PM

ISSN 0105-8525

SIS — *Semantics Implementation System*

Tested Examples

by
Peter Mosses

DAIMI MD-33
August 1979

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

ISSN 0105-8525

SIS — *Semantics Implementation System*

Tested Examples

by
Peter Mosses

DAIMI MD-33
August 1979

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

FOREWORD

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

Lambda-Calculus with Atoms

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:

```
(\ 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.

```
GRAM    "LC-Parser"

SYNTAX

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

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
```

```
DSL     "LC-Semantics"

DOMAINS
! Syntactic:
exp  :  Exp   =   ["\" Ide "." Exp]  /  [Exp Exp]  /
                  Ide  /  Num  /  ["(" Exp ")"]  ;
ide  :  Ide   =   ["IDE" Q]  ;
num  :  Num   =   ["NUM" N]  ;

! Semantic:
e   :     E   =    [N]  /  [F]  ;
f   :     F   =    E -> E  ;
n   :     N  ;
q   :     Q  ;
r   :     R   =    Ide -> 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 -> N -> 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
```

LOOP

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:

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

Example Data:

```
LAMB "Data"

<3>

END
```

Result of Interpreting:

```
LAMB "LOOP-Semantics(Program)(Data)"

< 6>

END
```

Degree of Testedness: High.

```
GRAM "LOOP-Parser"                                                    ! 01

SYNTAX                                                                ! 02

prog ::=        read-cmd  ";"  cmd-seq  ";"  write-cmd  ;             ! 03

read-cmd  ::=   "READ"  var*-","       : ["READ" var*]  ;             ! 04

write-cmd  ::=  "WRITE"  exp+-","      : ["WRITE" exp+]  ;            ! 05

cmd-seq  ::=    cmd-seq  ";"  cmd      : [cmd-seq ";" cmd]  /         ! 06
                cmd                    : cmd  ;                       ! 07

cmd  ::=        var  ":="  exp  /                                     ! 08
                "TO"  exp  "DO"  cmd  /                               ! 09
                "("  cmd-seq  ")"      : cmd-seq  ;                   ! 10

exp  ::=        exp  add-op  exp-a  /                                 ! 11
                exp-a                  : exp-a  ;                     ! 12

add-op  ===     "+"  /  "-"  ;                                        ! 13

exp-a  ::=      exp-a  mult-op  exp-b  /                              ! 14
                exp-b                  : exp-b  ;                     ! 15

mult-op  ===    "*"  /  "/"  ;                                        ! 16

exp-b  ::=      var  /                                                ! 17
                num  ;                                                ! 18

var  ::=        "VAR"  q               : q  ;                         ! 19

num  ::=        "NUM"  n               : n  ;                         ! 20



DOMAINS                                                               ! 21

cmd-seq, cmd    :  Cmd;                                               ! 22

exp, exp-a, exp-b       :  Exp  ;                                     ! 23

add-op, mult-op         :  Op  ;                                      ! 24
```

```
LEXIS                                                                    ! 25

program  ::=    word+              : CONC word+  ;                       ! 26

word  ::=       var                : <OUT"VAR", var>  /                  ! 27
                num                : <OUT"NUM", num>  /                  ! 28
                comment            : <>  /                               ! 29
                layout+            : <>  ;                               ! 30

var  ::=        letter  letter-digit*                                    ! 31
                                   : QUOTE(letter PRE letter-digit*)  ;  ! 32

letter  ===     "a"..."z"  ;                                             ! 33

letter-digit  ===  "a"..."z"  /  "0"..."9"  ;                            ! 34

num  ::=        digit+             : NUMBER digit+  ;                    ! 35

digit  ===      "0"..."9"  ;                                             ! 36

comment  ::=    "C"  "M"  "T"  comment-char*    : ?  ;                   ! 37

comment-char  =\=  ";"  ;                                                ! 38

layout  ===     " "  /  CC"C"  /  CC"L"  /  CC"T"  ;                     ! 39


END                                                                      ! 40
```

```
DSL "LOOP-Semantics"

!       The "direct" style of semantics is used, to enable comparison
!       with Tennent's semantics for LOOP [CACM 19:8].

!       Expressions cannot have side-effects in LOOP.  As there are no
!       declarations in LOOP, environments are not used in the semantics


DOMAINS

!       SYNTACTIC:

prog  :         Prog  =          [Read-cmd ";" Cmd ";" Write-cmd]  ;

read-cmd:       Read-cmd  =      ["READ" Var*]  ;

write-cmd  :    Write-cmd  =     ["WRITE" Exp+]  ;

cmd  :          Cmd  =           [Cmd ";" Cmd]  /  [Var ":=" Exp]  /
                                 ["TO" Exp "DO" Cmd]  /  ["(" Cmd ")"]  ;

exp  :          Exp  =           [Exp Op Exp]  /  [Var]  /  [Num]  ;

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

var  :          Var  =           Q  ;

num  :          Num  =           N  ;


!       SEMANTIC:

s  :    S  =    Var -> N  ;      ! States
n  :    N  ;                     ! Numbers
q  :    Q  ;                     ! Quotations


!       FUNCTIONS:

pp  :=          Prog -> N* -> N+  ;

cc  :=          Cmd -> S -> S  ;

ee-list  :=     Exp+ -> S -> N+  ;

ee  :=          Exp -> S -> N  ;

oo  :=          Op -> <N,N> -> N  ;

repeat  :=      N -> (S -> S) -> S  ;

update-list  := <Var*,N*> -> S -> S  ;

initial-s  :=   S  ;

update  :=      <Var,N> -> S -> S  ;
```

```
DEF     pp[read-cmd ";" cmd ";" write-cmd](n*): N+  =

        LET  ["READ" var*] = read-cmd
        ALSO ["WRITE" exp+] = write-cmd
        LET  s1 = update-list(var*,n*)(initial-s)
        LET  s2 = cc(cmd)(s1)
        IN   ee-list(exp+)(s2)


WITH    cc(cmd0)(s): S  =

CASE cmd0

        /[cmd1 ";" cmd2] ->      cc(cmd2)( cc(cmd1)(s) )

        /[var ":=" exp] ->       LET  n = ee(exp)(s)
                                 IN   update(var,n)(s)

        /["TO" exp "DO" cmd] ->  LET  n = ee(exp)(s)
                                 IN   repeat(n)( cc(cmd) )(s)

        /["(" cmd ")"] ->        cc(cmd)(s)

ESAC


WITH    ee-list(exp0+)(s): N+  =

CASE exp0+

        /<exp> ->                <ee(exp)(s)>

        /exp PRE exp+ ->         ee(exp)(s) PRE ee-list(exp+)(s)

ESAC


WITH    ee(exp0)(s): N  =

CASE exp0

        /[exp1 op exp2] ->       LET  n1 = ee(exp1)(s)
                                 ALSO n2 = ee(exp2)(s)
                                 IN   oo(op)(n1,n2)

        /[var] ->                content(var)(s)

        /[num] ->                num : N

ESAC
```

```
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
```

PL

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 Ide 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
```

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.

```
GRAM    "PL-Parser"

SYNTAX

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

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

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

ide  ::=         "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:

cmd  :  Cmd  =   [Cmd ";" Cmd]  /
                 ["BEGIN" Cmd "END"]  /
                 ["BEG" Dec "IN" Cmd "END"]  /
                 [Ide ":=" Exp]  /
                 ["IF" Exp "DO" Cmd]  /
                 ["WHILE" Exp "DO" Cmd]  /
                 ["BREAK"]  /
                 ["WRITE" Exp]  ;

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

exp  :  Exp  =   [Exp "->" Exp "," Exp]  /
                 [Exp Op Exp]  /
                 ["TRUE"]  /
                 ["FALSE"]  /
                 [Num]  /
                 ["-" Num]  /
                 ["READ"]  /
                 [Ide]  ;

ide  :  Ide  =   Q  ;

num  :  Num  =   N  ;

op   :  Op   =   "&"  /  "/"  /  "<"  /  "="  /
                 "+"  /  "-"  /  "*"  ;

!       SEMANTIC:

a  :    A  ;                          ! Answers
c  :    C  =    S -> A  ;             ! command Continuations
d  :    D  =    E  /  [C]  ;          ! Denoted values
e  :    E  =    V  /  [L]  ;          ! Expressed values
i  :    I  ;                          ! Inputs
k  :    K  =    E -> C  ;             ! expression Kontinuations
l  :    L  ;                          ! Locations          — not needed for
n  :    N  ;                          ! Natural numbers      this segment
q  :    Q  ;                          ! Quotations
r  :    R  =    Ide -> D  ;           ! envIRonments
s  :    S  ;                          ! States
t  :    T  ;                          ! Truths
v  :    V  =    [N]  /  ["-" N]  /  [T]  ;       ! storable Values
x  :    X  =    R -> C  ;             ! declaration Xontinuations


!       FUNCTIONS:

cc   :=  Cmd -> R -> C -> C  ;

dd   :=  Dec -> R -> X -> C  ;

ee   :=  Exp -> R -> K -> C  ;

IN
```

```
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

/       ["BEG" dec "IN" cmd "END"] ->    dd(dec)r; LAM r'. cc(cmd)r'; c

/       [ide ":=" exp] ->                CASE    r(ide)
                                         /[l] -> ee(exp)r; LAM v.
                                                 update(l,v); c
                                         / 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

/       [dec1 ";" dec2] ->      dd(dec1)r; LAM r'. dd(dec2)r'; x

/       ["CON" ide "=" exp] ->  ee(exp)r; LAM v. x(r\ide<-v)

/       ["VAR" ide ":=" exp] -> ee(exp)r; LAM v.
                                new-loc; LAM [l].
                                update(l,v); x(r\ide<-[l])

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[TT]

/       ["FALSE"] ->            k[FF]

/       [num] ->                LET n = num IN k[n]

/       ["-" num] ->            LET n = num IN k["-"n]

/       ["READ"] ->             read; k

/       [ide] ->                CASE    r(ide)
                                /[l] -> content(l); k
                                /[t] -> k[t]
                                /[n] -> k[n]
                                /["-"n] ->      k["-"n]
                                ESAC

ESAC


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

END
```

```
DSL     "PL-Machine"

DOMAINS
a  :    A  =    Q*  ;                 ! Answers
c  :    C  =    S -> A  ;             ! command Continuations
d  :    D  ;                          ! Denoted values
e  :    E  =    V  /  [L]  ;          ! Expressed values
i  :    I  =    N*  ;                 ! Inputs
k  :    K  =    E -> C  ;             ! expression Kontinuations
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]> ->     k[TT]
/       <"<",[n1],["-"n2]> ->     k[FF]

/       <"=",[n1],[n2]>
/       <"=",["-"n1],["-"n2]> ->LET t = n1 EQ n2 IN      k[t]
/       <"=",["-"n1],[n2]>
/       <"=",[n1],["-"n2]> ->     k[FF]

/       <"+",[n1],[n2]> ->        LET n = n1 PLUS n2 IN    k[n]
/       <"+",["-"n1],["-"n2]> ->LET n = n1 PLUS n2 IN    k["-"n]
/       <"+",["-"n1],[n2]> ->     op-val("-")([n2],[n1]); k
/       <"+",[n1],["-"n2]> ->     op-val("-")([n1],[n2]); 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]>
/       <"*",["-"n1],["-"n2]> ->LET n = n1 MULT n2 IN    k[n]
/       <"*",["-"n1],[n2]>
/       <"*",[n1],["-"n2]> ->     LET n = n1 MULT n2 IN
                                  n EQ 0 -> k[n], k["-"n]

/       ? ->                      wrong"? op-val"

ESAC
```

```
LET     init-r : R  =

        LAM ?. ?


LET     init-s(i) : S  =

        <LAM l. ?, 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 OK">)(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:

```
label[append;
\[[l1;l2];
  [eq[l1;NIL]-> l2;
   T          -> cons[car[l1];
                      append[cdr[l1];l2]]
  ]
 ]
]
[(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">>>>

END
```

Result of Applying Pretty:

```
LAMB "Pretty(M-Lisp-Semantics(Program)(M-Lisp-Machine))"

< "A", "B", "C", "D">

END
```

Degree of Testedness: Low.

```
GRAM    "M-Lisp-Parser"

SYNTAX

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

case  ::=       form  "->"  form  ;

var  ::=        ide  :          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  :         atom  /
                "("  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"  ;

ld  ===         "a"..."z"  /  "0"..."9"  ;

END
```

```
DSL     "M-Lisp-Semantics"

DOMAINS            ! Syntactic

e  :    E  =    [S]  /                    ! Forms
                [X]  /
                [Fn "[" E* "]"]  /
                ["[" C* "]"]  ;

c  :    C  =    [E "->" E]  ;             ! Cases

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

f  :    F  =    Q  ;                      ! Fn-identifiers

x  :    X  =    Q  ;                      ! Variables

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

q  :    Q  ;                              ! Quotations

DOMAINS            ! Semantic

d  :    D  =    S  /  Funval  ;           ! Denoted values

fv  :   Funval  =      S* -> S  ;

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

                   ! 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    ff(fn)r  : Funval =
        CASE fn
        /["car"] ->                 car
        /["cdr"] ->                 cdr
        /["cons"] ->                cons
        /["atom"] ->                atom
        /["eq"] ->                  eq
        /[f] ->                     %check-f  r(f)(r)
        /["\" x* ";" e] ->          LAM s*. ee(e)( lay r x* s* )
        /["label" f ";" fn] ->      ( FIXLAM v:(Env -> Funval).
                                          LAM r'. ff(fn)(r'\f<-v) ) %app (r)

        ESAC


IN      ee(e0)(LAM q. ?)

END
```

```
DSL      "M-Lisp-Machine"

DOMAINS            ! Syntactic

e  :    E  =    [S]  /                        ! Forms
                [X]  /
                [Fn "[" E* "]"]  /
                ["[" C* "]"]  ;

c  :    C  =    [E "->" E]  ;                 ! Cases

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

f  :    F  =    Q  ;                          ! Fn-identifiers

x  :    X  =    Q  ;                          ! Variables

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

q  :    Q  ;                                  ! Quotations

DOMAINS            ! Semantic

d  :    D  =    S  /  Funval  ;               ! Denoted values

fv  :   Funval  =      S* -> S  ;

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

                   ! 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  :    O  =    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

ASPLE

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

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

IRIA Rapport de Recherche 330 (1978).

Available from:

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

BASIC

Jens Dohn, Karsten Staer:

"BASIC Semantics".

DAIMI Internal Report 79-4-3, Aarhus (1979).

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

MD-33 SIS - TESTED EXAMPLES

TRYK: DAIMI/RECAU