SEMANTICS FOR LISP

WITHOUT REFERENCE TO AN INTERPRETER

W. M. Lippe F. Simon

Abstract: The goal of the paper is to show that copy-rule semantics can handle higher order functionals in the sence of FUNARGs in LISP. A language LISP/N, which is derived from pure LISP, is introduced and the semantics are defined without reference to an interpreter; its definition is essentially based on a copy-rule for LISP/N.

Institut für Informatik und Praktische Mathematik
Universität Kiel
Olshausenstraße 40 - 60
D-2300 Kiel 1

I. INTRODUCTION

The definition of the programming language LISP by an interpreter [Mc p. 13 and p. 70/71] has several deficiencies. Both interpreters are written in a subset of LISP, wherefore several authors [Weg, Sto, Gor] doubt whether this metacircular definition is a true definition for LISP. McCarthy [Sto] admits that the understanding of the interpreters is based on intuition and experience.

The semantics definition for LISP is purely operational and implies several major differences to the semantics for ALGOL-like programming languages, where it is well known how to establish a formal semantics definition e. g. [Ho, La]. M. J. Gordon gives a formalization of the operational semantics for LISP and proves it to be equivalent to a non-standard denotational semantics definition [Go]. LISP has been derived from λ-calculus but it cannot be considered merely as a machine implementation of λ-calculus, nor as an interpretation of λ-calculus schemata. On the other hand it is possible to construct a well defined LISP-interpreter by means of software engineering, starting from α- and β-reductions in λ-calculus [Per, Gr, Ro]. Our goal is to introduce copy-rule semantics for LISP-like languages.

Our work started from an observation by M. J. Fischer [Fi] concerning the run time storage management for LISP: In principle only a stack, well known from ALGOL-like languages, is necessary (deletion strategy); there is no need for a heap (retention strategy). In [Si, Si/Tr, Si I] it has been shown that LISP is essentially an ALGOL-like programming language. In this paper we are going to define an ALGOL-like syntax and semantics for LISP that reflect the important features of LISP, whereas some of the "exotic" ones are dropped. We call this language LISP/N; the suffix /N has been chosen to remind of "call by name". The semantics of LISP/N are defined without reference to an interpreter, its definition is essentially based on a copy-rule for LISP/N. This copy-rule is slightly more sophisticated than the well known ALGOL 60-copy-rule in order to handle unrestricted higher order functionals (FUNARGs), in particular procedures occurring as values of (function) procedures.

At last we want to make some remarks on the design of LISP/N. We refer to the second of the two interpreters given by McCarthy [Mc], the one used for practical purposes in LISP-systems, but we drop all features belonging to LISP 1.5 and not to pure LISP, e. g. PROG,

FEXPR, APVAL etc. [Si I]. This interpreter properly handles upward and downward FUNARGs, i. e. procedures passed from or to procedures, and it is more rigid concerning correct parameter transmissions. With our definition of LISP/N we are only going to model pure LISP, but it should not be too difficult to extend LISP/N such that important members of the LISP-familiy like LISP 1.5, LISP 1.6 or INTERLISP are covered.

The language MLISP [Smi] is one of the early attempts to use an ALGOL-like notation for LISP-programs. In modern LISP-systems like VLISP and INTERLISP, there is a strong tendency to weaken the language constructs of pure LISP in such a way that a programmer is able to write his programs in an ALGOL-like style. A careful analysis of different versions of LISP-interpreters by Steele and Sussman [St/Su] shows that the ALGOL binding mechanism for variables (static scoping) is the suitable one for LISP; the role of dynamic scoping [Mc p. 13] is restricted to express certain structured forms of side effects.

Like all ALGOL-like languages LISP/N has a compiler which translates LISP/N-programs into executable machine code for a real computer (or an abstract machine). In larger LISP-systems there usually exist compilers, but these ones are only able to compile certain "modular" parts of LISP-programs into machine code; for difficult situations they rely on run time support by the LISP-interpreter, c. f. LISP 1.6, INTERLISP.

If we consider LISP as an ALGOL-like language, we also have to accept some important differences between LISP/N and pure LISP. In pure LISP free variables may occur and they are bound to values by dynamic scoping. In LISP/N there are so called "global formal" parameters i. e. a formal procedure parameter that is global with respect to a given procedure. These parameters, only possible within procedures with nestings ≥ 2, are bound to values by static scoping and they allow a simulation of most free variables occurring in practical programs.

There are no mode specifications required in pure LISP, so several dangerous errors cannot be recovered by the interpreters given in [Mc]. In LISP/N completely specified modes, finite or infinite, are defined by mode equations put in front of the program, but they are so simple that we don't run into problems as in ALGOL 68. As a consequence of the static mode checks in LISP/N-programs there is

no wrong procedure call occuring at run time. So we don't need parameter checks at run time.

It is known that "call by value" as computation rule does not always coincide with "call by name" c. f. ALGOL 60. Since we wanted to introduce copy-rule semantics in LISP, we decided to have "call by name". In the particular situation of LISP, where we don't have assignment statements, the differences that may be introduced by "call by value" are rather irrelevant for practical programming. In [Fr/Wi] it is shown how multiple computations of the same "call by name"-parameter can be avoided using "call by need".

Parameter transmissions with "call by name" are the main reason why in LISP/N the run time storage for procedure activation records may be organized as a stack. In principle only this stack is necessary as run time store [Si]. The heap, keeping data (s-expressions), is a concession to achieve a compact representation of s-expressions, since nobody would accept lots of procedure calls as encodings of s-expressions.

II. LISP/N

We only have s-expressions as values. An atomic s-expression is a finite sequence of capitals and digits, beginning with a capital. If s_1 and s_2 are s-expressions then $s_3 = (s_1.s_2)$ is a s-expression as well. Furthermore we have the following abbreviations called list notations:

$$(s_1.(s_2 \ldots \ldots (s_n.\mathrm{NIL})\ldots)) = (s_1\ s_2 \ldots s_n), \quad n \geq 1$$

NIL = ( ) is a special s-expression corresponding to the empty list.

In LISP/N we have five standard-procedures:

$$car(s) =_{Df} \begin{cases} s_1 & \text{if } s = (s_1.s_2) \\ \text{undef.} & \text{else} \end{cases}$$

$$cdr(s) =_{Df} \begin{cases} s_2 & \text{if } s = (s_1.s_2) \\ \text{undef.} & \text{else} \end{cases}$$

$$cons(s_1,s_2) =_{Df} (s_1.s_2)$$

$$atom(s) =_{Df} \begin{cases} T & \text{if } s \text{ is an atomic s-expression} \\ F & \text{else} \end{cases}$$

$$eq(s_1,s_2) =_{Df} \begin{cases} T & \text{if } s_1,s_2 \text{ are atomic s-expressions and } s_1 = s_2 \\ F & \text{if } s_1,s_2 \text{ are atomic s-expressions and } s_1 \neq s_2 \\ \text{undef.} & \text{else} \end{cases}$$

T and F are special s-expressions corresponding to the Boolean values true and false.

The set of syntactical LISP/N-programs is generated by a context-free grammar*)[Li/Si]. Informally a LISP/N-program is devided into 3 parts: the mode declaration part, the procedure declaration part and the main program. The main program consists of exactly one expression which defines the result of the program - a s-expression. The body of a procedure consists of a procedure declaration part (nested declarations) and a single expression. The mode declaration part is a system of equations which defines the modes of all procedures declared in the program. It is required that every procedure has a complete mode declaration. We don't have the mode 'formal'.

Starting with s-expressions as values and the five standard-procedures as operators expressions are defined similar to arithmetic expressions in ALGOL 60. According to the definition of LISP we have to introduce a second type of expressions which ranges over procedures (FUNARGs).

---

*) The productions are given in appendix A.

Example 1:

```
begin  mode mf = proc (mr) mr;
       mode mr = proc (s-expr) s-expr;
       mf : twice (mr:f) mr; {mr : p (s-expr:x) s-expr;
                                                {f(f(x))};
                                  p};
       twice (cdr)((A B C))
end
```

The result of this program is computed by the following sequence of procedure calls: twice (cdr)((A B C))→p((A B C))→cdr(cdr(A B C))→(C).

Example 2:

```
begin  mode m = proc (s-expr,s-expr) s-expr;
       m : reverse (s-expr:x, s-expr:y) s-expr;
        {if atom(x) then y else reverse(cdr(x),cons(car(x),y)) fi};
       reverse((A B C D),NIL)
end
```

This program is the well known reverse-function for LISP-lists.

Similar to ALGOL [La] we can establish a binding relation δ between applied and defining occurrences of identifiers in a syntactical program Π. A syntactical program is called closed, if the relation δ is a function, totally defined on the set of all occurrences of identifiers in Π. The property to be a closed program is decidable and δ is a computable function.

We want to define, when a closed program is called a compilable program. Informally this means that any applied occurrence of an identifier satisfies a static type checking procedure with respect to the mode declaration part of the programm. The complete definition is given in [Li/Si].

III. THE COPYRULE

The basic idea behind copy-rule semantics consists of two steps. First we assume that for every LISP/N program π without any procedure calls (except car, cdr, ..., eq) we can construct an input/output function $f_\pi | E^n \to E$, where n is the number of inputs to π and E is the set of all s-expressions. This may be achieved e. g. by denotational, algebraic or axiomatic methods. In the second step we consider LISP/N-programs in general, i. e. with procedure calls. The construction of input/output functions for such programs is essentially based on a copy-rule for LISP/N and the previous step. In the most simple case the iterated application of the copy-rule to a program π with procedure calls generates a finite set $\{\pi_1, \pi_2, \ldots, \pi_k\}$ of programs without any procedure calls and the input/output function of π is $f_\pi \equiv \bigcup_{j=1}^{k} f_{\pi_j}$.

The main part of a program π is that part of the <main program> of π which is outside of all procedures declared within this <main program>.

If we have a conditional statement then we distinguish between the if-part, then-part and else-part in the following manner:

$$\underbrace{\text{if } \ldots.}_{\text{if-part}}\ \underbrace{\text{then } \ldots.}_{\text{then-part}}\ \underbrace{\text{else } \ldots. \text{ fi.}}_{\text{else-part}}$$

Let $f(a_1^0,\ldots,a_{n_0}^0) \ldots (a_1^r,\ldots,a_{n_r}^r)$ be a procedure statement. If $r = 0$ then we have a procedure statement with an eventually empty parameter list similar to ALGOL 60. If $r > 0$ then f is a higher order functional. We say that a procedure statement resp. a procedure expression α occurs on an actual parameter position in a call of f, if α appears in one of the actual parameters a_i^j, $0 \leq j \leq r$, $1 \leq i \leq n_j$.

We are now ready to define a copy-rule for LISP/N in a similar way as for ALGOL 60. If the expression being the main program of a given program π is a procedure call, then the copy-rule defines how to replace this call by a modified body of the called procedure. So by application of the copy-rule to π we get a program π'. Notice that new procedure declarations may occur in the main part of π'. The main difference between the ALGOL 60-and the LISP/N-copy-rule comes from

the extended procedure concept in LISP/N where we have procedures occuring as results of (function)-procedures.

Definition 1:

Let π be a compilable program. A program π' is called to result from π by application of the copy-rule ($\pi \longmapsto \pi'$) if the following holds:

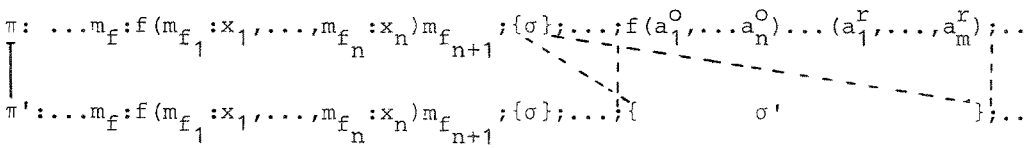
Let $f\ (a_1^o,\ldots,a_n^o)\ \ldots\ (a_1^r,\ldots,a_m^r)$ be a procedure statement in the main part of π, not occurring on an actual parameter position in a call of a non-standard procedure.

Let $m_f : f\ (m_{f_1}:x_1,\ldots,m_{f_n}:x_n)\ m_{f_{n+1}}\ ;\ \{\sigma\}$

be the associated procedure declaration. Then

$f\ (a_1^o,\ldots,a_n^o)\ \ldots\ (a_1^r,\ldots,a_m^r)$

is replaced by a generated block $\{\sigma'\}$ where σ' is the modified body σ:



The modifications of σ are:

i) (Substitution)
The formal parameters x_i occurring in σ are replaced by the corresponding actual parameters a_i^o.

ii) (Propagation of parameter lists)
If $r > 0$ then all procedure identifiers p and all procedure statements $p'(b_1^o,\ldots,b_{n'}^o)\ldots(b_1^t,\ldots,b_{m'}^t)$ in the <expression> of the procedure body $\{\sigma\}$, except those which occur on actual parameter positions and those which occur in if-parts, are replaced by $p(a_1^1,\ldots,a_l^1)\ldots(a_1^r,\ldots a_m^r)$
resp. $p'(b_1^o,\ldots,b_{n'}^o)\ldots(b_1^t,\ldots,b_{m'}^t)(a_1^1,\ldots,a_l^1)\ldots(a_1^r,\ldots,a_m^r)$.

iii) (Renaming)
All identifiers which are local to σ' are admissible renamed by identifiers which do not yet occur in σ'.

Let $\overset{+}{\longmapsto}$ resp. $\overset{*}{\longmapsto}$ be the transitive and transitive-reflexive closures of $\longmapsto$.

Remark:

The copy-rule allows to expand a procedure call $f(a_1,\ldots,a_n)$ in the following contexts:

1. ...; $f(a_1,\ldots,a_n)$ ... or
2. ...; $car(f(a_1,\ldots,a_n))$ ... or
3. ...; $cdr(f(a_1,\ldots,a_n))$ ... or
4. ...; $eq(x, car(f(a_1,\ldots,a_n)))$ ...

whereas the copy-rule may not be applied in these cases:

a) ...; $g(x_1,\ldots,f(a_1,\ldots,a_n),\ldots,x_m)$ ... or
b) ...; $cdr(g(x_1,\ldots,f(a_1,\ldots,a_n),\ldots,x_m))$ ... or
c) ...; $g(x_1,\ldots,cdr(f(a_1,\ldots,a_n)),\ldots,x_m)$ ...

Example 3:

To illustrate the copy-rule we consider example 1. Notice that the propagation of parameter list ((A B C)) by modification ii) causes a call of the procedure p' in the modified body of the procedure twice

```
begin ... mf : twice(mr:f)mr;{mr:p(s-expr:x)s-expr;{f(f(x))};p};
twice (cdr)((A B C))  end
... {mr:p'(s-expr:x')s-expr;{cdr(cdr(x'))}; p'((A B C))}  ...
... {cdr(cdr((A B C)))}
```

A program π is called original if it is not derived by application of the copy-rule.

Definition 2:

Let π be an original program, then $E_\pi =_{Df} \{\pi' \;/\; \pi \longmapsto \pi'\}$ is called the execution of π. The set

$$T_\pi =_{Df} \{\pi' \;/\; \pi' \varepsilon E_\pi \text{ and } \pi' \text{ has at most one innermost generated block}\}$$

is called the execution tree of π.

Example 4:

In this program it is shown how the copy-rule handles nested procedure declarations (p2), procedure expressions (p2) and procedure calls with "fat" actual parameters (p1).

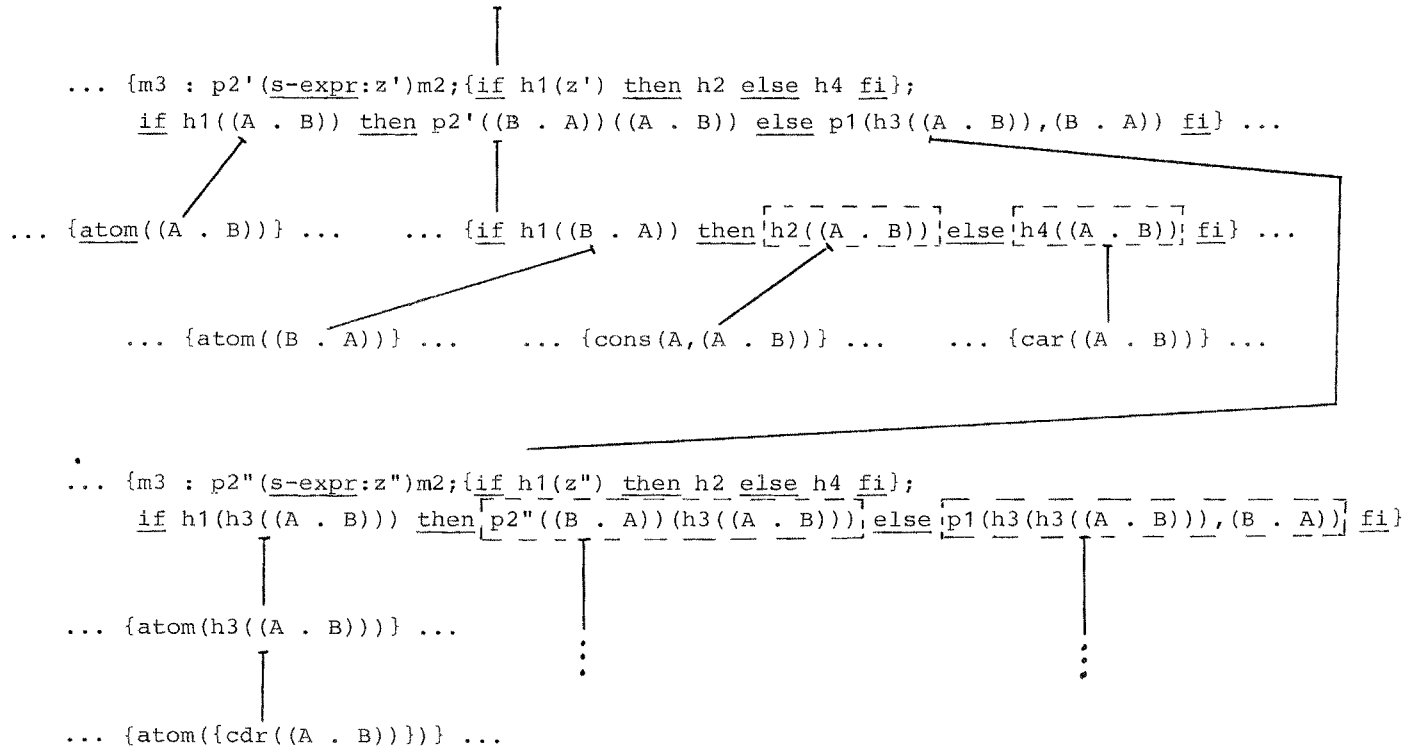

... {m3 : p2'(s-expr:z')m2;{if h1(z') then h2 else h4 fi};
if h1((A . B)) then p2'((B . A))((A . B)) else p1(h3((A . B)),(B . A)) fi} ...
... {atom((A . B))} ...
... {if h1((B . A)) then h2((A . B)) else h4((A . B)) fi} ...
... {atom((B . A))} ...
... {cons(A,(A . B))} ...
... {car((A . B))} ...
... {m3 : p2"(s-expr:z")m2;{if h1(z") then h2 else h4 fi};
if h1(h3((A . B))) then p2"((B . A))(h3((A . B))) else p1(h3(h3((A . B))),(B . A)) fi}
... {atom(h3((A . B)))} ...
... {atom({cdr((A . B))})} ...

fig. 1

```
begin  mode m1 = proc (s-expr,s-expr) s-expr;
       mode m2 = proc (s-expr) s-expr;
       mode m3 = proc (s-expr) m2;
       m1 : p1 (s-expr:x, s-expr:y) s-expr;
         {m3 : p2 (s-expr:z) m2; {if h1(z) then h2 else h4 fi};
          if h1(x) then p2(y)(x) else p1(h3(x),y) fi};
       m2 : h1 (s-expr:x1) s-expr; {atom(x1)};
       m2 : h2 (s-expr:x2) s-expr; {cons(A,x2)};
       m2 : h3 (s-expr:x3) s-expr; {cdr(x3)};
       m2 : h4 (s-expr:x4) s-expr; {car(x4)};
       p1 ((A . B),(B . A))
end
```

An initial segment of the infinite execution tree is given in fig. 1. Modifications according to ii) of the copy-rule are marked.

IV. SEMANTICS DEFINITION

We are going to define the semantics of a program π with procedure calls by programs without procedure calls. These programs are essentially those of E_π resp. T_π. The construction of an input/output function for π depends on the following facts:

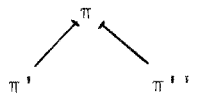
Theorem

Let π be a compilable LISP/N-program. Then $(E_\pi, \overset{*}{\longrightarrow})$ is a lattice.

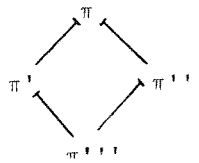
The proof [Li/Si] is based on these lemmata:

Lemma 1:

If the diagramm

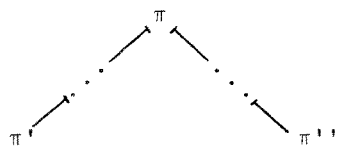


holds for the programs π, π', π'' then either π' and π'' are identical or there is a unique program π''' with

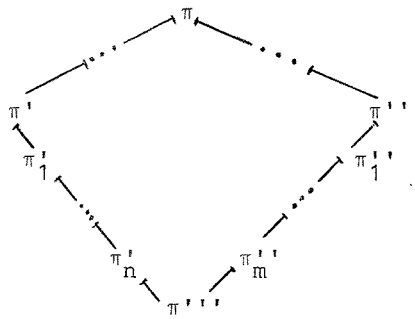


Lemma 2:

If the diagramm



holds then either π' and π'' are identical or there exist programs $\pi'_1,\ldots,\pi'_n$, $\pi''_1,\ldots,\pi''_m$, π''' with



To define the semantics of a LISP/N-program we consider programs without procedures first. A compilable LISP/N-program $\tilde{\pi}$ without procedures can be understood as a function

$$f_{\tilde{\pi}} \mid E^n \longrightarrow E$$

where n is the number of inputs to $\tilde{\pi}$ and E denotes the set of all s-expressions. In general $f_{\tilde{\pi}}$ will be only partially defined, because $\tilde{\pi}$ may run into an infinite loop or terminate irregular for certain input-data.

Now we consider a compilable LISP/N-program π with the execution E_π. From $\pi' \varepsilon E_\pi$ we get the program $\tilde{\pi}'$ without procedures if we eliminate all procedure declarations and if we replace every remaining procedure statement by car(A), a syntactical representative for the undefined value (i. e. Ω). Notice that there are no procedure expressions in $\tilde{\pi}'$. Let $\pi' \longmapsto \pi''$, then we have functions $f_{\tilde{\pi}'}, f_{\tilde{\pi}''} \varepsilon [E^n \longrightarrow E]$ with $f_{\tilde{\pi}'} \sqsubseteq f_{\tilde{\pi}''}$. By Theorem 2. we have seen that $(E_\pi, \overset{*}{\longmapsto})$ is a lattice. Therefore we can define the input/output function $f_\pi \varepsilon [E^n \longrightarrow E]$ of the original program π by the union

$$f_\pi =_{Df} \bigsqcup_{\pi' \varepsilon E_\pi} f_{\tilde{\pi}'}$$

So we have defined the semantics of the original compilable program π by the function f_π. On the other hand we can take the execution tree T_π instead of E_π for the construction of f_π, because E_π is a distributive lattice isomorphic to the lattice $\mathfrak{T}_\pi$ of all finite subtrees of T_π [La].

We have shown that copy-rule semantics can be defined for languages with higher order functionals.

References

[Be] Berry,D.M.: Block Structure: Retention or Deletion. Proc. of the Third Annual ACM Symp. on Theory of Computing, 1971

[Bo/We] Bobrow,D.G.,Wegbreit,B.: A Model and Stack Implementation of Multiple Environments. CACM 4, 10, 1973

[Fi] Fischer,M.J.: Lambda Calculus Schemata. SIGPLAN Notices 7 (1), 1972

[Fr/Wi] Friedman,D.P.,Wise,D.S.: CONS should not Evaluate its Arguments. Third ICALP, Edinburgh, 1976

[Gor] Gordon,M.J.C.: Models of Pure LISP (a worked example in semantics). Dept. of Machine Intelligence, Univ. of Edinburgh, Rep. 31, 1973

[Gr] Greussay,P.: Contribution à la définition interprétative et à l'implementation des λ-langages. Thèse ès Sciences, Paris VII, 1977.

[Hes] Hesse,W.: Two-level Graph Grammars. Technische Universität München, TUM-INFO-7833, 1978

[Ho] Hoare,C.A.R.: An axiomatic basis for computer programming. CACM 12, 1969

[Ka/Li] Kaufholz,G.,Lippe,W.M.: A Note on a Paper of H. Langmaack about Correct Procedure Parameter Transmission. Universität des Saarlandes, Bericht Nr. A 74/06, 1974

[La] Langmaack,H.: On Correct Procedure Parameter Transmission in Higher Programming Languages. Acta Informatica 2, 1973

[Li] Lippe,W.M.: Über die Entscheidbarkeit der formalen Erreichbarkeit von Prozeduren bei monadischen Programmen.

[Li/Si] Lippe,W.M.,Simon,F.: LISP/N - Basic Definitions and Properties Bericht Nr. 4/79 des Instituts für Informatik und Praktische Mathematik der Universität Kiel

[Mc] McCarthy,J.et.al.: LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1966

[Mc I] McCarthy,J.: A New EVAL Function. MIT Artificial Intelligence Memo No. 34

[McG] McGowan,C.L.: The "most-recent" Error: its Causes and Correction. Proc. of an ACM Conf. on Proving Assertions about programs. SIGPLAN Notices Vol. 7. No. 1, 1972

[Mo] Moses,J.: The Function of FUNCTION in LISP. SIGSAM Bull. 15, 1970

[Per] Perrot,J.F.: LISP et λ-calcul. in: B.Robinet (Ed.) Lambda Calcul et Sémantique Formelle des Langages de Programmation, Paris, 1979

[Ro] Robinet,B.: Petit precis de lambda-calcul. Ecole de l'IRIA: Implementation et Interprêtation des LISP, Toulouse, 1978

[Sa] Sandewall.E.: A Proposed Solution to the FUNARG Problem. Uppsala Univ. Dept. of Comp. Sci., Rep. 28, 1970

[Schw] Schwartz,R.L.: A W-grammar Description of LISP. Modeling and Measurement Note No. 34, Comp. Dept. School of Eng. and Appl. Sci., UCLA, 1975

[Si I] Simon,F.: Zur Charakterisierung von LISP als ALGOL-ähnliche Programmiersprache mit einem strikt nach dem Kellerprinzip arbeitenden Laufzeitsystem. Dissertation, Kiel, 1978

[Si] Simon,F.: Cons-freies Programmieren in LISP unter deletion-Strategie. in: Informatik Fachberichte, Bd. 1, Springer Verlag, 1976

[Si/Tr] Simon,F.,Trademann,P.: Eine Beziehung zwischen consfreiem LISP und Stackautomaten. Journal of Information Processing and Cybernetics (EIK), Vol.14, No. 12, 1978

[Smi] Smith,D.C.: MLISP. Stanford Artificial Intelligence Project Memo AIM-135, Computer Science Department Report No. CS 179, Oktober 1970

[Ste] Steele,G.J. Jr.: Macaroni is Better than Spaghetti. Proc. of the Symp. on AI and Progr. Lang., ACM, 1977

[St/Su] Steele,G.J.`Jr.,Sussman,G.J.: The Art of the Interpreter or the Modularity Complex. AI Memo No. 453, MIT AI-Lab., 1978

[Sto] Stoy,J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press Series in Computer Science, Cambridge, Mass., 1977

[Weg] Wegner,P.: Three Computer Cultures.

Appendix A: Syntax of LISP/N-programs

<program>::=begin<mode decl. part><proc. decl. part><main program>end

<mode decl. part>::=<empty>/<mode decl.>;[<mode decl.>;]

<mode decl.>::=mode<midf.>=<struct. mode>

<midf.>::=<identifier>

<struct. mode>::=proc ( )<result mode>/proc(<mode>[,<mode>])<result mode>

<result mode>::=<midf.>/void/s-expr

<mode>::=<midf.>/s-expr

<proc. decl. part>::=<empty>/<proc. decl.>;[<proc. decl.>;]

<proc. decl.>::=<midf.>:<identifier>(<form. par. list>)
<result mode>;<proc. body>

<form. par. list>::=<empty>/<mode>:<identifier>[,<mode>:<identifier>]

<proc. body>::={<proc. decl. part><expression>}

<expression>::=<empty>/<Bool. expr.>/<s-expr. expr.>/<proc. expr.>/
<proc. st.>

<Bool. expr.>::=T/F/<identifier>/<proc. st.>/ atom(<s-expr. expr.>)/
eq(<s-expr. expr.>,<s-expr. expr.>)/
if<Bool. expr.>then<Bool. expr.>else<Bool. expr.>fi

<s-expr. expr.>::=<s-expr.> /<identifier>/<proc. st.>/
cons(<s-expr. expr.>,<s-expr. expr.>)/
cdr(<s-expr. expr.>)/car(<s-expr. expr.>)/
if<Bool. expr.>then<s-expr. expr.>else<s-expr.expr.>fi

<proc. expr.>::=<identifier>/
if<Bool. expr.>then<proc. expr.>else<proc. expr.>fi

<proc. st.>::=<identifier>(<act. par. list>)[(<act par. list>)]/
if<Bool. expr.>then<proc. st.>else<proc. st.>fi

<act. par. list>::=<empty>/<act. par>[,<act. par.>]

<act. par.>::=<Bool. expr.>/<s-expr. expr.>/<proc. expr.>/<proc. st>

<main program>::=<empty>/<s-expr. expr.>/<Bool. expr.>/
<proc. st.>/<gen. expr.>

<gen. block>::={<gen. block>}/<proc. body>

<gen. expr.>::=<gen. block>/
atom(<gen. expr.>)/
cdr(<gen. expr.>)/
car(<gen. expr.>)/
eq(<gen. expr.>,<gen. expr.>)/
eq(<gen. expr.>,<expression>)/
eq(<expression>,<gen. expr.>)/
cons(<gen. expr.>,<gen. expr.>)/
cons(<gen. expr.>,<expression>)/
cons(<expression>,<gen. expr.>)/
if<gen. expr.>then<gen. expr.>else<gen. expr.>fi/
if<gen. expr.>then<gen. expr.>else<expression>fi/
if<gen. expr.>then<expression>else<gen. expr.>fi/
if<gen. expr.>then<expression>else<expression>fi/
if<expression>then<gen. expr.>else<gen. expr.>fi/
if<expression>then<gen. expr.>else<expression>fi/
if<expression>then<expression>else<gen. expr.>fi

Remark

The bracket symbols [,] are used to denote finite repetitions of the enclosed symbols resp. their omission.