SIS — *Semantics Implementation System*

Tested Examples

by

Peter Mosses

DAIMI MD-33
August 1979
SIS — Semantics Implementation System

Tested Examples

by
Peter Mosses

DAIMI MD-33
August 1979
This document gives some examples of language descriptions which have been tested using SIS. The amount of testing carried out varies considerably. It is NOT claimed that there are no "bugs" left in the examples.

It is hoped that a study of the examples will help the reader to use GRAM and DSL. However, one is warned against slavishly following the style and layout conventions of the examples: most of them were formulated several years ago, and neater versions could surely be made, even within the confines of the current version of DSL.

Please let me know if you find any bugs in the examples. I would also welcome further contributions to this document, especially ones illustrating a radically different style.

References to separately-published tested examples are given at the end.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREWORD</td>
<td>11</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>Lambda-Calculus with Atoms</td>
<td>1</td>
</tr>
<tr>
<td>LOOP</td>
<td>5</td>
</tr>
<tr>
<td>PL</td>
<td>11</td>
</tr>
<tr>
<td>M-Lisp</td>
<td>21</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>31</td>
</tr>
</tbody>
</table>

- 111 -
This is just the lambda-calculus, with the natural numbers as atoms. The symbol '\' represents 'lambda'.

The identifiers 'plus' and 'mult' are pre-defined to be suitable operators on the natural numbers.

The example serves as a gentle introduction to SIS for students -- give them the LC-Parser, let them work out and test LC-Semantics for themselves.

Example Program:

```latex
(\ double.
 \ twice.
 \ thrice.

thrice(twice double)1
)
(\n. plus n n)
(\f. \n. f(f(n)))
(\f. \n. f(f(f(n))))
```

Result of Compiling:

```
LAMB "LC-Semantics(Program)"
"N"NODE<64>
END
```

Degree of Testedness: High.
"LC-Parser"

**SYNTAX**

```plaintext
exp ::= \" \" ide \".\" exp \\
/ exp-a : exp-a \\
exp-a ::= exp-a exp-b \\
/ exp-b : exp-b \\
exp-b ::= ide : ide \\
/ num : num \\
/ \"(" exp \")\" \\
ide ::= \"IDE\" q \\
num ::= \"NUM\" n \\

**DOMAINS**

exp, exp-a, exp-b : Exp \\

**LEXIS**

```plaintext
exp ::= symb+ : CONC symb+ \\
symb ::= ide : <OUT"IDE", ide> \\
/ num : <OUT"NUM", num> \\
/ layout+ : <> \\
ide ::= letter+ : QUOTE letter+ \\
letter ::= \"a\"...\"z\" \\
num ::= digit+ : NUMBER digit+ \\
digit ::= \"0\"...\"9\" \\
layout ::= \"\" / CC"C" / CC"L" / CC"T" \\

END
```
Lambda-Calculus with Atoms

DSL
"LC-Semantics"

DOMAINS

<table>
<thead>
<tr>
<th>Syntactic</th>
<th>Semantic</th>
</tr>
</thead>
</table>
| ide : Ide = ["IDE" Q] ; | f : F = E \rightarrow E ;
| num : Num = ["NUM" N] ; | n : N ;
| | q : Q ;
| | r : R = Ide \rightarrow E ;

DEF

ee(exp0)(r): E =

CASE exp0
/["\" Ide "," exp] -> LET f = LAM e. ee(exp)(r\ide<-e)
IN [f];
/[exp1 exp2] -> LET [f] = ee(exp1)(r)
IN f(ee(exp2)(r));
/["IDE" q] -> r(exp0);
/["NUM" n] -> [n]
/["(" exp ")"] -> ee(exp)(r)
ESAC

WITH r0 = LAM ["IDE" q].

CASE q
/ "plus" -> fun(LAM n1. LAM n2. n1 PLUS n2)
/ "mult" -> fun(LAM n1. LAM n2. n1 MULT n2)
ESAC

WITH fun(g : (N \rightarrow N) \rightarrow N) : [F] =

LET f1[n1] : [F] =
LET f2[n2] : [N] =
LET n = g(n1)(n2)
IN [n]
IN [f2]
IN [f1]
IN LAM exp. ee(exp)(r0)
END
This example aims to help comparison of DSL with the more traditional notation used in Tennent's survey paper [Comm.ACM 19:8]. The reader should refer to the original paper for an informal explanation of the semantics of LOOP.

Example Program:

```plaintext
READ n;
TO n DO
    n := n + 1;
WRITE n
```

Example Data:

```plaintext
LAMB "Data"
<3>
END
```

Result of Interpreting:

```plaintext
LAMB "LOOP-Semantics(Program)(Data)"
<6>
END
```

Degree of Testedness: High.
GRAM "LOOP-Parser"

SYNTAX

\[
\begin{align*}
\text{prog} & ::= \text{read-cmd } ";" \text{ cmd-seq } ";" \text{ write-cmd } \; \\
\text{read-cmd} & ::= \text{"READ" } \text{var}*-";" \; : \{ \text{"READ" } \text{var}\} \; \\
\text{write-cmd} & ::= \text{"WRITE" } \text{exp}*-";" \; : \{ \text{"WRITE" } \text{exp}\} \; \\
\text{cmd-seq} & ::= \text{cmd-seq } ";" \text{ cmd} \; : \{ \text{cmd-seq } ";" \text{ cmd}\} \; / \\
\text{cmd} & ::= \text{var } ";" \text{ exp} / \\
& \quad \text{"\text{\textbackslash T}" } \text{ exp } \text{"\text{\textbackslash D}" } \text{ cmd } / \\
& \quad \text{"\text{\textbackslash (" \text{cmd-seq } ")"} } : \text{ cmd-seq } \; \\
\text{exp} & ::= \text{exp } \text{add-op} \text{ exp-a } / \\
& \quad \text{exp-a } : \text{exp-a } \; \\
\text{add-op} & ::= \text{"+" } / \text{"-" } ; \\
\text{exp-a} & ::= \text{exp-a } \text{mult-op} \text{ exp-b } / \\
& \quad \text{exp-b } : \text{exp-b } \; \\
\text{mult-op} & ::= \text{"\text{\textbackslash *}" } / \text{"\text{\textbackslash /}" } ; \\
\text{exp-b} & ::= \text{var } / \\
& \quad \text{num } ; \\
\text{var} & ::= \text{"\text{VAR" } q } \; : \; \text{q } ; \\
\text{num} & ::= \text{"\text{NUM" } n } \; : \; \text{n } ; \\
\end{align*}
\]

DOMAINS

\[
\begin{align*}
\text{cmd-seq, cmd} & : \text{Cmd} ; \\
\text{exp, exp-a, exp-b} & : \text{Exp} ; \\
\text{add-op, mult-op} & : \text{Op} ; \\
\end{align*}
\]
LEXIS

program ::= word+ : CONC word+ ;

word ::= var : <OUT"VAR", var> /
num : <OUT"NUM", num> /
comment : <> /
layout+ : <> ;

var ::= letter letter-digit*
      : QUOTE(letter PRE letter-digit*) ;

letter ::= "a"..."z" ;
letter-digit ::= "a"..."z" / "0"..."9" ;

num ::= digit+ : NUMBER digit+ ;

digit ::= "0"..."9" ;

comment ::= "C" "M" "T" comment-char* : ? ;

comment-char ::= "!" ;

layout ::= "-" / CC"C" / CC"L" / CC"T" ;

END
DSL "LOOP-Semantics"

The "direct" style of semantics is used, to enable comparison with Tennent's semantics for LCOP [CACM 19:8]. Expressions cannot have side-effects in LCOP. As there are no declarations in LOOP, environments are not used in the semantics.

DOMAINS

SYNTACTIC:

\[
\begin{align*}
\text{prog} : & \quad \text{Prog} = \{\text{Read-cmd }; \text{Cmd }; \text{Write-cmd}\}; \\
\text{read-cmd} : & \quad \text{Read-cmd} = \{\text{"READ" Var *}\}; \\
\text{write-cmd} : & \quad \text{Write-cmd} = \{\text{"WRITE" Exp +}\}; \\
\text{cmd} : & \quad \text{Cmd} = \{\text{Cmd }; \text{Cmd}\} / \{\text{Var }:=\text{ Exp}\} / \{\text{"TO" Exp }"DO" \text{ Cmd}\} / \{\text{"(" Cmd ")"}\}; \\
\text{exp} : & \quad \text{Exp} = \{\text{Exp Op Exp}\} / \{\text{Var}\} / \{\text{Num}\}; \\
\text{op} : & \quad \text{Op} = \{\text{"+" }/ \text{ "-" }/ \text{ "*" }/ \text{ "/"}\}; \\
\text{var} : & \quad \text{Var} = \{\text{Q}\}; \\
\text{num} : & \quad \text{Num} = \{\text{N}\};
\end{align*}
\]

SEMANTIC:

\[
\begin{align*}
\text{s} : & \quad \text{S} = \text{Var} \to \text{N}; \\
\text{n} : & \quad \text{N}; \\
\text{q} : & \quad \text{Q}; \\
\end{align*}
\]

FUNCTIONS:

\[
\begin{align*}
\text{pp} : & \quad \text{Prog} \to \text{N*} \to \text{N+}; \\
\text{cc} : & \quad \text{Cmd} \to \text{S} \to \text{S}; \\
\text{ee-list} : & \quad \text{Exp +} \to \text{S} \to \text{N+}; \\
\text{ee} : & \quad \text{Exp} \to \text{S} \to \text{N}; \\
\text{oo} : & \quad \text{Op} \to \langle \text{N,N} \rangle \to \text{N}; \\
\text{repeat} : & \quad \text{N} \to \langle \text{S} \to \text{S} \rangle \to \text{S}; \\
\text{update-list} : & \quad \langle \text{Var}, \text{N*} \rangle \to \text{S} \to \text{S}; \\
\text{initial-s} : & \quad \text{S}; \\
\text{update} : & \quad \langle \text{Var}, \text{N} \rangle \to \text{S} \to \text{S};
\end{align*}
\]
DEF
pp[read-cmd ";" cmd ";" write-cmd](n*) : N+ = ! 32
LET ["READ" var*] = read-cmd ! 33
ALSO ["WRITE" exp+] = write-cmd ! 34
LET s1 = update-list(var*,n*)(initial-s) ! 35
LET s2 = cc(cmd)(s1) ! 36
IN ee-list(exp+)(s2) ! 37

WITH cc(cmd0)(s): S = ! 38
CASE cmd0 ! 39
/[cmd1 ";" cmd2] -> cc(cmd2)(cc(cmd1)(s)) ! 40
/[var "=" exp] -> LET n = ee(exp)(s) ! 41
IN update(var,n)(s) ! 42
/["TO" exp "DO" cmd] -> LET n = ee(exp)(s) ! 43
IN repeat(n)(cc(cmd))(s) ! 44
/["(" cmd ")"] -> cc(cmd)(s) ! 45
ESAC ! 46

WITH ee-list(exp0+)(s): N+ = ! 47
CASE exp0+ ! 48
/<exp> -> <ee(exp)(s)> ! 49
/exp PRE exp+ -> ee(exp)(s) PRE ee-list(exp+)(s) ! 50
ESAC ! 51

WITH ee(exp0)(s): N = ! 52
CASE exp0 ! 53
/[exp1 op exp2] -> LET n1 = ee(exp1)(s) ! 54
ALSO n2 = ee(exp2)(s) ! 55
IN oo(op)(n1,n2) ! 56
/[var] -> content(var)(s) ! 57
/[num] -> num : N ! 58
ESAC ! 59
10 TESTED EXAMPLES

WITH oo(op)(n1,n2): N =

CASE op

/""+" -> n1 PLUS n2
/""-" -> n1 MINUS n2 ! gives ? if n2 greater than n1
/""*" -> n1 MULT n2
/""/" -> n1 DIV n2 ! gives ? if n2 is zero
ESAC

WITH repeat(n)(c:(S -> S))(s): S =

n EQ ? -> ?,
 n EQ 0 -> s,
repeat(n MINUS 1)(c)( c(s) )

WITH update-list(var0*,n0*)(s): S =

SIZE var0* EQ 0 -> s,
 LET var PRE var* = var0*
 ALSO n PRE n* = n0*
 IN update-list(var*,n*)( update(var,n)(s) )

WITH initial-s : S =
LAM var. ?

WITH update(var,n)(s): S =
 s \ var <- n

WITH content(var)(s): N =
s(var)

IN pp : (Prog -> N* -> N+)
END
This example deals with a (not very) original language designed for use in connection with a course on denotational semantics (using Joe Stoy's book).

The students were given the abstract syntax of PL, and the PL-Machine (auxiliary functions) -- they had to work out and test the rest themselves. There were some difficulties in the beginning, in getting the PL-Parser to produce the correct labels in the parse-trees. This was due (in part) to the fact that Int and Num are handled differently here, compared to the Lambda-Calculus with Atoms, which was used as the initial exercise.

Example Program:

```
BEG
  CON n = 27;
  VAR a := 0
  IN
  WRITE n;
  WRITE a;
  a := n;
  WRITE a;
  BEG
    VAR a := 0;
    VAR n := 0
    IN
      WRITE a + n;
      a := a - 1;
      n := - 2;
      WRITE a + n
    END;
    WRITE a;
    WRITE n
  END
END
```

Example Data:

```
LAMB "Data"
<27>
END
```

Result of Compiling and Executing:

```
LAMB "PL-Semantics(Program)(PL-Machine)(Data)"
< "27", "0", "27", "0", "-3", "27", "27", "Terminated OK">
END
```

Degree of Testedness: Medium.
"PL-Parser"

SYNTAX

```plaintext
cmd ::= "BEGIN" cmd-seq "END" / "BEGIN" dec-seq "IN" cmd-seq "END" / "IF" bool-exp "DO" cmd / "WHILE" bool-exp "DO" cmd / "BREAK" / "WRITE" exp ;

cmd-seq ::= cmd-seq ";" cmd / cmd ;

dec-seq ::= dec-seq ";" dec / dec ;

dec ::= "CON" ide "=" exp / "VAR" ide "=" exp ;

exp ::= bool-exp "->" exp "," exp / bool-exp : bool-exp / int-exp : int-exp ;

bool-exp ::= bool-exp-a log-op bool-exp-a / int-exp rel-op int-exp / bool-exp-a : bool-exp-a ;

bool-exp-a ::= "TRUE" / "FALSE" / ide / 
"(" bool-exp ")" : bool-exp ;

int-exp ::= int-exp-a int-op int-exp-a / int-exp-a : int-exp-a ;

int-exp-a ::= num / "=" num / ide / 
"READ" / 
"(" int-exp ")" : int-exp ;

dec ::= 
"IDE" q : q ;

num ::= 
"NUM" n : n ;

log-op ::= 
"&" / "/" ;

rel-op ::= 
"<" / ">" ;

int-op ::= 
"*" / "+" / "-" / "+" ;

DOMAINS

cmd-seq, cmd : Cmd ;
dec-seq, dec : Dec ;
exp, bool-exp, bool-exp-a, int-exp, int-exp-a : Exp ;
log-op, rel-op, int-op : Op ;
```
LEXIS

prog ::= symbol+ : CONC symbol+ ;

symbol ::= identifier : <OUT "IDE", identifier> /
numeral : <OUT "NUM", numeral> /
layout+ : <> /
":=" comment* ::= ;

identifier ::= lower+ : QUOTE.lower+ ;
numeral ::= digit+ : NUMBER digit+ ;

lower ::= "a"..."z" ;
digit ::= "0"..."9" ;

layout ::= " " / CC"C" / CC"L" / CC"P" / CC"T" ;

comment =\= CC"C" / CC"L" / CC"P" ;

END
DSL "PL-Semantics"

DOMAINS

! SYNTACTIC:

```plaintext
cmd : Cmd = [Cmd ";" Cmd] /
    ["BEGIN" Cmd "END"] /
    ["IF" Dec "IN" Cmd "END"] /
    ["HEAD"] /
    ["WRITE"] Exp ;
```

```plaintext
dec : Dec = [Dec ";" Dec] /
    ["CON" Ide ";" Exp] /
    ["VAR" Ide ";" Exp] ;
```

```plaintext
    ["TRUE"] /
    ["FALSE"] /
    [Num] /
    ["=" Num] /
    ["READ"] /
    [Exp] ;
```

```plaintext
ide : Ide = Q ;
```

```plaintext
num : Num = N ;
```

```plaintext
op : Op = ";" / "/" / "<" / "=" / "+" / "-" / "*" ;
```

! SEMANTIC:

```plaintext
A ;
C = S -> A ;
D = E / [C] ;
E = V / [L] ;
I ;
K = E -> C ;
L ;
N ;
Q ;
R = Ide -> D ;
S ;
T ;
V = [N] / ["-" N] / [T] ;
X = R -> C ;
```

! FUNCTIONS:

```plaintext
cc := Cmd -> R -> C -> C ;
dd := Dec -> R -> X -> C ;
ee := Exp -> R -> K -> C ;
```
LAM cmd'. ! To give this segment the correct functionality for
! use with the compile command!

! PRIMITIVES:
LAM <
wrong : (Q -> C),
op-val : (Q -> (V,V) -> K -> C),
new-loc : (K -> C),
content : (L -> K -> C),
update : ((L,V) -> C -> C),
read : (K -> C),
write : (V -> C -> C),

exec : ((R -> C -> C) -> I -> A)
>

! MAIN SEMANTIC FUNCTIONS:

DEF cc(cmd0)r;c : C =

CASE cmd0 |
/ [cmd1 ";" cmd2] -> cc(cmd1)r; cc(cmd2)r; c
/ ["BEGIN" cmd "END"] -> cc(cmd)r; c
/ ["BEGIN" dec "IN" cmd "END"] -> dd(dec)r; LAM r'. cc(cmd)r'; c
/ [ide "=" exp] -> CASE r(ide)
/ [] -> ee(exp)r; LAM v.
/ d -> Wrong"ide:=exp"
ESAC

/ ["IF" exp "DO" cmd] -> ee(exp)r; LAM [t].
( t -> cc(cmd)r, LAM c'. c''); c
/ ["WHILE" exp "DO" cmd] -> LET r' = r \ "BREAK" <= (c) IN
FIXLAM c'.
ee(exp)r; LAM [t].
 t -> cc(cmd)r'; c', c

/ ["BREAK"] -> CASE r("BREAK")
/ [c'] -> c'
/ d -> Wrong"BREAK"
ESAC

/ ["WRITE" exp] -> ee(exp)r; LAM v. write(v); c

ESAC
WITH dd(dec0)r; x : C =
CASE dec0
/ [decl ";" dec2] -> dd(dec1)r; LAM r'. dd(dec2)r'; x
/ ["CON" ide "=" exp] -> ee(exp)r; LAM v. x(r\ide<-v)
/ ["VAR" ide ":=" expl] ->
  ee(expl)r; LAM v,
  new-loc; LAM [1].
  update(1,v); x(r\ide<-[1])
ESAC

WITH ee(exp0)r; k : C =
CASE exp0
/ [exp1 "->" exp2 "," exp3] ->
  ee(exp1)r; LAM (t).
  (t -> ee(exp2)r, ee(exp3)r); k
/ [exp1 op exp2] ->
  ee(exp1)r; LAM v1.
  ee(exp2)r; LAM v2.
  op=val(op)(v1,v2); k
/ ["TRUE"] ->
  k[T]
/ ["FALSE"] ->
  k[F]
/ [num] ->
  LET n = num IN k[n]
/ ["-" num] ->
  LET n = num IN k["-"n]
/ ["READ"] ->
  read; k
/ [ide] ->
  CASE r(ide)
  /[1] -> content(1); k
  /[t] -> k[t]
  /[n] -> k[n]
  /["-"n] ->
  k["-"n]
ESAC

IN exec( cc(cmd') ) : (I -> A)

END
DSL "PL-Machine"

DOMAINS

d : A = Q* ; ! Answers
C : S -> A ; ! command Continuations
d : D ; ! Denoted values
e : E = V / [L] ; ! Expressed values
i : I = N* ; ! Inputs
k : K = E -> C ; ! expression Continuations
l : L = N ; ! Locations
m : M = L -> V ; ! Memories
n : N ; ! Natural numbers
q : Q ; ! Quotations
r : R = ? -> D ; ! environments
s : S = <M, L, I> ; ! States
t : T ; ! Truths
v : V = [N] / ["" -" N] / [T] ; ! storable Values

LET wrong(q)s : A =
<"Error: ", q>

DEF op\-val(q)(v1,v2); k : C =

CASE <q, v1, v2>
/ "&", [t1], [t2] \-> LET t = t1 AND t2 IN k[t]
/ "/", [t1], [t2] \-> LET t = t1 OR t2 IN k[t]
/ "",
"", [n1], [n2] \-> LET t = n1 LS n2 IN k[t]
/ ",", ["" -"n1], ["" -"n2] \-> LET t = n2 LS n1 IN k[t]
/ ",", [n1], ["" -"n2] \-> LET t = n1 EQ n2 IN k[T]
/ "",
"", [n1], ["" -"n2] \-> LET t = n2 EQ n1 IN k[T]
/ "",
"", [n1], [n2] \-> LET t = n1 PLUS n2 IN k[t]
/ "",
"", ["" -"n1], [n2] \-> LET t = n1 PLUS n2 IN k[T]
/ "",
"", [n1], ["" -"n2] \-> LET t = n1 PLUS n2 IN k[T]
/ "",
"", [n1], [n2] \-> op\-val("-")([n2],[n1]); k
/ "",
"", [n1], ["" -"n2] \-> op\-val("-")([n2],[n1]); k
/ "",
"", [n1], [n2] \-> n1 GE n2 ->
LET n = n1 MINUS n2 IN k[n],
LET n = n2 MINUS n1 IN k["" -"n]
/ "",
"", ["" -"n1], ["" -"n2] \-> op\-val("-")([n2],[n1]); k
/ "",
"", ["" -"n1], [n2] \-> LET n = n1 PLUS n2 IN k["" -"n]
/ "",
"", [n1], ["" -"n2] \-> op\-val("+")([n1],[n2]); k
/ "",
"", [n1], [n2] \-> LET n = n1 MULT n2 IN k[n]
/ "",
"", ["" -"n1], ["" -"n2] \-> LET n = n1 MULT n2 IN k["" -"n]
/ "",
"", [n1], ["" -"n2] \-> LET n = n1 MULT n2 IN k["" -"n]
/ ? \-> wrong"? op\-val"
LET init-r : R =
LAM ?, ?

LET init-s(i) : S =
<LAM 1. ?, 0, i>

LET new-loc(k)(s) : A =
LET <m, l, i> = s
LET l1 = l PLUS 1
IN k(l1)<m, l1, i>

LET content(l)(k)(s) : A =
LET <m, l', i> = s
LET v = m(l)
IN (v NE ?) AND (l LE l') -> k(v)(s),
    wrong"? content"(s)

LET update(l,v)(c)(s) : A =
LET <m, l1, i> = s
IN l LE l1 -> c<m\l'-v, l1, i>,
    wrong"? update"(s)
LET read(k)(s) : A =

LET <m, l, n0*> = s
IN
CASE n0*
/  n PRE n* -> k[n]<m, l, n*>
/  <> -> Wrong"? read"(s)
ESAC

DEF write(v)(c)(s) : A =
quote(v) PRE c(s)

WITH quote(v) : Q =
CASE v
/  [n] -> LET NUMBER q* = n
    IN QUOTE q*
/  ["-"n] -> LET NUMBER q* = n
    IN QUOTE("-"PRE q*)
/  [t] -> t -> "TRUE", "FALSE"
ESAC.

LET exec(f:(R -> C -> C))(i) : A =
f(init-r)(LAM s. "Terminated CK") (init-s(i))

IN
<wrong, op-val, new-loc, content, update, read, write, exec>
END
M-Lisp

This description was worked out during a short visit to Edinburgh. The aim was to take Mike Gordon’s semantics for M-Lisp (in the usual notation) and convert it to DSL as simply as possible.

No problems were encountered -- apart from choosing systematic names for the meta-variables -- and the final product was used to illustrate a talk on SIS. It was not felt relevant to implement all the usual primitive M-Lisp operators.

Note the use of the pretty-printer for the output (S-expressions) -- the LAMB-notation for NODEs makes the original output rather unreadable.

Example Program:

```lisp
label(append; \[ll;12];
  [eq(ll;NIL) -> 12;
   T -> cons[car(ll);
      append[cdr(ll);12]]
  ]
]
[(A B);(C D)]
```

Result of Compiling and Applying:

```
LAMB "M-Lisp-Semantics(Program)(M-Lisp-Machine)"
 "(S.S)"NODE"A",
 "(S.S)"NODE"B",
 "(S.S)"NODE"C", "(S.S)"NODE"D", "NIL"
```

Result of Applying Pretty:

```
LAMB "Pretty(M-Lisp-Semantics(Program)(M-Lisp-Machine))"
 < "A", "B", "C", "D">
```

Degree of Testedness: Low.
GRAM  "M-Lisp-Parser"

SYNTAX

form ::= s-expr /
       var /
       func "[" form* ";" "]" /
       "[" case+ ";" "]" ;
case ::= form ";" form ;
var ::= ide ;

func ::= "car" /
       "cdr" /
       "cons" /
       "atom" /
       "eq" /
       ide /
       "\" "[" " var* ";" "]" ; form "]" :
       "\" "var* ";" form "]" ;
       "label" "[" ide ";" func "]" ;
       "label" ide ";" func ;
ide ::= "ID" q ; q ;

s-expr ::= atom :
          s-expr /
          "(" s-expr ";" s-expr ")" ;
          "(" s-expr-seq ")" :
          s-expr-seq ;

s-expr-seq ::= s-expr s-expr-seq 
             [: "(" s-expr ";" s-expr-seq ")" ] ;
             "NIL" ;

atom ::= "AT" q ; q ;

DOMAINS

form : E ;
case : C ;
func : Fn ;
ide : F ;
var : X ;
s-expr : S ;
s-expr-seq : S ;
atom : Q ;
LEXIS

prog ::= symb+ ; CONC symb+ ;

symb ::= layout+ ; <> /
atom ::= <OUT"AT", atom> /
ide ::= <OUT"ID", ide> ;
atom ::= u ud* ; QUOTE(u PRE ud*) ;
ide ::= l ld* ; QUOTE(l PRE ld*) ;
layout === " " / CC"C" / CC"L" / CC"P" / CC"T" ;
u === "A"..."Z" ;
ud === "A"..."Z" / "0"..."9" ;
l === "a"..."z" ;
lid === "a"..."z" / "0"..."9" ;

END
DSL "M-Lisp-Semantics"

DOMAINS ! Syntactic

e : E = [S] / [X] / [Fn "['E ']"] / ["['C ']"] ;

c : C = [E "=" E] ;


f : F = Q ;

x : X = Q ;

s : S = ["(" S "," S ")"] ;

q : Q ;

DOMAINS ! Semantic

d : D = S / Funval ;

fv : Funval = S* -> S ;

r : Env = Q -> Env -> D ;

! Functions

ee := E -> Env -> S ;

ee-s := E* -> Env -> S* ;

cc-s := C* -> Env -> S ;

ff := Fn -> Env -> Funval ;

IN LAM e0.

LAM <lay: (Env -> X* -> S* -> Env),

<car, cdr, cons, atom, eq>: Funval*,

check-s:(D -> S),

check-f:(D -> Funval),

app: (((Env -> Funval), Env) -> Funval) ;
DEF ee(e)r : S =
CASE e
/[s] -> s
/[x] -> %check-s r(x)(r)
/[fn "[" e* "]"] -> ff(fn)r( ee-s(e*)r )
/["[" c* "]"] -> cc-s(c*)r
ESAC

WITH ee-s(e*)r : S* =
CASE e*
/ <> -> <>
/ e1 PRE e1* -> ee(e1)r PRE ee-s(e1*)r
ESAC

WITH cc-s(c*)r : S =
CASE c*
/ <> -> <>
/ [e1 "-->" e2] PRE c1* ->
CASE ee(e1)r
/ "NIL" -> cc-s(c1*)r
/ ? -> ee(e2)r
ESAC

ESAC
WITH \( \text{ff(fn,r} : 	ext{Funval} = \)

\[
\begin{align*}
\text{CASE fn} \\
/\text{"car"} & \rightarrow \text{car} \\
/\text{"cdr"} & \rightarrow \text{cdr} \\
/\text{"cons"} & \rightarrow \text{cons} \\
/\text{"atom"} & \rightarrow \text{atom} \\
/\text{"eq"} & \rightarrow \text{eq} \\
/\text{f} & \rightarrow \%\text{check-f } r(f)(r) \\
/\text{\"x* \";} e & \rightarrow \text{LAM s* e(e)(lay r x* s* )} \\
/\text{"label" f \";} fn & \rightarrow (\text{FIXLAM v:(Env \rightarrow Funval).} \\
& \quad \text{LAM r'. ff(fn)(r'\text{\"f<=v\")} %app (r)}
\end{align*}
\]

ESAC

IN \( \text{ee(e0)}(\text{LAM q, r}) \)

END
DSL "M-Lisp-Machine"

DOMAINS

Syntactic

e : E = [s] /
[X] /
[Fn "[" E* "]"] /
"[" C* "]" ;

Cases

c : C = [E "=>" E] ;

Functions

fn : Fn = ["car"] /
["cdr"] /
["cons"] /
["atom"] /
["eq"] /
[F] /
["\" X* ";" E] /
["label" F ";" Fn] ;

Variables

f : F = Q ;

S-expressions

s : S = Q /
["(" S "." S ")"] ;

Quotations

q : Q ;

Semantic

d : D = S / Funval ;

Denoted values

fv : Funval = S* -> S ;

Environments

r : Env = Q -> Env -> D ;

Functions

lay := Env -> X* -> S* -> Env ;

car, cdr, cons, atom, eq := Funval ;

check-s := D -> S ;

check-f := D -> Funval ;

app := ((Env -> Funval), Env) -> Funval ;
DEF  lay r x* s* : Env =
  CASE <x*, s*>  
/ <<>, <>>  ->  r
/ <x1 PRE x1*, s1 PRE s1*>  ->  LET r1 = r \ x1 <= (LAM r'. s1)
  IN  lay r1 x1* s1*
  ESAC

LET  car<s> : S =
  CASE s  /
["(" s1 "." s2 ")"]  ->  s1
  ESAC

LET  cdr<s> : S =
  CASE s  /
["(" s1 "." s2 ")"]  ->  s2
  ESAC

LET  cons<s1,s2> : S =
["(" s1 "." s2 ")"]

LET  atom<s> : S =
  CASE s  
/ QUOTE ?  ->  "T"
/["(" s1 "." s2 ")"]  ->  "NIL"
  ESAC

LET  eq<s1,s2> : S =
  CASE <s1, s2>
/ <QUOTE ?, QUOTE ?>  ->  (s1 EQ s2  ->  "T", "NIL")
/ <?, ?>  ->  "NIL"
  ESAC
LET check-s d : S =
CASE d
/ QUOTE ?
/ ["(" s1 ")." s2 ")"] -> d
ESAC

LET check-f d : Funval =
CASE d
/ (LAM ?. ?) -> d
ESAC

LET app(v: (Env -> Funval), r) : Env =
v(r)

IN < lay,
<car, cdr, cons, atom, eq>,
check-s,
check-f,
app >

END
DSL    "Pretty"

DOMAINS

s :  S = Q /
    {"(" S "." S ")"} ;

o :  D = Q /
    Q* ;

DEF pretty(s) ; O =

CASE s

/ QUOTE ? -> s
/
/ {"(" s1 "." s2 ")"} ->

    LET o1 = pretty(s1)
    ALSO o2 = pretty(s2)
    IN o2 IS "NIL" -> <o1>,
    o2 IS ?* -> o1 PRE o2, <o1,o2>

ESAC

IN pretty

END
REFERENCES

ASPE

V. Donzeau-Gouge, G. Kahn, B. Lang:

"A Complete Machine-Checked Definition of a Simple Programming Language using Denotational Semantics".


Available from:

IRIA
B.P.105
F-78150 Le Chesnay
France.

BASIC

Jens Dohn, Karsten Staer:

"BASIC Semantics".


(N.B. About 100 pages long -- only recommended to those intending to write a BASIC semantics themselves!)