (special foo))
         foo)
     foo))

 (let ((foo 3))
      (declare (special foo))
      (bar 4)) ;⇒ nil
```

In Common Lisp, in spite of sharing the identifier, static and dynamic variable bindings are completely independent. In Lisp 1.5 [7], APL, Pascal and R^3 Scheme [11] the question doesn't arise since there is a single variable scoping rule.

Aspect IV. Simultaneous Access to Multiple Variable Name Spaces

Common Lisp:

```
(let ((var 'special-val))
    (declare (special var))
    (let ((var 'lexical-val))
        (eq
           (locally (declare (special var))
               var) ;⇒ special-val
           var ;⇒ lexical-val
    ))) ;⇒ nil
```

Chez Scheme:

```
(define var 'global-val)

(fluid-let ((var 'special-val))
    (let ((var 'lexical-val))
        (eq
            ;** we can't reference the
            ;fluid binding since it has
            ;been lexically shadowed.
            var)))
```

Though Common Lisp provides simultaneous access to the lexical and dynamic bindings, `(locally (declare (special ...)))` is less concise than below.

PC Scheme:

```
(define var 'global-val)

(fluid-let ((var 'special-val))
    (let ((var 'lexical-val))
        (eq
             (fluid var);⇒ special-val
             var ;⇒ lexical-val
))) ;⇒ nil
```

The large difference between the treatment of multiple variable namespaces in PC Scheme [18] and Chez Scheme [3] demonstrate the range of possible solutions. (Chez Scheme has both lexical and dynamic scoping rules, but only one environment). PC Scheme's solution is similar to that proposed in [9]. A variety of choices is analyzed in [2].

Aspect V. Constant Data

Constant data (when available) takes one of two forms: constant bindings or constant data objects. Since Pascal constrains the value of constants to instances of built-in first-class atomic types, it avoids the distinction. However, one can rebind an identifier which has an active constant binding.

In Common Lisp, `defconstant` declares a constant dynamic binding which may not be modified nor may the identifier be dynamically rebound. The atomicity question is not addressed, since it would require implementation support to forbid physical modification of composite types. T's `define` is distinguished from `lset` by the creation of immutable bindings.

Aspect VI. Global Bindings

Interactive languages have the special property that global bindings can be affected/created (more on this distinction later) after capture of the global environment.

Common Lisp

```
(setq foo
    (lambda () (bar)))
(defun bar ()
    'new-value)
   (foo) ;⇒ new-value.
```

In strict lexical scoping, with `defun` considered to modify the "outermost" or "built-in" lexical scope, `bar` would not have been present in the closed-over environment of the closure in `foo`. This forward-reference (`letrec`) ability is an important aspect of the toplevel, and precludes strict lexical scoping.

Global references (that are affected by neither lexical nor dynamic bindings) exist in various languages. For example, Interlisp's [17] `GETTOPVAL`, and the `access` primitive of T allow direct access to the global environment. In the design of Common Lisp, direct access to the global environment is probably excluded because of the expense in shallow-bound implementations.

2 Terminology

In order to precisely discuss the aspects of the preceding examples, we will need a few terms:

identifier a sequence of characters used as a name. In various languages, identifiers may name: variables, functions, special forms, data locations (including the data structure "`Symbol`"). code locations, etc.

location an element of the store of a computer capable of containing a reference to any object (not necessarily first-class).

binding an assocation between an identifier and a location. These are created with binding forms. (A Pascal declaration, `let` and `block` in Lisp, etc.)

variable a particular case of binding that can contain any first-class object. Note that variables are not defined to be identifiers that name a location (as in [4]) because they also occur in intentional references (e.g. free references) which need not name a location at all times.

reference a use of an identifier within the scope of a binding form to intentionally refer to the associated location or its contents. In this paper we instead prefer *binding mention* to distinguish the general case (reference) from the particular use to access the contents of a binding (dereference).

scope the set of calculations within which a binding has an effect.

namespace a domain in which identifier equality implies referent equality.

environment a collection of bindings within a single namespace existing at a given time.

3 Questions

In the preceding terms, the treatment of a reference comprises the following steps:

1. What is the reference (namespace plus identifier) intended by the mention of an identifier?
2. What is the location (which must be found via the relevant binding form and associated environment) intended by the reference?
3. What are the legitimate operations on this location?

Step 1

In most languages, the local context uniquely determines the name space which is required together with the identifier to determine a reference. For example, `(go label)` in Common Lisp, requires a tag, so the reference is `(tag label)`. In R^3 Scheme, there is only one namespace. So in both `(label 1 2 3)` and `(car label)` the identifier `label` determines the same reference which is `(identifier label)`. The fact that the scoping rule for identifiers is lexical is orthogonal.

Since unqualified identifiers may refer to two variable namespaces in COMMON LISP, declarations (via `declare, the` and `proclaim`) determine the namespace intended (thus answering Question 1). For example `(locally (declare (special x)) x)` the reference is `(dynamic x)`. In `(let ((x 3)) x)`, the reference is `(lexical x)`.

Step 2

Question 1's answer provides a reference. Within a given language and implementation there are well-known implementations of the binding representation and lookup [12]. Of course, reference semantics (e.g. lexical variables in Pascal or COMMON LISP) that permit a static implementation are preferable for performance reasons. Contrast this with dynamic variable lookup in APL or Lisp 1.5 where step 2 cannot be accomplished before run-time. Note that shallow binding trades off function call speed to allow Step 2 to be determined at read-time.

Step 3

The legal operations for a given location in a given language are clear. One can "go" to the contents of the location named by the reference `(tag label)`. But one cannot assign to the location. In Pascal, one can read the location named by the reference `(global name)`, but it may be constant, and hence unwritable. See 4.4 for the particular case of Lisp variables.

4 Binding/Reference scheme

We now suggest a binding/reference scheme for Lisp that incorporates our desiderata. Bearing in mind the the notion of binding and reference as they are employed in various computer languages, can we combine the best aspects of these binding/reference frameworks:

- multiple variable scoping regimes (COMMON LISP, PC Scheme)
- constant binding (Pascal, COMMON LISP)
- global environment (Lisp in general)

while avoiding their pitfalls?

- asymmetric access to bindings.
- insufficient generality (not possible everywhere, nor for every type.)
- insufficient control of creation of and access to global bindings.

Yes! Let's examine how each of these requirements dictates aspects of the final design of a binding/reference framework. We will first develop this solution within a Lisp_1 (unified function/value cell Lisp) and later show how it naturally extends to Lisp_2.

4.1 Multiple Variable Scoping Regimes

The experience of COMMON LISP and PC Scheme have shown the utility of having both lexical and dynamic variable binding in Lisp-like languages [16, 14]. However, both scoping regimes must enjoy equal status for both binding and reference. For example, consider the COMMON LISP fragment (derived from the example in aspect IV but with the order of lexical and dynamic binding inverted.)

```
(let ((x 'lexical-value))     ;lexical binding
   . .
     .
  (let ((x 'dynamic-value)) ;dynamic binding
       (declare (special x))
     (list
         x ;reference to the dynamic binding ⇒ 'dynamic-value.
         ?? ) ;no reference permits access to the lexical binding...
   ))
```

(1)

This example shows the absence of an consistent reference to bindings. COMMON LISP provides `(locally (declare ...))` to treat the case of specifying references different from the binding, but it treats only local dynamic reference.

These considerations encourage a separation of scoping specification for each type of reference permitted (to this point, lexical and dynamic.) The generalized `let` construct of our solution provides an introduction to the forms of reference allowed:

Initial description of `with-binding`:

```
with-binding-form     ::= (WITH-BINDING binding* body)
binding               ::= (mention initial-value-form)
mention               ::= ( name-space identifier )
name-space            ::= [ dynamic | lexical ]
body                  ::= form*
```

e.g.

```
(with-binding (((dynamic x) ;creation
                3)
       ((lexical y)  4)
       ((lexical v)  4))
          :
       (setf (dynamic z) 3) ;modification
       (setf (lexical v) 5)
          :
       (list (dynamic x) (lexical y) (dynamic z) (lexical v))) ;dereference


⇒ (3 4 3 5)
```

Here we see the three different uses of a binding mention:

creation a mention here is to create a binding. The same mention in the other two contexts will refer to the binding created here.

modification here the mention refers to the location of the created binding.

dereference the mention refers to the contents of that location.

So (1) would be written as follows:

```
(with-binding (((lexical x) 'lexical-value))     ;lexical binding
     . .
       .
   (with-binding (((dynamic x) 'dynamic-value)) ;dynamic binding
     (list
        (dynamic x)
        (lexical x))))
```

All reference forms (`lexical` and `dynamic` to this point) are special forms which accomplish Step 2 in the binding/reference model.

4.2 Global Reference

The primary interest of a global reference is the possibility to avoid variable shadowing. In this sense, global binding must be distinguished from "outermost lexical binding" which is the Pascal view. The Pascal view provides no new abilities, since it is simply the outermost lexical binding. No direct access is provided and intervening lexical bindings shadow the global value.

Is direct access to the global environment interesting? Yes, consider the problem of variable capture in macro expansion. Direct global access allows macro expansions to be sure that no matter where they are employed, no accidental capture will occur. (of course, simply expanding in a null environment (lexical, dynamic, or any other) only guarantees that the expansion will be the same in all cases. However, free variable references in the expansion, including functions in Lisp_1, are anyone's guess.)

So, we extend the reference forms with a name-space "global". This provides direct access to the global environment. This resembles Interlisp's `GETTOPVAL` and T's `access` (when given a named toplevel `locale`.) On the other hand, Common Lisp provides access to the toplevel by declaring a toplevel constant binding (with `defconstant`). Access in fact, follows the dynamic chain, but since all intervening bindings have been forbidden, this is guaranteed to be the toplevel. This is unnecessarily restricting, but is efficiently implementable in a shallow bound Lisp. The following is an example of using `global` in the aforementioned macro case:

```
(setf (global count) 0) ;modifying the "pre-existing" binding.

(defun write-byte (char stream)
  ;counts the number of bytes that have been written to all streams.
  (setf (global count) (+ 1 (global count))))
```

No matter where `write-byte` is called, there is no possibility of access/modification of the wrong binding for `count`. This is the primary reason to provide global reference/binding.

4.3 Extendable Environments

What else distinguishes the global environment from simply an outermost environment? It is that the global environment is extendable, while all other environments are fixed in number of bindings at the time of creation. Also, all new binding creation (Lisp's toplevel `setq`, and Scheme's `define`, but not Scheme's `setq`) will happen at this level. Finally, a global environment usually provides an "indefinite `letrec`". (To ensure that forward references are not considered errors, the global environment must make it appear that a binding (to unbound) pre-exists for every possible global reference). These three facts are what produce the behavior we expect from a "toplevel".

However, most languages provide the global behavior exactly once. This means that interactive languages cannot have multiple independent read-eval-print loops. T makes a positive step in trying to provide the "toplevel" ability in a general way. A `locale` guarantees the `letrec` and new binding capture. However, it does not guarantee the semantics of forward reference. Bearing in mind the behavior in example VI, consider the following T code:

```
(let ((even (lambda (n) n)))
    (locale ()
       (define odd (lambda (n)
          (or (= n 1)
              (even (1- n)) ) )) ;(lexical even) or ("forward" even)?

       (odd 1) ;⇒ T
       (odd 2) ;⇒ undefined ??

       (define even (lambda (n)
          (or (= n 0)
              (odd (1- n)))))
```

```
(odd 3)) ;if pure letrec then ⇒ T
         ;else ⇒ 2
```

T does not guarantee whether the reference will be considered forward or lexical to the enclosing lexical contour. If the choice were forward T's `locale` would satisfy our criteria for "toplevel".

4.4 Mutability

Immutablility serves the following major purposes:

1. It serves as a signal to human readers of the code that the immutable object is never modified.
2. It justifies to the compiler that zealous optimization is safe.
3. It allows inadvertent modification to be trapped.

Immutability of data types is generally provided by not offering destructive operations on the type. (Static zones can provide still more protection and allow sharing at the multi-user level, but presuppose sophisticated memory management.) In the particular case of bindings, mutability is verified by the binding operators. (Like all other properties of bindings, mutability can be statically verified in the lexical case, and must be dynamically checked for dynamic and global binding modification).

Thus we extend `with-binding` as follows

```
with-binding-form     ::= (WITH-BINDING binding* body)
binding               ::= (mention initial-value-form mutability)
mutability            ::= [mutable | immutable]
mention               ::= ( name-space identifier )
name-space            ::= [ dynamic | lexical | global ]
body                  ::= form*
```

Given the introduction of `global`, we needn't exclude rebinding of an identifier which has a constant binding. For example:

```
(with-binding (((dynamic flag) '(fer . sure) immutable))
    (foo))

(defun foo ()
    (rplacd (dynamic flag) 'ment) ;succeeds
    (with-binding (((dynamic flag) '(fer . tile) mutable)) ;succeeds
         (print flag))
    (setf (dynamic flag) '(sherly . not))) ;error
```

4.5 Abbreviations

While the rules we have introduced for binding and reference are quite general, they suffer from verbosity. To alleviate this, we introduce defaults. These defaults must be comprehensible and convenient to the user, as well as unambiguous and efficient to implement.

Our single example of such methods to disambiguate references and bindings are the declarations of Common Lisp. While they are concise, they have two major problems:

1. the indefinite scope of proclaim follows Newton. That is, action at a distance can take place. (By proclaiming an identifier special, the effects on previously defined code is unsure. Uncompiled code is affected in most Common Lisps, while compiled code doesn't see the difference.)

2. the examples in Section 4.1 show that the declarations are asymmetric with respect to dynamic reference and lexical reference.

We propose a mechanism to locally introduce abbreviations for binding mentions (both binding creation and reference.) The special form `with-reference` associates a full binding mention with an unqualified identifier (for a given name-space.)

```
(with-reference
      ((x (global x))
       (y (dynamic x)))
    (with-binding (((dynamic z) 3))
         (setf x 'new-global-val)
         (setf y 'new-dyn-val)
         (setf (dynamic z) (list x y))))
```

Note that `with-reference` provides concise access to a binding. `with-binding` introduces bindings within the name-space specified. But the references to the identifier bound in `with-binding` (`z`) still had to be qualified (dynamic). The natural user level definition of let is the following:

```
(defmacro let (pairs . body)
  ;for unqualified identifiers produces a lexical binding. Qualified
  ;references produce a binding in the specified namespace. In addition,
  ;a default abbreviation for references is in effect for the body.
  `(with-binding
    ,(mapcar
        (lambda (pair)
          `(,(if (symbolp (car pair))
                 `(lexical ,(car pair)) ;lexical binding is default in let.
                 (car pair))    ;but a generalized binding form is permitted.
             ,(cadr pair)))
         pairs)
    (with-reference
      ,(mapcar (lambda (pair)
                 (if (symbolp (car pair))
                     `(,(car pair) (lexical ,(car pair)) mutable)
                     `(,(cadar pair) ,(car pair) mutable)))
               pairs)
      ,@body)))
```

$\equiv$

```
(let ((x 3)
     ((dynamic y) 5))
   (+ x y)))
```

```
(with-binding (((lexical x) 3)
               ((dynamic y) 5))
    (with-reference
        ((x (lexical x) mutable)
         (y (dynamic y) mutable))
        (+ x y)))
```

Thus `with-reference` allows `with-binding` to have unambiguous semantics without enforcing an overly cumbersome syntax on the user. Equally important, the determination of a reference from an identifier (Step 1 in the reference determination model) can be very efficiently implemented. In the case of a raw symbol in the interpreter, the lookup is in a context which is maintained parallel to the lexical variable environment. Of course, this implies that Step 1 is accomplished statically in the compiler.

While writing a meta-circular interpreter for a Lisp incorporating the reference framework suggested here, two alternative semantics for `with-reference` were considered. Both semantics agree for all the cases we have presented in this paper. However, consider the following:

```
(let (((dynamic x) 'outer-dynamic-val))
   (with-reference ((z (dynamic x)))
       (let ((dynamic x) 'inner-dynamic-val)
       z))) ;⇒ outer-dynamic-val or inner-dynamic-val?
```

The two possibilities arise from two different readings of `with-reference`.

1. The abbreviations allowed by `with-reference` act like macros. That is the abbreviation (`z` in the example) can be viewed as being replaced textually by its full reference `(dynamic x)` in the example.

2. With-reference dereferences the references to associate a location with the abbreviation. This dereferencing happens on entry to the `with-reference`.

Reading 1 has the advantage of very clear readability. With-reference is viewed as (context relative) textual replacement.

Reading 2 has the advantage of providing access to bindings other than the most recent. While this reading provides more general access to bindings, it is difficult to judge the usefulness since it has not generally been possible.

We are currently studying the relative advantages of the two readings. Their merits will be further developed in the final paper.

4.6 $\mathrm{Lisp}_1/\mathrm{Lisp}_2$

We have discussed reference in the context of Lisp_1, however, the choice of $\mathrm{Lisp}_1/\mathrm{Lisp}_2$ is orthogonal. In the case of Lisp_2, we have the the additional requirements of a lexical function reference. So, we extend the model with the access forms `lexical-function` and (global) `function`.

```
(let ((b 3)
      (list 4))
    (list b)  ;≡ ((function list) (lexical b))
    (b list)) ;≡ ((function b) (lexical list))
```

Of course, the interpreter and compiler use the same mechanism explained for variable access. In Lisp_2, `funcall` and the "car of form is symbol" case in `eval` access the function reference context established by `with-reference`[2]. Syntactically, this allows Lisp_2 code to be written as is traditional. In implementation, this introduces no overhead in the interpreter because Step 1 and Step 2 are chained, and are already performed in current Lisp_2 interpreters. For the compiler, as in the variable case, Step 1 is always accomplished statically. COMMON LISPs (lexical) `flet` is a macro which expands into `with-binding` and `with-reference` in a manner analogous to the `let` definition given above[3].

4.7 Relationship between Global and Other Environments

To be developed in the final paper. (The point is that when the global environment is viewed as the root of the dynamic and/or lexical environment (as in COMMON LISP) certain distinctions are blurred. This blurring has advantages (global reference doesn't require its own special form) and disadvantages (see section on global reference).)

[2] The accessor (`lookup-reference`) for the context established by `with-reference` enters here. As with the rest of the binding/reference scheme, its denotational semantics will be specified in the final paper.

[3] Access can be provided to the global, lexical and dynamic environments. Note that a dynamic function reference space has usually been forbidden because it would burden compilation (even when it is not used, one cannot efficiently compile function calls because the call may occur inside the scope of a dynamic redefinition.) However, in the reference scheme we suggest, the dynamic function reference is unambiguous, and thus can be allowed.

```
(let (((dynamic-function car) (lambda (model) (list 'ford model))))
  ((dynamic-function car) (car 'T)))
```

In the above example, the "dynamic `flet`" does not impede the compilation of the call to the primitive `car`. Nor does it have any effect on calls compiled in the dynamic scope of the binding.

5 Related Work

In [14] and [16], lexical variable binding and unified function value cell are introduced for Lisp-like languages. This was in part an effort to incorporate desirable aspects from other programming languages into Lisp. T [13] continued this tradition with the introduction of immutable binding, global reference and "multiple toplevel" behavior (see section 4.3). Common Lisp[15] incorporates the constant aspect, as well as lexical/dynamic variable binding together with local/global disambiguation rules. Our conception of binding attempts to include these features with non-ambiguity and modularity as our primary concerns. A result of this is the purely lexical effect of all disambiguation rules in our proposal.

In [4], Gabriel and Pitman discuss many aspects of the $\text{Lisp}_1/\text{Lisp}_2$ choice. The decision remains an interesting one, but we view it as orthogonal to the manner of establishing bindings and specifying references.

6 Conclusions

This paper has presented a semantics of identifier use comprising three steps: determining the reference (name space plus identifier) from the mention, determining the location from the reference, and finally inferring the legal operations on this location. Given examples of desirable properties from existing languages, a binding/reference mechanism was presented which incorporates these properties without sacrificing precision. In addition efficient local disambiguation rules allow concise expression of generalized binding and reference. Finally, the issues surrounding the $\text{Lisp}_1/\text{Lisp}_2$ question are shown to be orthogonal.

Acknowledgements

The authors wish to thank Jérôme Chailloux, Pierre Cointe, Eugen Neidl and other members of the EuLispcommittee for many helpful discussions of this and related topics.

References

[1] Jérôme Chailloux, Matthieu Devin, Jean-Marie Hullot, Le-Lisp*: A Portable and Efficient Lisp System*, 1984 ACM Symposium on Lisp and Functional Progamming, Austin, Texas.

[2] William Clinger, *The Scheme Environment: Dynamic Variables* Lisp Pointers, Vol. 1, #3

[3] Kent Dybvig, *The Scheme Programming Language*, Prentice-Hall, 1987.

[4] Richard P. Gabriel, Kent M. Pitman, *Issues of Separation in Function and Value Cells*, X3J13 document 86/010, December 1987.

[5] Leonard Gilman, Allen J. Rose, *APL, An Interactive Approach*, John Wiley and sons, 1976.

[6] Katherine Jensen, Niklaus Wirth *The Pascal Manual and Report*, Springer-Verlag 1976.

[7] John McCarthy *et al*, *Lisp 1.5 Programmers Manual*, MIT Press, 1962

[8] David A. Moon, *The MacLisp Reference Manual* MIT Project MAC, April 1974.

[9] Julian Padget *et al*, *Desiderata for the Standardisation of Lisp*, 1986 ACM Conference on Lisp and Functional Programming, Cambridge, MA.

[10] Kent M. Pitman, *The Revised MacLISP Manual*, MIT/LCR/TR 295, MIT LCS, May 1983.

[11] Jonathan Rees, William Clinger it eds., *The* R^3 *Report on Scheme*, SIGPLAN Notices 21, 12, Dec 86.

[12] Hanan Sammet, *Deep and Shallow Binding: The Assigment Operation*, Computer Languages, Vol. 4, pp 1878–198, 1979.

[13] Stephen Slade, *The T Programming Language, a Dialect of Lisp*, Prentice-Hall 1987.

[14] Guy L. Steele, Jr., Gerald J. Sussman, *The Art of the Interpreter or, The Modularity Complex (parts zero, one and two)*, MIT AI Lab Memo 453, May 1978.

[15] Guy L. Steele, Jr., *Common Lisp the Language*, Digital Press, 1984.

[16] Gerald J. Sussman, Guy L. Steele, Jr., *Scheme: An Interpreter for Extended Lambda Calculus*, AI Memo 349, MIT AI Lab, Decembre 1975.

[17] Warren Teitelman, *The Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

[18] Texas Instruments, *PC Scheme*