# Proof by semantic attributes of a LISP compiler

P. Deransart

*IRIA, Domaine de Voluceau, BP 5—Rocquencourt, 78150 Le Chesnay, France*

**This paper illustrates the possibility to prove correctness of a compiler defined by semantic attributes. Source language and object code are first described. The source language corresponds to LISP compilable functions and the object code to an abstract machine. Then the translation between them is defined and the proof technique is described and illustrated by some examples. The interest of this technique is the modularity of the proof.**

**Cet article montre comment il est possible de prouver la correction d'un compilateur décrit par attributs sémantiques. Le langage source et le code objet sont introduits en premier lieu. Le premier correspond aux fonctions LISP compilables et le second à une machine abstraite. La traduction de l'un dans l'autre est ensuite définie et la méthode de preuve est exposée. L'intérêt de cette méthode repose sur l'extrême modularité de ce type de preuve.**

(Received September 1977)

Compiler description by the method of semantic attributes is a well known technique and is now used frequently. Not so familiar are the possibilities to use this technique for proving a compiler correct. This involves using the syntax-oriented description to prove recursively correctness. We show here the possibility to apply such a proof method, to a LISP compiler.

We were led to consider LISP because of its simplicity and the possibility to describe the LISP semantics with an 'apply' function. The compiler is correct if, given any sufficient arguments and any environment, the execution of the generated code of the compiled function is equivalent to the function 'applied' to the same arguments, with the same environment. The LISP used here is a subset of a compilable LISP, defined in order to study the definition of LISP by a compiler and its portability. We shall present first the source language and its semantics defined by the 'eval' and 'apply' functions, and then the object code for the target abstract machine (i.e. an intermediate code suitable for implementation on an IBM machine (Lux, 1975) as well as a DEC machine (London, 1971)). Finally, we prove the translation, verifying at the same time the correctness of the given semantic attribute description. A certain knowledge of LISP and semantic attributes is assumed.

## 1. Source language: syntax and evaluator

A context-free subset of the source language syntax is given in **Appendix 1.1,** described in a BNF-like notation, where the terms LAMBDA, QUOTE, COND, T and NIL are considered as reserved words. With the exception of the LABEL function, it includes the usual LISP expressions.

The evaluator defines the syntax of evaluable expressions implicitly by means of tests. It operates like the pure LISP evaluator. It takes two arguments: an expression and the alist. The alist argument is formed, as shown by the function 'pairlis', by a left-branching list of dotted pairs, including a variable and the associated value. The alist corresponds to the evaluation context. It will be presumed that the evaluated expressions have no free variables, implying that each variable is defined at least once in the alist. It is easy to understand the evaluation process, reading the function 'eval' (**Appendix 3**) which evaluates each argument of a function with the same context. One will note the principal difference from pure LISP: if a function is an identifier, it is only a function name and not a variable; in particular, there are no functions as arguments of LAMBDA expressions.

The relationship between syntax and evaluation are studied in Deransart (1977). We consider now here that the evaluator evaluates only well formed expressions, as defined by the given syntax.

## 2. Object language and implementation

The abstract target machine (see **Appendix 2.2**) consists of:

1. A memory zone including a potentially infinite number of cells. We don't describe this zone here, assuming that the notation 'S-expr corresponds to the memory address of the S-expression (*'adtc'* in the object code described in Appendix 2).
2. A potentially unlimited atom table, where each identifier is represented only once and where the recursivity of LISP implies that each atom does not point a value, but a stack of values, called a bound-stack. The notation 'Ident denotes the address of the atom identifier.
3. A unbounded stack-zone. Each atom has a bound-stack and there is also an argument-stack and a return-stack with the return addresses of the called subprogram. The top of the argument-stack is denoted *sp* and the content *c*(*sp*).
4. A zone for programs and subprograms.
5. An instruction counter, denoted *co*.

All these elements are organised as shown in **Fig. 1.** There are ten machine instructions described in Appendix 2.2: 4 concern the argument-stack manipulations, 2 the bound-stacks, 2 the return-stack and 2 are branching instructions. The machine sequentially executes the instructions of the program, the counter being initialised to 1 (the first instruction).

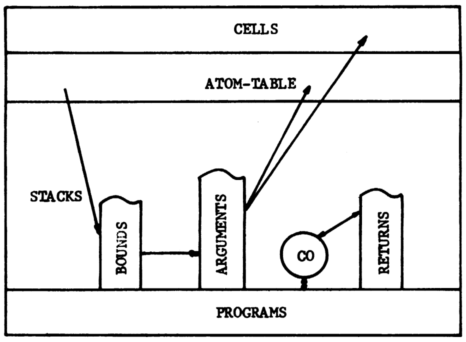

**Fig. 1 Abstract machine**

The evaluator executing an abstract machine program is denoted by EV.

## 3. Description by semantic attributes

Each attribute represents a concept useful for generating the object code. At each node of the syntactic tree where it is defined, the attribute has a calculated value. So the attribute 'CAPPLY' denotes the generated code at the node ⟨Function⟩ (rules 7 and 8 in Appendix 1.3) and denotes the complete generated code at the node ⟨Prog⟩ (rule 0).

We will use five sets of attributes (see Appendix 1.1):

(*a*) *CEVAL*, *CAPPLY*, *CEVCON*, *CEVLIS*: four names for the same concept—the generated code. This denomination makes easier the identification within the evaluator.

(*b*) *PBOUND*, *OBOUND*: for generating the push and pop instructions of the bound-stack, used only in the LAMBDA expressions.

(*c*) *LAB*, *OUTC*, *LAB1*, *LAB2*: These attributes denote the addresses of the generated instructions. LAB denotes the next instruction to be generated (H-LAB(Function) in the rule 0 is initialised to one and S-LAB(Function) in the same rule denotes the address of the last instruction of the program: 'S-LAB(Function) : RETURN'). OUT C(output address of conditional form) is used to generate the code for a conditional branch, and LAB1 and LAB2 to build the generated code of PBOUND and OBOUND.

(*d*) *NARG*, *NVAR*, *TOP*, *ALIST*: The first two attributes count respectively the number of arguments of a function and the number of variables to be bound to the arguments values. In the sequel, we will assume that the following condition is satisfied:

$$\text{H-NARG(Function)} \geqslant \text{S-NVAR(listvar)}$$

in each occurrence of rule 8, since the result is not defined otherwise (more variables than arguments).

TOP simulates the evolution of the top of the argument-stack, ALIST the alist.

(*e*) *TEXT* denotes the original source language text corresponding to the syntactic node (terminal or not).

The translation using semantic attributes and syntax of the source language is described in Appendix 1.3. To understand the translation it is sufficient to read the 'circulation' of the attributes, denoting the dependency of the attributes at each node. This is illustrated in rule 11 by the 'circulation' of the attribute LAB of the node ⟨Listcond⟩ (see **Fig. 2**).

The dependency of the attributes at different nodes is fundamental to understanding the organisation of the proof. In Fig. 2, we see that H-LAB(Form) depends on H-LAB(Listcond), meaning that to calculate H-LAB(Form), it is necessary first to calculate H-LAB(Listcond). This result is inherited. By contrast, the value of S-LAB(Listcond) is synthesised from S-LAB(Listcond'). Furthermore the calculation of the inherited value of H-LAB(Form') depends on the value of the synthesised attribute S-LAB(Form). This is called the 'circulation' of the attributes. The dependencies must not be cyclic or, in other words, the attributes descriptions be coherent. The compiler will be proved correct under the assumption of the coherency of this description.

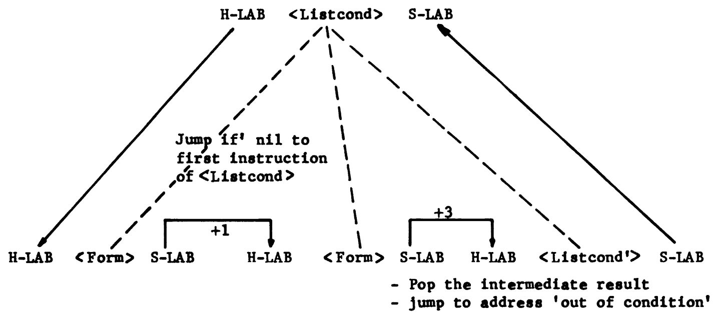
H-LAB
<Listcond>
S-LAB
Jump if' nil to first instruction of <Listcond>
+1
+3
H-LAB
<Form>
S-LAB
H-LAB
<Form>
S-LAB
H-LAB
<Listcond'>
S-LAB
- Pop the intermediate result
- jump to address 'out of condition'

**Fig. 2 'circulation' of the attributes LAB in the rule 11. Indications about the generated code added to the inherited attribute giving the synthesised attribute**

## 4. Presentation of the proof

To prove that the compiler is correct, we have to prove the identity of the results obtained by two methods: first the result of a function applied to arguments localised at the top of the argument-stack with an alist equivalent to the bounds-stacks and corresponding values in the argument-stack. Secondly the results of the execution of the corresponding generated code with the same arguments at the top of the stack and the same context (alist). This is illustrated by the commutativity of the diagram of Fig. 2:

$$\begin{tikzcd}
\text{Function, } (c(sp-n+1) : c(sp)), \text{ alist} \arrow[rr, "\text{apply}"] \arrow[d, "\text{translation}"] & & \text{results} \\
\text{Object code, same arguments, alist } c(sp-n+1) \text{ to } c(sp) \arrow[urr, "\text{EV}"'] & &
\end{tikzcd}$$

The generated code is correct if and only if the abstract machine being in an initial state characterised by:

$$co = 1$$
$$sp = a$$
$$\text{list of } c(sp-n+1) : c(sp) = \text{list of evaluated arguments}$$
$$\text{alist} = \alpha$$

The execution of the code produces a final state such:

$$co = \text{return instruction}$$
$$sp = a + 1 \text{ (add only one argument to the argument-stack)}$$
$$c(sp) = \text{apply (function, list of evaluated arguments, } \alpha)$$

Since the apply function (by definition) does not modify the context, this last identity proves the conservation of the context through evaluation. To express the properties to be verified, we use the Hoare notation (1969) with pre- and post- conditions. Thus, S-CAPPLY representing an object programme, we denote:

$$\text{EV(S-CAPPLY( )) by \{S-CAPPLY( )\}}$$

and the last assertion will be expressed as:

*pre-condition* *post-condition*

$co = 1$ , $sp = a$ , alist $= \alpha$

$la = [c(sp - n + 1) : c(sp)]$

{S-CAPPLY(Prog)}

$co =$ S-LAB(Prog), $sp = a + 1$

$c(sp) =$ apply(Prog, $la$, $\alpha$)

where $n$ is the number of arguments presumed to be at the top of the argument-stack. By convention we will use

$$n = \text{S-NVAR(Prog)} \quad \text{(rule 0, Appendix 1.3)} .$$

In this notation we use the same symbol (Prog) as an argument supplied to apply and as a representation of the non-terminal Prog in the attribute occurrence. There is in fact a strict correspondence between the non-terminals of the grammar and the elements supplied as arguments of functions eval, apply, evlis or evcon (Deransart, 1977). Thus, according to the non-terminal ⟨Form⟩, we have the assertion:

*co* = H-LAB(Form), *sp* = H-TOP(Form)
*sp* = H-TOP(Form) + 1

{S-CEVAL(Form)}
*co* = S-LAB(Form),
*c*(*sp*) = eval(Form, H-ALIST$^c$(Form))*

There are analogous assertions associated with the non-terminals:

⟨Listvar⟩, ⟨Listcond⟩, ⟨Listarg⟩ and ⟨Function⟩

The proof technique studied in Pair, Amirchahy and Néel (1976) consists of associating one of the attributes of the considered non-terminal symbol to the assertion. Here it is the attribute S-CEVAL for the non-terminal symbol ⟨Form⟩, S-CAPPLY for ⟨Function⟩, S-CEVLIS for ⟨Listvarg⟩ and so on. The proof is performed by induction, the order of recurrence being the order of evaluating the associated attributes.

## 5. Proof

We consider each rule in turn, rule 0 to rule 14. For each rule we organise the proof in three parts: hypothesis, conclusion and proof of the conclusion, using the semantic definitions. The termination of the proof results from the finiteness of the syntactic tree of a finite sentence and of the coherency of the description by attributes.

We show the proof on only one example, i.e. for rule 6.

R6 ⟨Form⟩ = (⟨Function⟩ ⟨Listarg⟩)

*Semantic definitions*: see Appendix 1.3, rule R6, S1 to S9

*Hypothesis*

H1:
*co* = H-LAB(Function)
*sp* = H-TOP(Function)
*la* = [*c*(*sp*-H-NARG(Function) + 1) : *c*(*sp*)]

{S-CAPPLY(Function)}
*co* = S-LAB(Function)
*sp* = H-TOP(Function) + 1
*c*(*sp*) = apply(Function, *la*, H-ALIST$^c$(Function))

H2:
*co* = H-LAB(Listvarg)
*sp* = H-TOP(Listvarg)

{S-CEVLIS(Listarg)}
*co* = S-LAB(Listarg)
*sp* = H-TOP(Listarg) + S-NARG(Listarg)
[*c*(*sp*-H-NARG(Listarg) + 1) : *c*(*sp*)] = evlis(Listarg, H-ALIST$^c$(Listarg))

*Conclusion*
*co* = H-LAB(Form)
*sp* = H-TOP(Form)   *sp* = H-TOP(Form) + 1

{S-CEVAL(Form)}
*co* = S-LAB(Form)
*c*(*sp*) = eval(Form, H-ALIST$^c$ (Form))

*Proof*

The definition S1 gives the calculation of S-CEVAL(Form):

S-CEVAL(Form) =
S-CEVLIS(Listarg) S-CAPPLY(Function)
S-LAB(Function $\underline{:\text{ POP}}$ S-NARG(Listarg) $\underline{;}$

*H-ALIST$^c$ (Form) denotes the attribute H-ALIST, where the address of the associated value has been replaced by its contents.

By the definition of H-LAB and S-LAB (S5 to S7), the elements of S-CEVAL(Form) are evaluated sequentially. In this case we can use the Hoare rule of composition:

$$\text{If } P\{Q_1\}\ R_1 \text{ and } R_1\{Q_2\}\ R \text{ then } P\{Q_1\ ;\ Q_2\}\ R$$

Thus we give the four steps of the evaluation of S-CEVAL (Form):

Initial state
*co* = H-LAB(Form) = H-LAB(Listarg) (S6)
*sp* = H-TOP(Form) = H-TOP(Listarg) (S3)

state after evaluation of S-CEVLIS(Listarg) (H2)
*co* = S-LAB(Listarg) = H-LAB(Function) (S7)
*sp* = H-TOP(Listarg) + S-NARG(Listarg) = H-TOP(Function) (S3) (S4)
[*c*(*sp* − H-NARG(Listarg) + 1) : *c*(*sp*)] = evlis(Listarg, H-ALIST$^c$(listarg)) = 1*a*

state after evaluation of S-CAPPLY(Function) (H1)
*co* = S-LAB(Function)
*sp* = H-TOP(Function) + 1
*c*(*sp*) = apply(Function, evlis(Listarg, H-ALIST$^c$(Form)), H-ALIST$^c$(Form)) (S8, S9)

final state (after execution of the POP instruction):
*co* = S-LAB(Function) + 1 = S-LAB(Form) (S5)
*sp* = H-TOP(Function) + 1 − S-NARG(Listarg) = H-TOP(Form) + 1 (S4)
*c*(*sp*) = eval(Form, H-ALIST$^c$(Form)) (see definition of eval in this case—Appendix 3—and the POP instruction does not change the top content of the argument-stack).

QED.

## 6. Conclusion

We have shown, for a simple case, that it is possible to prove completely a compiler using a very modular method. It is also possible to introduce current optimisations (test on NIL, *T* clauses, open compiled functions, non-duplication of the POP . . .) without having to change all the proof, but only a few modules, eventually using new attributes. The method we have presented may be useful for the reliability of more complex compilers.

Proof for a more complex LISP compiler has been presented in Deransart (1978a) and the use of such a proof technique in order to synthesise attribute definitions has been developed in Deransart (1978b).

**Acknowledgements**

I am indebted to Messieurs Pair, Lorho and Schuman and Madame Gaudel for their contributions to this work.

## Appendix 1

1.1 *Definition of the attributes*

| *Attributes* | *Circulation** | *Type* | *Use* |
|---|---|---|---|
| CEVAL | S | Set of instructions | Generated code |
| CAPPLY | S | Set of instructions | Generated code |
| CEVCON | S | Set of instructions | Generated code |
| CEVLIS | S | Set of instructions | Generated code |
| PBOUND | S | Set of instructions | Generated code to push bounds |
| OBOUND | S | Set of instructions | Generated code to pop bounds |
| LAB1 | H | Positive integer | Address of the first instruction to push bounds |

| | | | |
|---|---|---|---|
| LAB2 | H | Positive integer | Address of the first instruction to pop bounds |
| LAB | H, S | Positive integer | Address of the next instruction |
| OUTC | H | Positive integer | Address of the first instruction after the condition |
| NARG | H, S | Non-negative integer | Number of functions arguments |
| NVAR | H, S | Non-negative integer | Number of functions bounded variables |
| TOP | H | Index | Index of the argument-stack top |
| ALIST | H | List of pairs | Inherited context |
| TEXT | S | String | Text corresponding to an atom or an expression. |

*H = inHerited, S = Synthesised.

1.2 *Syntax*

⟨Prog⟩ = (LAMBDA(⟨listvar⟩)⟨Form⟩)
⟨Form⟩ = number | ( ) | NIL | T | Ident | (QUOTE⟨S-expression⟩) | (COND⟨listcond⟩) | (⟨Function⟩ ⟨listarg⟩)
⟨Function⟩ = Ident | (LAMBDA(⟨listvar⟩)⟨Form⟩)
⟨Listarg⟩ = ⟨Form⟩ ⟨listarg⟩ | empty
⟨Listcond⟩ = (⟨Form⟩ ⟨Form⟩)⟨listcond⟩ | (⟨Form⟩ ⟨Form⟩)
⟨Listvar⟩ = Ident⟨listvar⟩ | empty
⟨S-expression⟩ = number | ( ) | NIL | Ident | (⟨list⟩) | (⟨Dotted pair⟩)
⟨List⟩ = ⟨S-expression⟩ ⟨List⟩ | ⟨S-expression⟩
⟨Dotted pair⟩ = ⟨S-expression⟩ . ⟨S-expression⟩

1.3 *Translation*

*Notations*:

Attributes are written with capital letters and the node is normally written with parenthesis: H-ALIST($x$)

Generated symbols are underlined : : PUSHQ TEXT(S-expr) ;

The calculations are written between brackets.

R0 ⟨Prog⟩ = (LAMBDA(⟨Listvar⟩)⟨Form⟩)
S-CAPPLY(Prog) = S-CEVAL(Form) S-PBOUND(Listvar) [S-LAB(Form) + S-NVAR(Listvar)] : RETURN ; S-OBOUND(Listvar)
H-LAB(Form) = [S-NVAR(Listvar) + 1]
H-TOP(Form) = S-NVAR(Listvar)
H-TOP(Listvar) = S-NVAR(Listvar)
H-ALIST(Form) = [*add to the head of the initial alist* H-ALIST(Prog) *the content of* S-ALIST(Listvar)]
H-NARG(Listvar) = S-NVAR(Listvar)
H-LAB1(Listvar) = 1
H-LAB2(Listvar) = S-LAB(Form)
S-NVAR(Prog) = S-NVAR(Listvar)

R1 ⟨Form⟩ = number
S-CEVAL(Form) = H-LAB(Form) : PUSHQ 'TEXT(number) ;
S-LAB(Form) = [H-LAB(Form) + 1]

R2 ⟨Form⟩ = ( ) | NIL | T
as above, but the generated code is:
H-LAB(Form) : PUSHQ 'NIL ;
or H-LAB(Form) : PUSHQ 'T ;

R3 ⟨Form⟩ = Ident
as above, but the generated code is:
H-LAB(Form) : PUSHV 'TEXT(Ident) [H-TOP(Form) − assoc(TEXT(Ident), H-ALIST(Form))] ;

R4 ⟨Form⟩ = (QUOTE⟨S-expression⟩)
as above, but the generated code is:
H-LAB(Form) : PUSHQ 'TEXT(S-expression) ;

R5 ⟨Form⟩ = (COND⟨Listcond⟩)
S-CEVAL(Form) = S-CEVCON(Listcond)
H-TOP(Listcond) = H-TOP(Form)
S-LAB(Form) = S-LAB(Listcond)
H-LAB(Listcond) = H-LAB(Form)
H-OUTC(Listcond) = S-LAB(Listcond)
H-ALIST(Listcond) = H-ALIST(Form)

R6 ⟨Form⟩ = (⟨Function⟩⟨Listarg⟩)
(S1) S-CEVAL(Form) = S-CEVLIS(Listarg) S-CAPPLY(Function) S-LAB(Function) : POP S-NARG(Listarg) ;
(S2) H-NARG (Function) = S-NARG(Listarg)
(S3) H-TOP(Listarg) = H-TOP(Form)
(S4) H-TOP(Function) = [H-TOP(Form) + S-NARG(Listarg)]
(S5) S-LAB(Form) = [S-LAB(Function) + 1]
(S6) H-LAB(Listarg) = H-LAB(Form)
(S7) H-LAB(Function) = S-LAB(Listarg)
(S8) H-ALIST(Listarg) = H-ALIST(Form)
(S9) H-ALIST (Function) = H-ALIST(Form)

R7 ⟨Function⟩ = Ident
S-CAPPLY(Function) = H-LAB(Function) : LINK H-NARG(Function) TEXT(Ident) ;
S-LAB(Function) = [H-LAB(Function) + 1]

R8 ⟨Function⟩ = (LAMBDA(⟨Listvar⟩)⟨Form⟩)
S-CAPPLY(Function) = S-PBOUND(Listvar) S-CEVAL(Form) S-OBOUND(Listvar)
H-LAB1(Listvar) = H-LAB(Function)
H-LAB2(Listvar) = S-LAB(Form)
H-TOP(Form) = H-TOP(Function)
H-TOP(Listvar) = H-TOP(Function)
H-ALIST(Form) = [*add to the head of* H-ALIST(Function) *the elements of the list* S-ALIST(Listvar)]
S-LAB(Function) = [S-LAB(Form) + S-NVAR(Listvar)]
H-LAB(Form) = [H-LAB(Function) + S-NVAR(Listvar)]
H-NARG(Listvar) = H-NARG(Function)
H-NVAR(Listvar) = O

R9 ⟨Listarg⟩ = ⟨Form⟩⟨Listarg⟩

```
    S-CEVLIS(Listarg)      = S-CEVAL(Form)
                             S-CEVLIS(Listarg')
    S-NARG(Listarg)        = [S-NARG(Listarg') + 1]
    H-TOP(Listarg')        = [H-TOP(Listarg) + 1]
    H-TOP(Form)            = H-TOP(Listarg)
    S-LAB(Listarg)         = S-LAB(Listarg')
    H-LAB(Listarg')        = S-LAB(Form)
    H-LAB(Form)            = H-LAB(Listarg)
    H-ALIST(Form)          = H-ALIST(Listarg)
    H-ALIST(Listarg')      = H-ALIST(Listarg)
R10 ⟨Listarg⟩         = empty
    S-CEVLIS(Listarg)      = empty set
    S-NARG(Listarg)        = O
    S-LAB(Listarg)         = H-LAB(Listarg)
R11 ⟨Listcond⟩        = (⟨Form⟩⟨Form⟩)⟨Listcond⟩
    S-CEVCON(Listcond)     = S-CEVAL(Form)
                             S-LAB(Form)
                             : BRN[S-LAB(Form') + 2] ;
                             S-CEVAL(Form')
                             S-LAB(Form') : POP 1 ;
                             [S-LAB(Form') + 1]
                             : BR H-OUTC(Listcond) ;
                             [S-LAB(Form') + 2]
                             : POP 1 ;
                             S-CEVCON(Listcond')
    H-TOP(Form')           = [H-TOP(Listcond) + 1]
    H-TOP(Listcond')       = H-TOP(Listcond)
    S-LAB(Listcond)        = S-LAB(Listcond')
    H-LAB(Form)            = H-LAB(Listcond)
    H-LAB(Form')           = [S-LAB(Form) + 1]
    H-LAB(Listcond')       = [S-LAB(Form + 3]
    H-OUTC(Listcond')      = H-OUTC(Listcond)
    H-ALIST(Form')         = H-ALIST(Listcond)
    H-ALIST(Listcond')     = H-ALIST(Listcond)
    H-ALIST(Form)          = H-ALIST(Listcond)
R12 ⟨Listcond⟩        = (⟨Form⟩⟨Form⟩)
    S-CEVCON(Listcond)     = S-CEVAL(Form)
                             S-LAB(Form)
                             : BRN H-OUTC(Listcond) ;
                             S-CEVAL(Form')
                             S-LAB(Form') : POP 1 ;
    H-ALIST(Form)          = H-ALIST(Listcond)
    H-ALIST(Form')         = H-ALIST(Listcond)
    H-TOP(Form)            = H-TOP(Listcond)
    H-TOP(Form')           = H-TOP(Listcond)
    H-LAB(Form)            = H-LAB(Listcond)
    H-LAB(Form')           = [S-LAB(Form) + 1]
    S-LAB(Listcond)        = [S-LAB(Form') + 1]
R13 ⟨Listvar⟩         = Ident⟨Listvar⟩
    S-PBOUND(Listvar')     = S-PBOUND(Listvar)
                             [H-LAB1(Listvar)
                             + H-NVAR(Listvar)]
                             : PUSHB
                             TEXT(Ident)
                             [H-NARG(Listvar)
                             + H-NVAR(Listvar) − 1];
    S-OBOUND(Listvar')     = S-OBOUND(Listvar)
                             [H-LAB2(Listvar)
                             + H-NVAR(Listvar)]
                             : POPB TEXT(Ident) ;
    H-TOP(Listvar')        = H-TOP(Listvar)
    H-NARG(Listvar')       = H-NARG(Listvar)
    S-NVAR(Listvar)        = [S-NVAR(Listvar') + 1]
    H-NVAR(Listvar')       = [H-NVAR(Listvar) + 1]
    S-ALIST(Listvar)       = [add to the head of
                             S-ALIST(Listvar') the pair
                             (TEXT(Ident).
                             [H-TOP(Listvar)
                             + H-NVAR(Listvar')
                             − H-NARG(Listvar) + 1])]
    H-LAB1(Listvar')       = H-LAB1(Listvar)
    H-LAB2(Listvar')       = H-LAB1(Listvar)
R14 ⟨Listvar⟩         = empty
    S-PBOUND(Listvar)      = empty set
    S-OBOUND(Listvar)      = empty set
    S-NVAR(Listvar)        = 0
    S-ALIST(Listvar)       = empty list
```

## Appendix 2 Object language

2.1 *Target abstract machine*

(*a*) Memory zone including potentially infinite number of cells.
(*b*) Potentially unlimited atom table.
(*c*) Argument-stack denoted by the top *sp* and *c*(*r*), contents of the element of row *r*.
(*d*) Bound-stacks denoted by *spx* and *c*(*r*).
(*e*) Return-stack.
(*f*) Zone for programs.
(*g*) Instruction counter (*co*).

The execution cycle is: execute the instruction with label contained in *co*, as long as possible. *co* is initialised with 1.

2.2 *Instructions* (Object code)

```
Instructions     Effect
PUSHQ adtc       sp         := sp + 1 ;
                 c(sp)      := adtc ;
                 co         := co + 1.
PUSHV X n       ⎧sp         := sp + 1 ;
                ⎨c(sp)      := c(sp − n − 1) ;
                ⎩co         := co + 1.
PUSHB X n       ⎧spX        := spX + 1 ;
                ⎨cX(spX)    := sp − n ;
                ⎩co         := co + 1 .
POPB X          ⎧spX        := spX − 1 ;
                ⎩co         := co + 1.
LINK n name     ⎧sp         := sp + 1 ;
                ⎨c(sp)      := name(n arguments) ;
                ⎩co         := co + 1.
POP n           ⎧c(sp − n)  := c(sp) ;
                ⎨sp         := sp − n ;
                ⎩co         := co + 1.
POP             ⎧sp         := sp − 1 ;
                ⎩co         := co + 1.
BRN lab         ⎧co         := if c(sp) = nil
                ⎨                then lab
                ⎩                else co + 1.
BR lab           co         := lab
RETURN           non described.
```

Remarks: the return-stack is not described here. *adtc* represents addresses in atom-table or cell zone.

## Appendix 3 Evaluator written in an ALGOL 68-like notation

```
eval(form, alist)
 if form = number, (  ) or NIL, T
 then Form
 elif form = Ident
 then assoc (Ident, alist)
 elif form = "(QUOTE S)"
 then S
 elif form = "(COND.LS)"
 then evcon(LS, alist)
 else co form = "(S1. S2)" co
   apply(S1, evlis(S2, alist), alist)
 fi
```

```
apply(function, listvarg, alist)
  if function = Ident
  then exec(Ident, Listarg, alist)
  else co function = (LAMBDA S1. S2) co
    eval(S2, pairlis(S1, listarg, alist))
  fi
evlis(Listarg, alist)
  if Listarg = NIL
  then NIL
  else co Listarg = (A1 Listarg') co
    (eval(A1, alist). evlis(Listarg', alist))
  fi
evcon(Listcond, alist)
  if Listcond = NIL
  then NIL
  else co Listcond = ((S1 S2) Listcond')) co
    if eval(S1, alist) ≠ NIL
    then eval(S2, alist)
    else evcon(Listcond', alist)
    fi
  fi
pairlis(listvar, listvarg, alist)
  if listvar = NIL
  then alist
  else co Listvar = (V. LV), Listvarg = (VA. LVA) co
               ((V . VA) . pairlis(LV, LVA, alist))
  fi
```

## Appendix 4 Example of object code

*Function FACT (Factorial)*
Source Language:

```
(LAMBDA(N) (COND((EQUAL N 0)1)
                 (T(TIMES N(FACT(SUB1 N))))))
```

*Object code;*

```
1  : PUSHB 'N 0 ;          12 : BRN 22 ;
2  : PUSHV 'N 0 ;          13 : PUSHV 'N 0 ;
3  : PUSHQ '0 ;            14 : PUSHV 'N 1 ;
4  : LINK 2 EQUAL ;        15 : LINK 1 SUB1 ;
5  : POP 2 ;               16 : POP 1 ;
6  : BRN 10 ;              17 : LINK 1 FACT ;
7  : PUSHQ '1 ;            18 : POP 1 ;
8  : POP 1 ;               19 : LINK 2 TIMES ;
9  : BR 21 ;               20 : POP 2 ;
10 : POP ;                 21 : POP 1 ;
11 : PUSHQ 'T ;            22 : POPB 'N
                           23 : RETURN
```

**References**

Deransart, P. (1977). Definition and implementation of a Lisp system, using semantic attributes, *5th annual III Conference.*

Deransart, P. (1978a). Technique de preuve par attributs appliquée à un compilateur Lisp, Coopération Franco-Soviétique, Paris.

Deransart P. (1978b). Proof and Synthesis of Semantic Attributes in compiler definition, IRIA—LABORIA report no. 333.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming, *CACM*, Vol. 12 no. 10.

London, R. L. (1971). Correctness of two compilers for a Lisp subset. Repot n° CS 240, University of Stanford, October 1971.

Lorho, B. (1974). De la définition à la traduction des langages de programmation: méthode des attributs sémantiques, Thèse, Université Paul Sabatier de Toulouse, 29 Novembre 1974 (French).

Lux, A. (1975). Etude d'un modèle abstrait pour une machine Lisp et de son implémentation, Thèse—Université Scientifique et Médicale de Grenoble, 19 mars 1975 (French).

Pair, C., Amirchahy, M. and Neel D. (1976). Preuve de descriptions de traitement de textes par Attributs, IRIA-LABORIA (French).

## Book reviews

*Current Trends in Programming Methodology, Vol III Software Modeling* edited by R. Jeh and K. M. Chandy, 1978; 379 pages. (*Prentice-Hall*, £13·85)

Mathematical modelling (often less aptly named operations research) has been widely used as an aid to management in improving the efficiency of a business. It is now recognised that the varied techniques which have evolved can also be applied to the design and tuning of computer systems, often noted for their misuse of resources. This volume, with an odd title, attempts to describe the main methods of modelling and apply them to a range of problems associated with computer performance. The individual authors of the nine chapters are able to write with authority.

The major topics discussed are statistical analysis of performance data; the analysis of a network of queues using simulation, theory and a comprehensive package called RESQ (Research Queuing Analyzer); graph theory applied to parallel computation, frequency monitoring, job scheduling; deadlocks; memory management; mathematical programming.

Each chapter is well supplied with examples relating to computer performance.

The book should be of value to the traditional system designer seeking additional quantitative support in decision-making; it should also interest the OR worker who is seeking new challenging problems. Finally, all undergraduates in computer science should be exposed to an introduction to the topics discussed in the book. It is well produced and has an extensive bibliography of 313 references; surprisingly, there is no index.

T. Vickers (Twickenham)

*Introduction to Formal Language Theory*, by M. A. Harrison, 1978; 594 pages. (*Addison-Wesley*, £15·75)

This book covers most of the material one would expect from the title. The author says he assumes knowledge of finite automata, Turing machines and computer programming; I feel maths is a more important prerequisite. The approach taken is relatively formal, proofs of the major theorems (constructions) are given but some proofs are left to the reader.

The first half of the book covers type 2 and 3 languages/grammars and their associated automata including some discussion on ambiguity and decision problems. A brief look at the rest of the Chomsky hierarchy and some representation theorems then lead to the meat of the remaining half of the book. This is one of the best accounts I have seen of deterministic languages and parsing problems.

The material as presented develops in a logical and coherent way but I would have preferred to have seen more explanation for the motivation of the study. My major criticism of the book is that the computer science professional (or indeed student) could work all the way through it yet still validly ask the question 'why bother?'. There are many good reasons for studying this subject and I wish the author had brought these out.

The book is well printed with good examples, problems and illustrations. There are commendably few printing errors for a book of this complexity.

In summary, I find this book sound yet unexceptional; it is neither inspired nor bad.

D. Simpson (Sheffield)