PM

ISSN 0105-8525

SIS — *Semantics Implementation System*

Reference Manual and User Guide

by
Peter Mosses

DAIMI MD-30
August 1979

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

ISSN 0105-8525

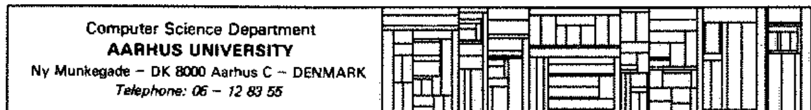
SIS – *Semantics Implementation System*

Reference Manual and User Guide

by
Peter Mosses

DAIMI MD-30
August 1979

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



ABSTRACT

The Semantics Implementation System, SIS, provides the following facilities:

- a parser-generator, producing parsing tables from grammars written in GRAM (an extension of BNF);
- a encoder-generator, producing "encoders" (code-generators) from semantic descriptions written in DSL (a variant of the Scott-Strachey notation for denotational semantics); and
- an interpreter, evaluating expressions in LAMB (a version of lambda-notation).

This document explains the general structure of SIS, and describes the notations GRAM, DSL and LAMB.

It is assumed that the reader is familiar with the method of denotational semantics, at least to the level of Tennent's tutorial paper [Comm.ACM,19:8].

SIS has been implemented in BCPL on a PDP-10. A hard copy of the implementation — which is reasonably portable — can be obtained by writing to the author.

ACKNOWLEDGEMENTS

SIS has been a long time under development, and people too numerous to list here have given advice and inspiration at various times. My thanks to them all — not least to the long-suffering students at Aarhus who experienced SIS' teething troubles!

The original inspiration for SIS came from the late Christopher Strachey. The encouragement and support from his Programming Research Group at Oxford were crucial to the initial development of SIS.

Karsten Bank Petersen, Aarhus, helped in rewriting parts of SIS to improve the portability. Gilles Kahn, IRIA, was brave enough to try out an early version of SIS, and his enthusiasm for SIS was a real help in completing what had turned into a rather long project.

The SIS project has been supported financially by the British Science Research Council, and by the Computer Science Department of Aarhus University.

CONTENTS

Abstract iii
Acknowledgements iv
Contents v

CHAPTERS

1. SIS 1

2. LAMB 5

   2.1. Domains 6
   2.2. Constants 7
   2.3. Identifiers 7
   2.4. Operators 7
   2.5. Enquiry 10
   2.6. Binding 12
   2.7. Miscellaneous 14

3. GRAM 17

   3.1. General 17
   3.2. Notes on Example 18
   3.3. Parsing 25

4. DSL 27

   4.1. General 27
   4.2. Notes on Example 29
   4.3. Cases 36
   4.4. Definitions 37
   4.5. Nodes 38
   4.6. Domains 39
   4.7. Type-checking 41

5. PRAGMATICS 43

5.1. LAMB 43
5.2. GRAM 45
5.3. DSL 47

References 50

APPENDICES

A. LAMB Syntax 53

B. GRAM Syntax 55

C. DSL Syntax 59

D. DSL Semantics 63

E. LAMB Reduction Rules 71

F. LAMB Evaluator 75

G. Loop Semantics (in LAMB) 87

1. SIS

SIS is basically a compiler-generating system. The part of it concerned with parsing is fairly conventional: it takes a context-free grammar, specified in an extension (called GRAM) of BNF, and produces a parsing table. This table can then be used to produce parse-trees (the parse-trees are more "abstract syntax trees" than derivation-trees) from programs in the specified language, as the first step towards compiling them. The parsing is usually split into two passes — lexical and syntax analysis — and use is made of the SLR(1) algorithm [Andersen,Eve&Horning73].

The rest of the system is less conventional. In contrast to other compiler-generating systems [Feldman&Gries68], the "encoder" (code-generator) part of a compiler is produced from an independently-useful formal semantics for the programming language. The type of formal semantics used by the system is so-called "denotational semantics" [Tennent76, Stoy77]. However, the original — rather exotic — notation of Scott and Strachey (used by Tennent and Stoy) is not very convenient for computer processing; so a variant of the notation, called DSL, has been devised (and formally defined, see Chapter 4). It is easy to translate Scott-Strachey notation into DSL, and vice versa.

The encoder, produced from the semantic description, takes the parse-tree of a program and gives what is basically an expression in lambda-notation [e.g.Stoy77]. The expression denotes the semantics of the program, usually a function from (a list of) inputs to (a list of) outputs. The particular version of lambda-notation used in SIS is called LAMB.

To run the code of a program with a particular input, the lambda-expression produced by the encoder is formally applied to a lambda-expression corresponding to the input; this application is then evaluated, i.e. reduced to "normal form", giving the output of the program. The reduction algorithm uses a "call-by-need" strategy [Vuillemin73] (due also to Chris Wadsworth).

The system is not closely tied to the use of the particular notations GRAM and DSL, which are described below. The user may define new notations, using the standard versions of GRAM and DSL.

The main components of SIS, which enable the above operations to be carried out, are as follows:

(P) The parser. It takes a source text and a parsing-table, and parses the text to produce a parse-tree.

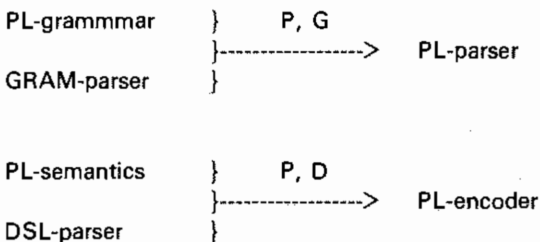
(G) The parser-generator. It takes the parse-tree of a grammar written in GRAM, and produces a parsing-table. (Standard parsing-tables are provided for LAMB, GRAM and DSL.)

(D) The encoder-generator. It takes the parse-tree of a semantic description written in DSL, and produces a LAMB-expression denoting the specified semantic function. When this LAMB-expression is applied to the parse-tree of a program, it produces a LAMB-expression denoting the semantics of the program (usually an "input-output" function).

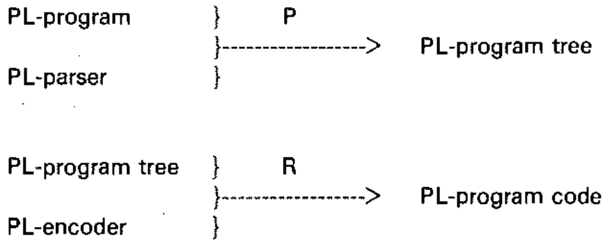
(R) The LAMB-reducer. It is used to evaluate applications of semantic functions to parse-trees of programs, and also applications of input-output functions to inputs. It can also be used for reducing arbitrary LAMB-expressions to "normal form".

The diagrams below illustrate the use of the main components of SIS. Let PL be some programming language.

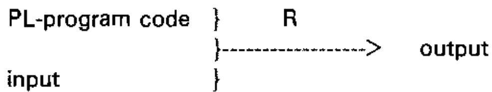
Compiler Generation:



Compilation:



Execution:



See [Mosses75,76] for more explanation of (and motivation for) the structure of SIS.

The following Chapters describe LAMB, GRAM and DSL in detail (albeit rather informally), and a final Chapter gives some practical advice on techniques which will enable the user to get the best out of SIS.

2. LAMB

This Chapter introduces LAMB, on which DSL and (to a lesser extent) GRAM are based. It is assumed that the reader is familiar with the lambda- notation used in denotational semantics (see [Tennent76, Stoy77]).

LAMB is a particular version of lambda-notation, formally based on Scott's LAMBDA [Mosses75, Scott76]. As such, it is a suitable notation, or "code", for representing abstract mathematical functions, such as semantic functions, or input-output functions. A concrete syntax for LAMB is given in Appendix A. The formal definition of the semantics of LAMB (by translation into LAMBDA) is not very illuminating, and therefore will not be given here; it is hoped that the informal description below will suffice for the general user of SIS.

LAMB-expressions satisfy certain "laws", which can be used as reduction rules for simplifying expressions. Note that, in contrast to the lambda-calculus, these laws are not 'a priori' axioms, but theorems provable from the semantics of LAMB. The reduction rules are given in Appendix E.

The LAMB-reducer in SIS is an algorithm for applying the reduction rules in a particular order. Thus LAMB-reduction of an expression produces a new expression denoting exactly the same value. For example, if a LAMB-expression f denoting a function is applied to another expression e, the LAMB-reducer can be used to "evaluate" the application f(e), giving some (hopefully simpler!) expression e'. The important point is that e' denotes exactly the same value as f(e)! In fact, it is only because of our limited ability to grasp the meaning of complex expressions, that LAMB-reduction is needed at all.

The reduction algorithm used in SIS is described in Appendix F. It uses a "call-by-need" strategy for applying the reduction rules.

The remaining sections of this Chapter give an informal description of the semantics of LAMB.

2.1. Domains

Let a "domain" be a complete partial order with a minimal element "bottom". For the use that is made of LAMB in SIS, it is unimportant whether a domain is taken to be a complete lattice, a chain-complete partial order, or whatever — all that is required is that solutions of domain equations and least fixed points of functions exist and are well-defined (up to isomorphism). Power domains are not provided in the current version of LAMB.

The meanings of LAMB-expressions belong to a domain E satisfying the following equation (up to isomorphism):

E = N + Q + T + E* + P + F + ?

where + is the coalesced sum, and

- N = non-negative integers (flat, i.e. no ordering between the proper elements)
- Q = so-called "quotations" (flat, countable)
- T = truth-values (flat)
- E* = finite tuples with components in E
- P = "parse-trees" with node-labels in Q and components in E
- F = E -> E, the continuous functions from E to E
- ? = the domain with just one proper element (which is also denoted by '?')

Note: E* = ? + E + E × E + ..., so the size of a tuple can be tested. This is in contrast to tuples in LAMBDA [Scott76], which are simply abbreviations for functions: they have no size, and cannot be concatenated.

In the following, the small letters n, q, t, p, f and e (possibly subscripted) will stand for arbitrary LAMB-expressions with meanings in the corresponding domains. A small letter followed by an asterisk '*' will represent an expression denoting a tuple with components in the indicated domain. Finally, the letter 'x' will stand for an arbitrary identifier of LAMB.

LAMB-expressions may be built up from constants, identifiers and operators, in accordance with the concrete syntax given in Appendix A. All syntactically-valid LAMB-expressions denote elements of E: the semantically-"nonsensical" expressions simply denote '?'.

2.2. Constants

The constants of LAMB consist of:

the decimal numerals 0, 1, ... , 9, 10, ...
denoting elements of N;

quoted sequences of characters, e.g. "I am a quotation", "!", ""
denoting elements of Q;

TT and FF
denoting elements of T;

and ?
denoting the proper element of the domain ?.

2.3. Identifiers

LAMB identifiers are basically sequences of small letters. However, the first letter may be a capital, and dashes '-' may be used. An identifier may be "decorated" with a subscript (a sequence of digits), or with one or more primes ('). (Also, the character '#' is used in LAMB-identifiers generated by SIS.)

2.4. Operators

Operators may be used as follows, to build up LAMB-expressions denoting more complex elements of E:

<e1, e2, ..., en>
denotes a tuple in E*, whose components are the values of e1,e2,...,en. The tuple may be empty, i.e. n=0. (Tuples with different numbers of elements are always distinct.)

SIZE e*
denotes the number of components of the tuple e*.

e* EL n
denotes the nth component of the tuple e*, provided that 1≤n≤SIZE e*.

e1* CAT e2*

denotes the concatenation of the tuples e1* and e2*.

e1* AUG e2

augments the tuple e1* with the value e2 (equivalently e1* CAT<e2>).

e1 PRE e2*

prefixes the value e1 to the tuple e2* (equivalently <e1>CAT e2*).

CONC e**

concatenates the tuple components of the tuple e**
(equivalently (e** EL 1) CAT ... CAT (e** EL SIZE e**)).

q NODE e*

denotes a "parse-tree" in P, whose label is q, and whose immediate branches are the components of the tuple e*.

LAM x. e

binds the identifier x in the expression e, and denotes the function in E —> E which takes the values of e as x varies over E. (See also Section 2.6.)

t —> e1, e2

is the conditional expression, equivalent to (i.e. denoting the same value as) e1 if t denotes true, equivalent to e2 if t denotes false.

Note that (all) the usual arithmetic and Boolean operators are verbalised in LAMB, to avoid confusion with the other uses of the symbols +,-,*,<,>.

n1 PLUS n2
n1 MINUS n2 (when n1≥n2)
n1 MULT n2
n1 DIV n2 (when n2>0)
n1 REM n2 (when n2>0)

all denote the obvious integer values.

n1 LS n2 "less"
n1 GR n2 "greater"
n1 LE n2 "less or equal"
n1 GE n2 "greater or equal"

are equivalent to TT if n1 is in the specified relation to n2
— otherwise equivalent to FF.

NOT t
t1 AND t2
t1 OR t2
all denote the obvious truth-values.

e1 EQ e2
is generally equivalent to TT if e1 and e2 denote the same value, and otherwise equivalent to FF. However:
— testing functional values always yields FF;
— testing 'bottom' always yields 'bottom'.

e1 NE e2
is the negation of e1 EQ e2.

The following operators can be used to construct "atomic" values (numbers, quotations, truths) from their character representations:

NUMBER q*
denotes the number whose decimal digits are the components of q*, provided that q* has at least one component.

QUOTE q*
denotes the quotation whose characters are the characters of the components of q*, provided that all these components denote single-character quotations. Note that QUOTE<> is equivalent to "". (Tuples of multi-character quotations may also be QUOTEd — distinct operand values yield distinct quotations.)

TRUTH q*
is equivalent to TT if q* is equivalent to <"T","T">, and to FF if q* is equivalent to <"F","F">.

CC q
denotes the quotation of a "special" character:
if q = "Q" then the quote-mark (");
if q = "C" then the carriage-return character;
if q = "L" then the line-feed (or new-line) character;
if q = "T" then (horizontal) tab;
if q = "P" then page-throw (form-feed); and
if q = "E" then end-of-file.

Function application is denoted by simple juxtaposition — but note that a space, or parentheses, may need to be inserted to separate the two operands (see Appendix A).

f e

denotes the value of the function f at the value e.

2.5. Enquiry

The "enquiry" operator

e1 IS e2

has a rather different nature from the operators described in the previous Sections. Its basic purpose in life is to tell whether the arbitrary value denoted by e1 is of the particular "form" described by e2 — e.g., whether e1 denotes a number, or a tuple, etc. More generally, it can be used to investigate structure to any depth, and resembles a "pattern-matching" operator (but without "back-tracking"). Constructor-operators for the particular forms are used in the "pattern"-expression e2, and '?' is used to match any value.

For example, the pattern to match any 3-tuple is <?, ?, ?>, hence:

e1 IS <?, ?, ?> is equivalent to

TT, if e1 denotes a 3-tuple;
FF, if e1 denotes some other tuple; also
FF, if e1 denotes a non-tuple — even if it is the special value '?' (but bottom, of course, if e1 denotes bottom).

There are pattern operators for matching each of the sub-domains of E (except for ?, which can however be tested for by the use of e1 EQ ?), and pattern expressions can be nested to any depth, in general.

The following operators may be used to build up pattern-expressions:

<e1, e2, ..., en>

matches tuples with n components.

e * (a pattern-expression followed by a star)

matches tuples with any number of components.

e +

matches tuples with at least one component.

e1 NODE e2

matches nodes.

LAM ?. ?

matches functions (note that this is only a 1-level pattern).

NUMBER e

matches numbers.

QUOTE e

matches quotations.

TRUTH e

matches truth-values.

CC e

matches special characters (newline, etc.).

Note that NUMBER, QUOTE, TRUTH and CC are the atom-constructor operators mentioned earlier. Hence one may specify the form of their operands. E.g.,

e1 IS QUOTE<?>

can be used to test whether e1 is the quotation of a single character. (Actually, this would also match, e.g., QUOTE<"ab"> — single character quotations c are exactly characterised by QUOTE<c> EQ c being equivalent to TT.)

Constants may be used in patterns, and (with the exception of '?') they simply match themselves.

For technico-pragmatic reasons, identifiers are permitted in pattern-expressions. However, in this context, they are completely equivalent to '?', matching any value. Therefore it is not possible to bind pattern-expressions to identifiers, and then use the identifiers in a pattern context. For example, LAM x.(e IS x) is absolutely equivalent to LAM x. TT .

To summarise the use of pattern-expressions with the operator 'IS':
e1 IS e2 is equivalent to:

TT, if the value denoted by e1 can be "constructed" by the pattern e2;
FF, if it cannot possibly be so constructed.

Pattern-expressions look basically the same as ordinary LAMB-expressions, but only "constructive" operators (NODE, NUMBER, etc.) may be used.

In effect, the notation 'e1 IS e2' is a generalisation of the "enquiry" notation used in [Tennent76, Stoy77] for separated sums, applied to the domain of values

E = N + Q + T + E* + P + F + ? .

2.6. Binding

If one has tested that a value is of a particular form, using e1 IS e2, it might then be desired to extract the (so-far untested) components of the value — perhaps for further testing. For example, if e1 IS <?*,?*> is true, then one might wish to extract the first and second components of e1, in order to test whether they are of the same size. LAMB follows ordinary lambda-notation in allowing tuples of identifiers in lambda-abstractions, to acheive this extraction. Thus x1 and x2 will be bound to the first and second components of e1

(LAM<x1,x2>. e)(e1) .

However, LAMB goes on to generalise this idea (after [Burstall69]): it allows not only tuples of identifiers, but also "pattern"-expressions (described in the previous Section) to occur in lambda-abstractions. If e' is any pattern-expression, then LAM e'. e is called a pattern-abstraction. Suppose e1 IS e' is equivalent to TT; then (LAM e'. e)(e1) denotes the value of e with the identifiers (if any) occurring in e' associated with the "corresponding" components of e1. E.g., if e1 IS QUOTE<?,?> is equivalent to TT, then (LAM QUOTE<x1,x2>. x1)(e1) denotes the first of the two characters of the quotation denoted by e1.

When a pattern-abstraction LAM e'. e is applied to a value e1 such that e1 IS e' is equivalent to FF, then the value denoted by the application is simply '?'. Hence, if e' is any pattern-expression, then LAM e'. e' denotes the function which is identity on the sub-domain of E corresponding to e', but which maps all other values to '?'.

The reader may have noticed that operators for selecting the labels and branches of nodes have not been introduced. This is because one can simply use the application

(LAM(x1 NODE x2). e1)(e2)

to select and bind the label (x1) and the tuple of branches (x2) of a node e2.

In a similar way, LAMB allows the tuple-prefixing operator 'PRE' to be used in pattern-expressions. This enables tuples to be regarded as lists, since the "head" and the "tail" of a tuple can be selected (and bound) by

(LAM(x1 PRE x2). e)(e') .

The nil-list is simply the empty tuple <>.

For symmetry, 'AUG' is also allowed in pattern-expressions. Note that 'CAT' is not allowed — it would introduce an unwelcome non-determinism into LAMB's semantics.

There are two forms of pattern-expression which do not always have an obvious meaning in a binding context (i.e. after 'LAM'). These are e * and e + (matching tuples of arbitrary and non-zero lengths). What identifier(s) should be bound in an application of LAM x*. e, for example? One could imagine some sophisticated scheme in which (LAM x*. e)(e1) would cause the creation and binding of "new" identifiers x-1, x-2, ..., x-n, where n would be the length of the tuple denoted by e1. LAMB steers clear of such a dynamic sort of binding, and takes a simple-minded view of LAM x*. e: x* is treated as a single identifier, and occurrences of x* in e refer to the whole (tuple) value bound to x* when the abstraction is applied. The same goes for x+ (for non-empty tuples), and for any identifier x followed by a sequence of * and/or + signs.

Pattern-expressions e *, e +, where e contains both identifiers and operators, can be used in binding contexts — they do NOT contribute to the binding, but do affect the pattern-matching. E.g.,

(LAM(QUOTE x)*. e)(e1)

is equivalent either to e or to ?, depending on the form of e1. (It would be "nicer" to use the equivalent

(LAM(QUOTE ?)*. e)(e1) .)

Finally, pattern-expressions may include the monadic operator VAL. This corresponds closely to "call-by-value" in Algol60 — semantically, it makes the enclosing operator ('LAM' or 'IS') "strict" (mapping bottom to bottom) in the qualified component. For example, LAM VAL x. e denotes a strict function, and <e1,e2> IS <?,VAL ?> will denote bottom if e2 denotes bottom. The main use of VAL in SIS is in achieving the desired termination semantics for programming languages.

(Note: Because it is possible for a LAMB-expression to have a non-bottom meaning, but not have a normal form, VAL-abstractions are "over-strict" in the present implementation. See Appendices E, F.)

There is one other binding operator in LAMB: the fixed-point operator, 'FIXLAM'. FIXLAM x. e is basically equivalent to y(LAM x. e), where y is the usual expression for the fixed-point operator:

LAM x1. (LAM x2. x1(x2(x2)))(LAM x2. x1(x2(x2))) .

As in LAM-abstractions, LAMB allows a pattern-expression in place of the bound identifier x. However, in FIXLAM e'. e, there is a minor restriction, in that e IS e' must be "manifestly" true. E.g. if e' is a tuple of identifiers, then e must also be a tuple-expression with the same number of components — it could not be of the form e1 CAT e2, even if e1 and e2 were tuple expressions with the correct (total) number of components.

2.7. Miscellaneous

The operators described above were mostly concerned with tuples and atoms, and with function abstraction. There are also operators providing function compositions commonly used in denotational semantics:

f1 ; e
f1 CIRC f2
f1 STAR f2 .

These operators can be explained simply in terms of LAM-abstraction and application:

f1 ; e = f1(e)

f1 CIRC f2 = LAM VAL x. (LAM VAL x1. f2(x1))(f1(x))

f1 STAR f2 =
LAM VAL x. (LAM VAL<x1,x2>. f2(x1)(x2))(f1(x)).

The semicolon operator ';' is useful, simply because it has a different precedence from the usual application operator (juxtaposition) — and it associates the opposite way, i.e. to the right (as do 'CIRC' and 'STAR', in contrast to all the other diadic operators). Thus

f1(f2(e))

can be written as

f1; f2; e

which facilitates the reading of large LAMB-expressions corresponding to continuation semantics for programs.

Note the order of composition for 'CIRC' and 'STAR', it is the (often more convenient) reverse of that for the usual circle and star operators. Note also that 'CIRC' and 'STAR' have been made "strict", anticipating their use in representing sequencing in DSL descriptions.

There are two remaining operations in LAMB:

SEG q

denotes the LAMB-expression residing on the "file" identified by q, thus facilitating the combination of independently-produced LAMB-expressions; and

ACTIVATE p

transforms the parse-tree of a LAMB-expression into the actual ("active") expression it denotes.

See the Pragmatics Chapter for further details.

So much for the meaning of the various constructs of LAMB. Of course, it is not claimed that the preceding informal description constitutes a complete definition of LAMB — though hopefully it is reasonably ambiguous. (One of the main points of incompleteness concerns the behaviour of the operators on '?'- and 'bottom'- operands: not all of them are strict.)

The Chapter concludes with a small example in LAMB. Further examples will be given in Chapter 5. (See also Appendix G for a larger example!)

Table 2.1

```
LAMB "Map-tot"

(LAM f.
 FIXLAM map-f.           ! applies f to all elements of n*
 LAM n*.
    SIZE n* EQ 0 -> <>,
    (LAM(n1 PRE n1*). f(n1) PRE map-f(n1*) )(n*)
)
(FIXLAM tot.
 LAM n.                  ! gives 0 + 1 + ... + n
    n EQ 0 -> 0,
    n PLUS tot(n MINUS 1)
)
<0,1,2,3,4,5,6,7,8,9>

END


LAMB "Map-tot"

< 0, 1, 3, 6, 10, 15, 21, 28, 36, 45>

END
```

3. GRAM

This Chapter presents GRAM, a notation for specifying syntax. GRAM has been designed to provide a transparent interface between concrete syntax (used for parsing) and abstract syntax (used in semantic descriptions). It is assumed that the reader is familiar with BNF [Naur63], and with the general concepts of context-free parsing.

3.1. General

GRAM is a formal notation, similar to BNF, for describing the context-free syntax of programming languages. SIS can take a syntactic description of a language, written in GRAM, and produce a parsing decision-table from it. The parsing algorithm of SIS can then interpret this table to parse programs in the described language, producing parse-trees conforming to a convenient abstract syntax. However, the grammar has to satisfy some constraints, corresponding roughly to the the SLR(1) condition [DeRemer71] — these constraints are described in Section **3.3**.

Usually, a description in GRAM consists of two parts: LEXIS and SYNTAX. This corresponds to parsing taking place in two successive passes. LEXIS describes the lexical analysis pass, which takes the source text (considered as a LAMB-tuple of single-character quotations) and recognises a sequence of "basic symbols", such as "reserved words", identifiers, numerals, strings, etc. The output of the lexical pass is a tuple formed from the recognised symbols, which are represented by LAMB-quotations (in general). This tuple is then input to the syntax analysis pass, described by SYNTAX, which parses the sequence of basic symbols to yield a parse-tree — composed of LAMB NODE-values.

It is possible to specify extra passes, to occur before or and after lexical analysis. Such a pass is called a TRANSFORM: it could, e.g., remove all layout characters, as for Algol60; or insert semicolons between certain combinations of basic symbols, for BCPL. (A TRANSFORM has the same structure as a LEXIS, and it will not be described further here.)

The same notation is used in GRAM for describing both lexical and syntactical analysis. The notation is basically BNF, but it allows explicit indication of the value to be yielded when an instance of an production is recognised. The LEXIS and SYNTAX parts differ in form only in that the values specified in LEXIS productions are generally quotations (or tuples of them), whereas those specified in SYNTAX productions are NODEs — and leaves — of parse-trees.

Apart for this explicit indication of values to be yielded, the main extensions of BNF in GRAM are "iterators" and "ranges". Iterators correspond to the Kleene-star, and ranges are a convenient way of specifying particular sets of terminal symbols.

For the concrete syntax of GRAM (in GRAM) see Appendix B. GRAM will now be described informally, with the help of the following example.

3.2. Notes on Example

Consider the example GRAM specification given in Table 3.1. The language described is a simple extension of LOOP (see [Tennent76]). The various features of GRAM will be explained with reference to the example, using '!n' to refer to the corresponding line. ('!' is used to introduce an "end-of-line" comment in GRAM descriptions.)

A GRAM specification starts with the symbol 'GRAM', followed by a string which is taken as the title !1. The SYNTAX !2 and LEXIS !25 are more or less of the same form: a sequence of "productions", each terminated by a semicolon ';'. The non-terminals are formed from lower-case letters and dashes ('exp', 'exp-a') whereas the terminal symbols of the grammar are quoted ("READ", ";") — the LAMB-notation for representing "control characters" is also allowed (CC"C" !39).

A production has a non-terminal to the left of '::=', and a list of "alternatives", separated by '/', to the right.

Table 3.1

```
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
```

Table 3.1 *(cont.)*

```
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
```

An alternative specifies a "phrase", consisting of a possibly-empty sequence of so-called "elements", which are usually simple "items", i.e. terminals or non-terminals. However, it is also possible for elements to be "iterators" !4 !5 !26 of the form:

item *

allowing zero or more occurrences of item;

item +

allowing one or more ...

item1 *- item2

allowing zero or more occurrences of item1, separated by occurrences of item2 (which is restricted to be a terminal); or

item1 +- item2

allowing one or more ...

Iterators allow the convenient expression of commonly-occurring constructs !4, and avoid the introduction of extra non-terminals. Of course, recursion (left !6, or right) may be used instead, if preferred. (Actually, the current implementation of iterators gives the same effect as using right-recursion, as regards the language accepted.)

For each alternative, the value to be produced when the phrase is recognised may be specified by a (restricted) LAMB-expression, following a colon ':' !4 !5 !6 !7. Identifiers (non-terminals) occurring in the expression refer to the values yielded by the recognition of the elements of the phrase. Tuple-identifiers, e.g. var* !4, exp+ !5, refer to the values yielded by the recognition of iterators — naturally enough, these values are always tuples. However, note that with a "separator" element, such as var*-"," !4, the tuple value has components corresponding only to the main item, here 'var'; furthermore, the separator '-","' is NOT used in the non-terminal referring to the tuple in the value expression.

Often, the identifiers occurring in the value-expression will be in the same order as the corresponding elements in the phrase which precedes it. (GRAM is designed for specifying simple-minded parsing, not for general syntactic translation.) However, there is no ambiguity when the same non-terminal occurs more than once in a phrase, e.g.

real ::= digit+ "." digit+ : QUOTE<digit+, digit+> ;

— the successive value-identifiers refer to the successive occurrences of non-terminals in the phrase, and subscripts on identifiers are neither needed nor allowed.

Value expressions may contain literal LAMB constants, i.e. numerals, strings, truths and '?'.

The only LAMB operators allowed in value-expressions are 'CAT', 'AUG' and 'PRE' (diadic) and 'NUMBER', 'QUOTE', 'CC' and 'CONC' (monadic). Tuples may be specified explicitly with the '<e1,...,en>'-notation. For specifying nodes of parse-trees, the DSL notation '[e1...en]' should be used !4 !5 !6. A full description of this notation is given in Chapter 4, the main idea is simply that '[e1...en]' specifies a node of a parse-tree, with a branch for each identifier 'ei'. The label of the node is formed partly from any literal strings occurring in the expression, partly from the identifiers occurring — more precisely, form their corresponding domain identifiers specified in the DOMAINS section of the grammar !21. For example, '[cmd-seq ";" cmd]' specifies a node with two branches identified by 'cmd-seq' and 'cmd', and with a label formed from "Cmd", ";" and "Cmd" (see !22). (The label is NOT simply the concatenation of the component strings — see Section 4.6 for details — so the direct use of the LAMB NODE operator is not recommended in GRAM.) For iterators, note that '["READ" var*]' !4 specifies a node with just one branch: the tuple identified by 'var*'.

In fact, it is seldom necessary to use the '[e1...en]' construction in GRAM descriptions. This is because there is a default convention in the SYNTAX part: if no value is specified explicitly, the "obvious" node is produced. Thus the value specifications in !4 !5 !6 — but not in !7 — are actually superfluous, and correspond to the implicit default values.

All this machinery enables one to obtain the desired "abstract syntax" with the minimum of effort. By using the DOMAINS to associate one domain-identifier with several non-terminals !22 !23 !24, one can cause the precedence information, present in the concrete syntax, to disappear from the abstract syntax. The reader should compare the abstract syntax in Table 4.1 with the concrete syntax in Table 3.1. Note that SYNTAX alternatives without explicit value specifications yield nodes with the same label (in the same abstract syntax domain) if and only if their phrases become identical on replacing non-terminals by their corresponding domain identifiers, and removing separators such as '-","' !4 !5. Note also that if there is no domain-identifier specified for a non-terminal, then one is provided automatically by putting the first letter of the identifier into upper case. E.g.

```
var: Var ;
```

is implicit in the example in Table 3.1.

Thanks to the above conventions, explicit value specifications can generally be omitted in SYNTAX. However, they are useful for inhibiting "chain-reduction" nodes, when alternatives have no significance for the abstract syntax. For example, the specification of the value 'cmd' in !7 (instead of the default '[cmd]') means that no node will be constructed when that alternative is recognised: the value yielded by the recognition of 'cmd' is simply passed along.

On the other hand, there is no default convention for value specifications in LEXIS. A glance at the variety of value specifications in the example (which is not atypical) will show why not.

Ranges were mentioned at the beginning of this Section. They are especially simple productions, capable of recognising only single terminal symbols !13 !16. The value yielded is always the symbol recognised, i.e. a quotation. Ranges are distinguished from productions by the use of '===' or '=\=' instead of '::=', after the non-terminal. Thus the range identified by 'add-op' !13 is equivalent to the production

add-op ::= "+" : "+" / "-" : "-" ;

(where the explicit specification of values is necessary, to avoid the default convention yielding the nodes '["+"]', '["-"]'). Apart from being a single terminal symbol, an alternative of a range can also be an interval !34 !36, consisting of two single-character quotations separated by three dots. In principle, the only meaningful intervals are "a"..."z", "A"..."Z", "0"..."9" and sub-intervals of these. Observe that the range identified by digit in the example !35 is exactly equivalent to

digit === "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ; .

When the sign '=\=' is used instead of '===' in a range, only terminals NOT in the specified intervals will be recognised. Again, the value yielded is the recognised symbol itself. Such ranges are particularly useful for describing the lexical analysis of strings and comments.

Finally, the special intervals 'QUOTE ?' and 'NUMBER ?' may be used (only) in ranges, to match arbitrary quotations and numbers. The range-identifiers 'q' and 'n' are pre-defined in GRAM, equivalent to specifying

q === QUOTE ? ;
n === NUMBER ? ;

See !19 !20 (and !27 !28) for examples of the use of 'q' and 'n'.

Perhaps the reader has noticed that the LEXIS of the example (Table 3.1) is rather small — it doesn't explicitly specify the recognition of the "ordinary" symbols, such as "READ", "WRITE", ";", which are used in SYNTAX. In fact, GRAM sees to this automatically: any literal string which occurs in a SYNTAX phrase, but which is not yielded by some LEXIS alternative, causes the addition of a suitable alternative to LEXIS. For example, the occurrence of "READ" in SYNTAX causes the automatic generation of an alternative

"R" "E" "A" "D" : "READ" /

in LEXIS.

Unfortunately, it is rather difficult to decide whether a quotation could be the result of an arbitrary value-expression, so the GRAM-user has to indicate explicitly just which literal strings are yielded by LEXIS. This done by preceding them with the "pseudo-operator" 'OUT', when they occur in value specifications in LEXIS !27 !28. 'OUT' has no other effect on values, it does NOT cause their "outputting"! In effect, 'OUT' prevents the generation of an extra alternative for the symbol it precedes — the symbol may then be safely used for "private communication" between the lexical and syntactical analysis, and does not get added to the language being described. In the example, the symbols

";" "READ" "," "WRITE" ":=" "TO" "DO"
"(" ")" "+" "-" "*" "/"

will be recognised by LEXIS, but not the symbols

"VAR" "NUM"

(which are not part of the language LOOP, and are used only as "markers" in the output of the lexical analysis pass).

Apart from this influence just described, the LEXIS and SYNTAX parts should be considered as specifying completely independent parsers, communicating only by the tuple of symbols produced by the lexical analysis. In particular, there is no interference between the names used for non-terminals in LEXIS and SYNTAX. Thus, in the example, the use of 'var' in both LEXIS and SYNTAX is purely coincidental, and does not contribute to the parsing process.

Appendix B gives the circular description of the concrete syntax of GRAM, and can serve as an additional example of the use of the various features of GRAM, as well as making precise some of the above informal comments about the form of the various constructs of LAMB.

3.3. Parsing

Finally, the problem of ambiguity should be faced. Completely unambiguous grammars are not very suitable for lexical anlysis: it happens quite often that some basics symbols are simply composed from others. A classic example is ':', '=' and ':=' in Algol60 — or identifiers 'a', 'b' and 'ab'. A suitable "disambiguating rule", adopted by GRAM, is that the longest possible symbol is always recognised — symbols continue until "stopped".

But consider the following example. Suppose a language has symbols 'REPEAT', 'REPEATUNTIL' and 'UNLESS' (BCPL). What should be the effect when the sequence of characters 'REPEATUNLESS' is met? Should it be an error, or should it be recognised as two symbols? This is not entirely a matter of esthetics, or "style" in language design: it affects the "power" of the parsing algorithm needed.

The parsing algorithm used in SIS is basically the SLR(1) algorithm described in [Andersen,Eve&Horning73], and thus has a one-symbol look-ahead but no "back-tracking" capability. It has been extended with the disambiguating rule mentioned above, and the net effect is that symbols such as 'REPEATUNLESS' will be treated as errors (although 'REPEAT UNLESS' would be OK).

No attempt will be made here to formalise the details of the semantics of GRAM in this (or, for that matter, in any other) respect. It is hoped that the meaning of GRAM is clear enough to enable the user to get a grammar "working" quickly — although it must be admitted that the required adherence to the SLR(1) condition can be tiresome at times. For some practical hints on writing SLR(1) grammars, see Chapter 5.

4. DSL

This Chapter describes DSL — Denotational Semantics Language — which is the semantic notation used with SIS. The reader is assumed to be familiar with LAMB (Chapter 2).

4.1. General

DSL is an extension of LAMB, in the direction of the so-called "Scott-Strachey notation" (SSN). It would have been nice to use SSN itself in SIS, so that the reader could be spared the details of yet another new notation; however, certain features of SSN make it rather unsuitable for computer processing. Among these features are: the lack of a formal definition of the notation; the many informally-described conventions, in particular those connected with "separated sums"; the use of the ellipsis (...) notation, which is very difficult to formalise; and the almost mandatory use of symbols and alphabets unavailable on (most) present hardware.

DSL is admittedly not as elegant or compact as SSN. However, it is hoped that it comes sufficiently close to the essence of SSN to make translation between the two notations quite easy. Programmers may even find comfort in using DSL, which has unashamedly "borrowed" features from such languages as Lisp, Gedanken and ISWIM. However, it should be stressed that DSL is a completely mathematical, non-imperative notation — there is no hidden dynamic state underlying its meaning.

LAMB is a sub-language of DSL, and was described in detail in Chapter 2. The remaining features of DSL are: the use of domain definitions, the form of function definitions, the 'CASE' construct, and the '[...]' notation for nodes (used also in GRAM, see Chapter 3).

Domain defintions in DSL have two purposes: they correspond to domain definitions in SSN, and they will aid the "type-checking" of DSL descriptions. Every identifier in a DSL description must have a domain-expression associated with it, either implicitly or explicitly. It is required that the domain information be sufficient to determine that all operations (including application) have type-compatible operands. This type-checking will catch most of the simple "bugs" in DSL descriptions.

A domain can be associated with an identifier when the latter is abstracted (defined), or, as in SSN, a domain can be "globally" associated with a whole family of identifiers, by means of the domain definition itself. For example, if 'n: N' occurs in the domain definitions, then this associates N with n, and also with any "decorated" version of n, such as n', n1, etc. — it also associates N* with the tuple identifiers n*, n'*, n1*, etc.

Function definitions in DSL are very similar to those in SSN. The parameters may be "Curried" in the usual way. Identifiers may also be defined to denote values other than functions, e.g. tuples and "atoms". Definitions may be combined either recursively, "simultaneously" or sequentially, to allow some control of scopes (lacking in SSN).

Recall that in SSN, a semantic function is generally defined by a group of "semantic equations". DSL does not try to make a special distinction between the defintions of semantic functions and other functions, and so it provides an explicit 'CASE'-selection construct: this can be used as the body of a function definition, and also in other contexts. A 'CASE' "works" in just the same way as the semantic equations do (insofar as the latter has ever been formalised [Scott76]).

An important feature of SSN is the use of the "syntactic brackets" (or quasi-quotes) '[...]'. Together with an abstract syntax specification, it allows a compact — and readable — description of functions defined on syntactic objects (e.g. semantic functions defined on programs) and avoids the need for verbose selector functions and predicates [McCarthy63]. DSL has more-or-less taken this feature directly from SSN. The notation in DSL is '[...]', where the items between '[' and ']' may be either identifiers or quoted strings (other literal LAMB constants are also allowed). Note, however, that '[...]' is an operator — it yields a LAMB node — whereas '[...]' is generally considered to be just a means for "insulating" the abstract syntax notation from the rest of SSN.

To avoid confusion with the '[...]' notation, DSL uses 'f\e1<—e2' instead of the SSN 'f[e2/e1]', for "perturbing" functions (usually representing environments and stores). Note the reversal of e1 and e2.

The formal definition of DSL consists of a concrete syntax (Appendix C) and a function giving a translation of DSL parse-trees into LAMB. A description of this function, written circularly in DSL, is given in Appendix D. However, this circular description is not put forward as the canonical definition of DSL. The canonical definition is a LAMB-expression corresponding to Appendix D. It is not included in this document, because, firstly, it is not easily comprehensible by itself; secondly, the user of SIS can produce it easily from the DSL version.

The remainder of this Chapter gives an example in DSL, explains the form of DSL desriptions with reference to the example, and gives an informal description of the semantics of the main features of DSL.

4.2. Notes on Example

The general form of DSL descriptions will now be described, with reference to the example in Table 4.1 (which matches the grammar in Table 3.1). '!n' will be used to refer to a particular line of the example. As in GRAM, '!' introduces an "end-of-line" comment.

A DSL description starts with the symbol 'DSL', followed by a string which is taken as the title of the description !1. The description finishes with the symbol 'END' !83. Usually, the body of the description will consist of some domain-definitions !6-!31, followed by a sequence of definitions of functions (or other values). The meaning of the whole description is given by a final expression !82, which is in the scope of all the preceding definitions.

In the domain definitions, there is a uniform treatment of syntactic (abstract syntax) domains and semantic domains. Domain identifiers begin with a capital letter ('Prog', 'S') whereas ordinary value-identifiers (sometimes referred to as "variables") begin with a small letter ('prog', 's'). Note that, to avoid confusion with the reserved words of DSL ('DSL', 'IN', etc.) capitals may not occur in the middle of identifiers. However, dashes '-' may be used, to aid readability ('Read-cmd'). Identifiers may be "decorated" with a subscript (to be a sequence of digits) or with a sequence of primes (') — or both.

Table 4.1

```
DSL "LOOP-Semantics"                                                    ! 01

!       The "direct" style of semantics is used, to enable comparison   ! 02
!       with Tennent's semantics for LOOP [CACM 19:8].                  ! 03

!       Expressions cannot have side-effects in LOOP.  As there are no  ! 04
!       declarations in LOOP, environments are not used in the semantics! 05


DOMAINS                                                                 ! 06

!       SYNTACTIC:                                                      ! 07

prog  :         Prog  =         [Read-cmd ";" Cmd ";" Write-cmd]  ;     ! 08

read-cmd:       Read-cmd  =     ["READ" Var*]  ;                        ! 09

write-cmd  :    Write-cmd  =    ["WRITE" Exp+]  ;                       ! 10

cmd  :          Cmd  =          [Cmd ";" Cmd]  /  [Var ":=" Exp]  /     ! 11
                                ["TO" Exp "DO" Cmd]  /  ["(" Cmd ")"]  ;! 12

exp  :          Exp  =          [Exp Op Exp]  /  [Var]  /  [Num]  ;     ! 13

op  :           Op  =           "+"  /  "-"  /  "*"  /  "/"  ;          ! 14

var  :          Var  =          Q  ;                                    ! 15

num  :          Num  =          N  ;                                    ! 16

!       SEMANTIC:                                                       ! 17

s  :    S  =    Var -> N  ;     ! States                                ! 18
n  :    N  ;                    ! Numbers                               ! 19
q  :    Q  ;                    ! Quotations                            ! 20

!       FUNCTIONS:                                                      ! 21

pp  :=          Prog -> N* -> N+  ;                                     ! 22

cc  :=          Cmd -> S -> S  ;                                        ! 23

ee-list  :=     Exp+ -> S -> N+  ;                                      ! 24

ee  :=          Exp -> S -> N  ;                                        ! 25

oo  :=          Op -> <N,N> -> N  ;                                     ! 26

repeat  :=      N -> (S -> S) -> S  ;                                   ! 27

update-list  := <Var*,N*> -> S -> S  ;                                  ! 28

initial-s  :=   S  ;                                                    ! 29

update  :=      <Var,N> -> S -> S  ;                                    ! 30
```

Table 4.1 *(cont.)*

```
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
```

Table 4.1 *(cont.)*

```
WITH    oo(op)(n1,n2): N  =                                                  ! 60

CASE op                                                                      ! 61

        /"+" ->          n1 PLUS n2                                          ! 62
        /"-" ->          n1 MINUS n2     ! gives ? if n2 greater than n1     ! 63
        /"*" ->          n1 MULT n2                                          ! 64
        /"/" ->          n1 DIV n2       ! gives ? if n2 is zero             ! 65
ESAC                                                                         ! 66


WITH    repeat(n)(c:(S -> S))(s): S  =                                       ! 67

        n EQ ? -> ?,                                                         ! 68
        n EQ 0 -> s,                                                         ! 69
        repeat(n MINUS 1)(c)( c(s) )                                         ! 70


WITH    update-list(var0*,n0*)(s): S  =                                      ! 71

        SIZE var0* EQ 0 -> s,                                                ! 72
        LET  var PRE var* = var0*                                            ! 73
        ALSO n PRE n* = n0*                                                  ! 74
        IN   update-list(var*,n*)( update(var,n)(s) )                        ! 75


WITH    initial-s : S  =                                                     ! 76

        LAM var. ?                                                           ! 77


WITH    update(var,n)(s): S  =                                               ! 78

        s \ var <- n                                                         ! 79


WITH    content(var)(s): N  =                                                ! 80

        s(var)                                                               ! 81

IN      pp :(Prog -> N* -> N+)                                               ! 82

END                                                                          ! 83
```

As in SSN, a domain definition can accomplish three things. Firstly, it can introduce a new domain-identifier. Secondly, it can specify, recursively with other domain definitions, the domain to be denoted by the identifier. Finally, it can associate the domain-identifier with a family of variables. In DSL — as in the DOMAINS part of GRAM — the specification of the family of variables comes first, and is followed by a colon ':'. (The variables should not be decorated here.) The specification of the domain is preceded by '=' — even when the domain is a "syntactic" one — and is terminated by a semicolon ';'. DSL provides standard domain-identifiers 'N', 'Q' and 'T' denoting the non-negative integers, the domain of LAMB-quotations, and the usual truth-values respectively !19 !20. Compound domains are formed with the aid of separated sums '/' !11-!14, products '<...>' !26 !28 !30 (also '+' and '*'), nodes '[...]' !8-!13 and functions '—>' !18 !22... . To allow the customary list of the types of the semantic functions, domain-identifiers may be omitted between the ':' and '=' !22-!31.

DSL is intended to be portable, and thus uses only a restricted character set. This rather restricts the choice of identifiers, compared to that in SSN [Milne&Strachey77]. Those who enjoy using Greek, Italic and Script alphabets (not mentioning Bold, Sans-serif and Gothic) are likely to feel frustrated in DSL. It is up to the reader to judge whether or not the rather strict conventions used in the example, such as double letters for semantic functions ('pp', 'ee-list'), and the close correspondence between the names of variables and their domains, are appropriate. The conventions are not mandatory, and may be varied (within the limits mentioned above) to suit personal tastes.

Moving on to the start of the function definitions !32, the usual form is a sequence of mutually recursive definitions, introduced by 'DEF' and separated by 'WITH'. The scope of the definitions is the expression following the matching 'IN' !82. The symbol 'IN' may in fact be omitted if the expression starts with more defintions. Note that

DEF d1 DEF d2 IN e

is not "as recursive" as

DEF d1 WITH d2 IN e

in that the scope of the definition d2 includes d1 in the latter form. Non-recursive definitions are introduced by 'LET' and separated by 'ALSO' !33 !34.

Each function definition specifies the domain of its result explicitly (after ':' !32 !38 etc.). The domains of the "Curried" parameters may be either implicit — using an association set up by a domain definition — or explicit, after ':' !67. In general in DSL, the domain of an expression or variable may be made explicit using ':' !58 !82. Although theoretically superfluous, such "assertions" of domains can do much to increase the comprehensibilty of complex semantic

descriptions (in this author's opinion).

Consider now the definition of 'cc' !39. This corresponds to a set of "semantic equations" in SSN, one for each of the alternatives !40 !41 !43 !45 of the 'CASE'-construct !39, which is terminated by 'ESAC' !46. The "test value" follows the symbol 'CASE' — it is usually one of the parameters of the function being defined. Each alternative starts with the symbol '/', and the following expression — terminated by '->' — gives the form of value which that alternative matches, in the same way that "pattern expressions" are used with the operator 'IS' in LAMB. The exact mechanism of the 'CASE'-construct will be explained later; the basic idea is that the first alternative whose pattern expression matches the test value is selected, the identifiers in the pattern expression are bound to the corresponding components of the test value, and the expression following the symbol '->' gives the value of the whole 'CASE'-construct. If no alternative matches the test value, the result is simply '?'.

There are two points to note about the "bodies" (following '->') of the alternatives. The first is that any DSL constructs may be used in them, e.g. nested 'CASE's, function definitions — in contrast to SSN. Secondly, the reader may have noticed that ordinary round parentheses '(', ')' have been used around the syntactic parameters in the example, e.g. cc(cmd1)(s) !40; whereas in SSN, the brackets '[', ']' would have been used. Aficionados of this feature of SSN — which can be helpful in a sea of round parentheses! — may be comforted to know that they may continue to "wrap up" syntax in square brackets '[',']', provided that they are consistent. The definition of 'cc' could have just as well been written as

```
    WITH  cc[cmd0](s) :S  =        ! N.B. [...]
    CASE  cmd0                     ! Not: CASE [cmd0]
    /     [cmd1 ";" cmd2] ->       cc[cmd2]( cc[cmd1](s) )
```

etc. — but note that the type of 'cc' is now

```
    cc :=  [Cmd] -> S -> S ;
```

where the domain-expression '[Cmd]' denotes the domain of parse-trees with the label "Cmd" and with a single branch in the domain identified by 'Cmd'. Putting it another way, everything will be OK if '[...]' in DSL is treated with just as much care as '<...>' (tupling) — for one does not expect '<e>' to be equivalent to 'e'.

The definition of 'pp' !32 is perhaps rather atypical — although it does resemble a semantic equation rather closely. The parameter of 'pp' is expected to be in the domain 'Prog', which means that it is a node with three branches, in the domains 'Read-cmd', 'Cmd' and 'Write-cmd'. This has been taken advantage of in the definition, by using the pattern expression

'[read-cmd ";" cmd ";" write-cmd]'

as a formal parameter. (Hopefully this no more mysterious than defining a function 'f(a,b)' expecting to be applied to a 2-tuple.) Such pattern expressions can also be used as left-hand-sides of ordinary (non-function) definitions !33 !34.

Moving on to the definition of 'ee-list' !47, the Lisp-programmer should start feeling more at home. Recall from the description of LAMB that 'e1 PRE e2*' prefixes the value e1 to the tuple e2*. When used in a "binding context", the operation is inverted, splitting a non-empty tuple into its "head" and "tail". (Note that the alternatives !49 !50 are mutually exclusive, thanks to the use of 'exp+' instead of 'exp*' in !50.)

The definition of 'update-list' !71 could also have been written like 'ee-list', using a 'CASE' instead of a conditional. Note that the value '?' will be given if the list of inputs 'n0*' is shorter than the list of variables 'var0*'.

The final three definitions !76 !78 !80 are those of the "primitive" auxiliary functions for handling states. Note that in the rest of the description, knowledge of the structure of the domain 'S' has not been used. It would have been possible — and in the author's opinion, preferable — to abstract away these definitions into a separate "segment", in attempt to introduce a smidgin of modularity into the semantic description. Details of how to do this will be given in Chapter 5, Pragmatics.

Before delving into the finer details of DSL, it should perhaps be mentioned that the aim with DSL has been to get as close to SSN as possible, so far as compatible with keeping DSL implementable. To a large extent, the development of DSL has been simply the formalisation of notations and conventions used by various authors in SSN — especially Tennent — with the help of some ideas of Burstall. The aim has not been to innovate (that is the next phase of the project, producing a version of DSL allowing high modularity in semantic descriptions). Thus it ought to be quite easy to transl(iter)ate SSN into DSL, and vice versa.

The only feature of SSN which may cause some difficulty, is the use of the ellipsis '...' convention, e.g.

```
ee[E1,...,En](r)(k) =
        ee[E1]r{λe1. ... ee[En]r{λen. k<e1,...,en>} ... }
```

— this must be completely reformulated in DSL, using a recursive list-evaluating semantic function (like 'ee-list' !47). DSL has been forced to diverge from SSN here, simply because it seems unlikely that a reasonable formal definition of the '...' convention can be given — in spite of its frequent use in mathematics. (The author would welcome suggestions!)

Now for a more detailed description of the constructs of DSL.

4.3. Cases

The 'CASE'-construct is perhaps the easiest feature of DSL to explain in detail — assuming that the reader understands the use of "pattern expressions" (consistently represented by primed meta-variables below, e.g. e1') and the operators 'IS' and 'LAM' in LAMB.

Consider an arbitrary CASE-expression in DSL:

```
CASE e      / e1' --> e1
            / e2' --> e2
            ...
            / en' --> en ESAC
```

where the ei' are pattern expressions, as in LAMB (but generalised to include the DSL NODE-constructor '[...]') and e, e1, ..., en are ordinary value-expressions. This entire construct is exactly equivalent to the following LAMB-expression:

```
(e IS e1') -->      (LAM e1'. e1)(e),
(e IS e2') -->      (LAM e2'. e2)(e),
...,
(e IS en') -->      (LAM en'. en)(e), ? .
```

Note that if any pattern is simply '?' (or a simple identifier x) then e IS ei' will always be true. Therefore the alternative

```
        / ?  --> en
```

acts as a "catch-all" (default) alternative — obviously it is only sensible to use it as the last alternative in a case, as any alternatives following it could never be selected.

4.4. Definitions

Definitions are also quite simple to explain in detail. First of all, function definitions may be "desugared" into simple value-definitions by making the LAM-abstractions of the formal parameters explicit. A definition of the form

x(e1')...(en') :d = e

— where x is an identifier, the ei' are pattern expressions, d is a domain expression and e is a value expression — becomes

x :(d1 —>...—> dn —> d) = LAM e1'. ... LAM en'. e

where the domain expressions di are given (implicitly or explicitly) by the formal parameters ei'. For example,

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

becomes

content :(Var —> S —> N) = LAM var. LAM s. s(var)

(The domain expressions in definitions will be omitted in the rest of this section, as they are only of interest in relation to type-checking — see Section 4.7.)

In fact, non-function definitions (and de-sugared function definitions) in DSL are of the general form

e' = e

where e' may be any pattern expression. The nice thing about this form is that combinations of definitions can now be "collapsed":

e1' = e1 WITH ... WITH en' = en and

e1' = e1 ALSO ... ALSO en' = en

can be rewritten as

<e1', ... ,en'> = <e1, ... , en> .

Because tuples of pattern expressions are perfectly good pattern expressions, this produces a valid definition.

Now the only thing left to do to explain definitions, is to de-sugar

LET e' = e IN e0

DEF e' = e IN e0

into LAMB. This is simple:

LET e' = e IN e0 becomes (LAM e'. e0)(e)

whereas

DEF e' = e IN e0 becomes (LAM e'. e0)(FIXLAM e'. e) .

Domain definitions contribute only indirectly to the meaning of DSL, via the '[...]' notation (see the next Section). They do not have direct counterparts in LAMB. For a description of their effect on type-checking in DSL, see Section 4.7.

4.5. Nodes

The '[...]' notation will now be explained. The reader is warned that the "mechanism" may seem overly complicated at first sight; however it is difficult to find a simpler method of formalising something pragmatically close the the SSN usage of "abstract syntax".

Consider the expression

    [e1 ... en]

where e1, ..., en are either identifiers, strings or other literal constants. The expression is exactly equivalent to

    q NODE <e1', ..., em'>

where e1', ..., em' are the non-strings occurring in e1, ..., en (if any), and where q is defined as follows:

q = QUOTE <q1, ..., qn> where, for i = 1, ..., n,
  if ei is an identifier, then
    qi is the (string formed from the) associated domain identifier;
  if ei is a string, then
    qi is the same string; or
  if ei is some other literal constant, then
    qi is the corresponding domain name ("N", "T" or "?").

E.g. [cmd1 ";" cmd2] =
        QUOTE<"Cmd",";","Cmd"> NODE <cmd1,cmd2>
    [var] = QUOTE<"Var"> NODE <var>
    ["+"] = QUOTE<"+"> NODE <>   (= "+" NODE <>)
    [27] = QUOTE<"N"> NODE <27>   (= "N" NODE <27>)

Tuple identifiers, e.g. var*, exp+, may also be used — domain names are constructed accordingly. For example,

    ["READ" var*] =
        QUOTE<"READ", QUOTE<"Var","*">> NODE <var*> .

A useful rule-of-thumb is that the labels on '[...]' nodes will be the same, if and only if they look the same (up to layout) when the non-strings in the expression are replaced by the associated domain identifiers.

Warning for SIS users: when LAMB-NODEs are printed by the system, no distinction is made between 1-level- and multi-level-QUOTEs, i.e. QUOTE<"Cmd",";","Cmd"> will be printed the same as "Cmd;Cmd".

The '[...]' notation may be used wherever the LAMB NODE-operator is allowed, i.e. in value expressions, in patterns, and in domain expressions — which brings us to the final feature of DSL to be described here.

4.6. Domains

Recall that in SSN, domains are specified using separated sums, Cartesian products and functions. DSL allows all these, and in addition introduces notation for domains of nodes.

Let the meta-variables d, d1, ... stand for arbitrary domain expressions. Then the following are all domain expressions:

```
<d1, ..., dn>       — n-tuples
d *                 — any-tuples
d +                 — non-empties
[d1 ... dn]         — nodes
        (where the di are identifiers or literal constants)
d1 —> d2            — functions
d1 / ... / dn       — union.
```

Domain identifiers may also be used. Literal constants (numerals, strings, 'TT', 'FF' and '?') all denote domains whose only "proper" element is that constant. See also the concrete syntax in Appendix D.

However, in DSL it is also possible to consider separated sums to be "ordinary unions". The aim of this is to help the user who thinks in terms of manipulating individual values, and who wants to forget about the isomorphisms, injections and projections connected with separated sums and the solution of recursive domain equations.

DSL achieves this conceptual flexibility by not providing any notation for injections, projections and "enquiries" — it is also necessary to forbid "circular" sums such as D = A / D. Injections and projections may be considered to be inserted automatically, where necessary. (This is commonly assumed in SSN as a "convention" — it is formalised in the semantics of DSL.)

As for enquiries, i.e. tests for which (summand) domain a value is in, one has to "implement" them oneself! When the components of two summands are structurally distinguishable, one can use the CASE-construct (or the IS-operator): either to define a particular enquiry funtion, e.g.

```
DOMAINS ...
        d: D = F / <A,B> ;
        f: F = D —> D ;

LET is-f(d):T =
        CASE d      / (LAM?.?) —> TT
                    / ? —> FF
        ESAC
```

or else to combine enquiry with (projection and) selection of components:

```
        CASE d      / <a,b> —> ...a...b... ESAC .
```

When the summands are not structurally distinguishable (e.g. different function domains) or when it is too tedious to list all the cases, then the DSL '[...]'-notation can be used to "label" the summands differently, thus making them distinguishable. E.g.

```
        DOMAINS ...
                d: D = [L] / [V] ;
                l: L = N ;
                v: V = N / T ;
        ...
        LET is-v(d):T =
                CASE d      / [l] —> FF
                            / [v] —> TT
                ESAC
```

or,combining with projection,

```
        CASE d / [v] —> ...v... ESAC .
```

"Injection" of a value v in V into D must now be done explicitly, either by

```
        LET d = [v] IN ...d...
```

or by

```
        LET v-in-d(v):D = [v] IN ...v-in-d(v')... .
```

The advantage of the second form is that it can be used with values which are in subdomains of (here) V: v-in-d(n) gives the expected value in D, whereas using [n] would give a value with the label "N" rather than "V". (This is a consequence of the implicitness of the label in the '[...]' notation — the problem does not arise if the LAMB NODE-operator is used explicitly.)

Note that the use of nodes as components of sums is exactly what is wanted for the syntactic domains. There, the summands will be distinguishable by virtue of their different labels, e.g.

```
        DOMAINS ...
                cmd: Cmd = [Dec ";" Cmd] / [Cmd ";" Cmd] / ...;
        ...
        CASE cmd
        / [dec ";" cmd ] —> ...
        / [cmd1 ";" cmd2] —> ...
        ...
        ESAC
```

Apart from being a useful aide-de-memoire, the domain definitions have an important use in DSL: they enable SIS to do "type-checking" on DSL descriptions, as described in the following Section.

4.7. Type-checking

This facility is not implemented in the current version of SIS, so it will not be described in detail here. As in SSN, all operators in DSL (including application) must be given operands of the correct type, and definitions may only bind identifiers to values of the associated type. The type of an operand will be considered correct, if it is possible to arrive at the desired domain by a series of injections and projections between sums and summands.

Although LAMB is basically type-free, the reducer does catch and warn about such things as mismatches between actual parameters and formal patterns. Thus SIS does provide some protection against type-errors in DSL, albeit dynamically.

Note that the operator '@' in DSL is entirely concerned with the type-checking of so-called "polymorphic" functions. Basically, '@' is used to abstract a domain as parameter of a function definition, and then a domain is supplied each time the function is used. For example,

    map-list @ Z (f: (Z ->Z)) (z*: Z*) : Z* = ...

defines a general-purpose list-processing function, which can be used on a particular domain as follows:

    LET n* = map-list@N(LAM n. n PLUS 1)<1,2,3,4>

So that is (the present version of) DSL. Suggestions for improvements are very welcome!

5. PRAGMATICS

This Chapter attempts to point out some of the known inadequacies of SIS and gives some hints on how to get the best out of SIS. It is based mainly on the experiences of the author and the students at Aarhus. Hopefully, a reading of this Chapter, supplemented by a careful study of tested examples [Mosses79b], will help the new user of SIS to avoid some of the potential pitfalls! See also the Operating Notes [Mosses79a].

5.1. LAMB

Termination

A LAMB-expression without a normal form does not always correspond to the value "bottom" (in the domain of meanings of LAMB-expressions). It is to be expected that one will encounter "sensible" expressions without normal forms, their reductions consequently not terminating. Of course it is impossible for SIS to predict such non-terminations in general, so it is up to the user to diagnose the situation.

One aid for diagnosing the cause of non-termination of a reduction is to limit the number of reduction cycles and inspect the approximate normal forms produced. Often, the approximate normal forms will expand in a regular way when the number of cycles is increased. Note, however, that "direct recursions" give rise to constant approximate normal forms, e.g.

    (FIXLAM f. LAM n. n EQ 0 --> 1, n MULT f(n PLUS 1))(1)

has — eventually — the constant approximate normal form '?'. (If one is lucky, the replacement of unreduced sub-expressions by '?' when the cycles limit is reached may trigger a warning from the reducer, showing what it was in the process of reducing at that moment.)

FIXLAM

Fortunately, the presence of the FIXLAM operator in LAMB does allow recursively-defined LAMB functions and lists to have normal forms. E.g.

(*) FIXLAM f. LAM n. n EQ 0 --> 1, n MULT f(n PLUS 1)

and

    FIXLAM t. <0,t>

are both in normal form. This is in contrast to the pure lambda-calculus, where the expression corresponding to (*) above has no normal form, because of the explicit use of the lambda-expression for the fixed-point operator.

However, note that it is quite possible for FIXLAM x. e to be in normal form, but for the reduction of

LAM x'. (FIXLAM x. e)(x')

to not terminate! Roughly speaking, the reducer leaves a FIXLAM expression unexpanded until one tries to use it in some way — here, by applying it to a "dummy" argument. Then it is as if the FIXLAM "explodes" into an infinite expression!

The reduction algorithm is technically inadequate (unsafe) here, in that the above expression might have a normal form if the FIXLAM were to be expanded only once. Usually, this seems to be not the case, as recursions are generally "genuine", with the variable 'x' being used (e.g. applied) in the body of the FIXLAM. The current reduction algorithm is rather more efficient (on expressions with normal forms) than the safe version would be.

LAM VAL

If one wants to model the evaluation of strict functions, then one should use LAM VAL x. e instead of LAM x. e . This will cause non-termination when applying the abstraction to an expression with no normal form. Note, however, that this gives rather "over-strict" functions, unless one ensures that all arguments of abstractions have the value bottom whenever they have no normal form. (This is true of expressions denoting values in flat domains.)

Subscripts

Don't be surprised if "subscripts" of the form '#n' get appended to identifiers by the reducer. It is to guard against the capture of free variables during the simulated substitution in beta-reduction.

Identifiers of the from '##n' are generated by SIS, mostly during the translation of DSL into LAMB.

SEG

Although without interest from the point of view of reduction, SEG is pragmatically rather useful. It allows the easy "linking" of independently-produced LAMB-expressions (e.g. corresponding to separate DSL descriptions).

For a simple example, suppose one wishes to test a LAMB-expression by applying it to several other expressions. If "fun" refers to the file containing (the LAMB-code of) the main expression, then one can avoid editing and parsing it

for each test by using the expression

(SEG "fun")(...) .

SEG q is not substituted for by the parser, this is done dynamically during reduction.

Some of the SIS commands involving combination and reduction (Apply, Compile, Execute, Interpret) could be implemented by simply using Reduce on small files containing SEG-expressions. (Warning: on the current DEC-10 implementation, one cannot refer directly to files produced by the parser — only to "code" files. Also, the form of the quotation given as operand to SEG is implementation-dependent.)

ACTIVATE

This operator is in LAMB (and DSL) only to allow the circular semantics of DSL to be expressed. In effect, it represents the semantic function for LAMB, taking trees representing LAMB-expressions and producing the expressions themselves. For cognoscenti only !

5.2. GRAM

SLR(1)

The author was originally persuaded by [Andersen,Eve&Horning73] that it was reasonable to impose the SLR(1) condition on grammars: "Amending a grammar to enable the use of the more restrictive [than LR(1)] SLR(1) algorithm ... is at worst a small additional burden which can be treated in conjunction with the problem of eliminating genuine ambiguities." Experience with SIS has indicated that the SLR(1) restriction is actually rather annoying — for example the grammar for DSL is not SLR(1), and it would probably take substantial distortion to make it satisfy the condition.

Ambiguities

Luckily GRAM has some features which help to minimise the annoyance of the SLR(1) restriction. The most important of these is the automatic resolution of most ambiguities ("genuine" or not). If there is a choice between recognising one alternative and continuing to scan another alternative, then latter action is taken. This gives the effect of recognising the (locally) longest instances possible, e.g. the "dangling 'else'" ambiguity is resolved as usual, giving

[IF exp THEN [IF exp THEN cmd1 ELSE cmd2]]

rather than

[IF exp THEN [IF exp THEN cmd1] ELSE cmd2] .

This automatic resolution of ambiguities is also invaluable in lexical analysis.

It is also possible for ambiguities between recognising two different alternatives to occur. The alternative which comes first in the grammar text is preferred.

If one is in doubt about the choices which are made to resolve ambiguities, then one can get SIS to write out the resulting parsing-table. The non-SLR(1) states are distinguished by comments in the table.

Watch out for alternatives which become "masked" by the resolution of ambiguities — SIS unfortunately gives no warning about this.

Note that using left-recursion instead of right-recursion can help to make a grammar SLR(1). See the definition of 'cmd-seq' in Loop, Table 3.1.

TRANSFORM

One can use a TRANSFORM pass before the LEXIS pass (i.e. last in the GRAM description) to do character conversion and removal of layout, for example. A TRANSFORM pass between LEXIS and SYNTAX could be used to handle things like BCPL's convention for insertion of semicolon and other symbols.

Abstract Syntax

The abstract syntax trees produced by the parser can be adjusted by a judicious use of the DOMAINS section and of the value-fields in the alternatives. The former allows syntactic categories to be combined, thus removing (typically) precedence information from the tree; the latter allows the elimination of redundant "chain-reduction" nodes. It is advisable to have a close look at some trees produced by the parser, to see whether they conform to expectations. Note that there is no automatic check that the sytactic domains specified in a DSL description match the trees produced by the corresponding GRAM grammar.

The abstract syntax should be chosen to make the definition of the semantics as natural as possible. The choices made in the examples (such as Loop) are not necessarily the best!

5.3. DSL

Typechecking

In the absence of the implementation of typechecking, careful hand-checking of the domains of operands in DSL descriptions is essential. The use of conventions linking value-identifiers to domain-identifiers — as used in the Loop description, Table 4.1 — helps by factoring the checking into two phases: first to check that the right-hand-sides of all definitions yield values in the domain associated with the identifier being defined; then to check all the uses of identifiers in expressions.

Experience has shown that some caution is necessary with the '[...]' notation. If using it to separate the summands of semantic domains, remember that the node labels depend entirely on the domain names associated with the variables used between '[' and ']': there are no automatic coercions between domains in this context.

A useful way of tracking down missing (or wrong) alternatives CASE-constructs is to use something like

    / ? —> ?"suitable message"

as the last alternative. If this alternative is selected during reduction, the warning from the reducer will include the quotation.

Segmentation

It is advisable to keep the size of DSL descriptions small by splitting larger ones into segments (of, say, 5 pages or less). Often, it is quite natural to factor a denotational semantics into parts such as static semantics (typechecking, normalisation), dynamic semantics (the main semantic functions) and the auxiliary functions (the storage model, etc.). These parts can then be combined using the various SIS-commands, or with the aid of the SEG-operator — see the description of this under LAMB above. Not only does this facilitate editing and parsing, it also allows a more systematic testing of the semantics: one can inspect the value produced by the static semantics before beginning to test the dynamic semantics, etc. Because of the abstraction of the auxiliary functions, it is much easier to read the "code" produced by the "compiler".

By the way, one should not have any free variables in DSL segments — so a warning from the reducer about free variables should be taken seriously, it usually indicates a spelling mistake in an identifier, or a missing definition (hereunder the mistaken use of 'LET' instead of 'DEF').

Termination

Unfortunately it is quite possible for the semantics of a particular program (i.e. some LAMB-expression denoting an input-output function) to have no normal form — even for a program that will always terminate when executed.

In this case one is unable to inspect the code of the program, all one can do is to provide the data for the program "at compile-time", thus getting the output of the program instead of its code. This clearly corresponds to interpreting the program rather than compiling it. Incidentally, this is the reason that SIS is called a Semantics Implementation System rather than a compiler-generating system — apart from the niceness of the acronym, that is!

One can guard against the lack of a normal form for the semantics of a program in some cases. Basically, the trick is to make sure that recursively-defined functions do not get applied until they can "evaluate out". Consider the Loop semantics, Table 4.1. There the (recursive) function 'repeat' is applied to 'n', which is the result of evaluating an expression, and not in general known at "compile-time". This makes it unsuitable for use as a "compiler": it should be used as an "interpreter" instead. The easiest way to enable it to be used as a compiler is to abstract 'repeat' (and maybe the other auxiliary functions) into a separate segment. The body of the main semantics segment would then start with

LAM prog'.
LAM <repeat:(N —>(S —> S)—> S —> S),...>.

and finish with

IN pp(prog') : (N* —> N*) .

Using the Compile command, this segment could now be applied to a program tree, and the resulting code inspected. The code would start with

LAM <repeat,...>. LAM n*.

and should be applied both to the segment defining <repeat,...> and to the data using the Execute command, to obtain the output.

It would also be possible to reformulate the Loop semantics using CIRC and STAR, so that compiled code would always have a normal form. Similarly, with a continuation semantics the use of the operator ';' instead of application can help in avoiding the premature application of (FIXLAM expsressions and) recursively-defined functions. However, one should not let this feature of SIS influence the way one writes denotational semantics: the technique of abstraction is better, and has the beneficial side-effect of introducing some modularity into DSL.

For advice on localising the cause of the non-termination of a reduction, see the comments on LAMB above, under Termination. It may be useful to inspect the LAMB-expressions produced by SIS from DSL descriptions.

General

At least to start with, it should be easier to sketch a semantics in (one's favourite flavour of) SSN, and then translate it into DSL — rather than trying to formulate the description in DSL straight away.

It pays to take some care in designing a "complete" set of test-programs, to explore all the corners of the semantic description. Also, when a semantics is split into segments, it is as well to test the first segment before even typing in the others: thus getting feedback on one's understanding of the abstract syntax — and of DSL!

That is all the advice that I can give at present on using SIS: the rest is up to you! Please send me any comments you may have about this Reference Manual and User Guide. Corrections can be published in the SIS Newsletter, and incorporated in any future reprinting of this document. Happy SISsing — and Good Luck!

REFERENCES

[Andersen,Eve&Horning73]
"Efficient LR(1) Parsers".
Acta Inf. **2**, 12-39 (1973).

[Burstall69]
"Proving Properties of Programs by Structural Induction".
Comp.J. **12**, 41-48 (1969).

[DeRemer71]
"Simple LR(k) Grammars".
Comm.ACM **14**, 453-460 (1971).

[Feldman&Gries68]
"Translator Writing Systems".
Comm.ACM **11**, 77-113 (1968).

[McCarthy63]
"Towards a Mathematical Science of Computation".
Information Processing 1962, 21-28, North-Holland, 1963.

[Milne&Strachey77]
"A Theory of Programming Language Semantics".
Chapman & Hall, 1977.

[Mosses75]
"Mathematical Semantics and Compiler Generation".
D.Phil. Thesis, Oxford Univ., 1975.

[Mosses76]
"Compiler Generation Using Denotational Semantics".
Proc.Symp. on Math.Found. of Comp.Sc., Gdansk, 1976;
Lect.Notes in Comp.Sc. **45**, 436-441, Springer-Verlag, 1976.

[Mosses79a]
"SIS - Semantics Implementation System - Operating Notes".
DAIMI MD-32, Comp.Sc.Dept., Aarhus Univ., 1979.

[Mosses79b]
"SIS - Semantics Implementation System - Tested Examples".
DAIMI MD-33, Comp.Sc.Dept., Aarhus Univ., 1979.

[Naur63]
"Revised Report on the Algorithmic Language Algol60".
Comm.ACM **6**, 1-17 (1963).

[Scott76]
"Data Types as Lattices".
SIAM J. Comp. **5**, 522-587 (1976).

[Stoy77]
"Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory".
The MIT Press, 1977.

[Tennent76]
"The Denotational Semantics of Programming Languages".
Comm.ACM **19**, 437-453 (1976).

[Vuillemin73]
"Correct and Optimal Implementations of Recursion in a Simple Programming Language".
IRIA-Laboria Rep.24 (1973).

A. LAMB Syntax

```
GRAM    "LAMB-Parser"

SYNTAX

segment ::=     "LAMB"  title  exp  "END"  ;

title  ::=      string  :       string  ;

exp  ::=        "LAM"  exp-a  "."  exp  /
                "FIXLAM"  exp-a  "."  exp  /
                exp-a  "->"  exp  ","  exp  /
                exp-a  seq-op  exp  /
                exp-a  :        exp-a  ;

exp-a  ::=      exp-a  di-op  exp-b  /
                exp-a  "NODE"  exp-b  /
                exp-a  "IS"  exp-b  /
                exp-b  :        exp-b  ;

exp-b  ::=      mon-op  exp-b  /
                exp-c  :        exp-c  ;

exp-c  ::=      exp-c  exp-d  /
                exp-d  :        exp-d  ;

exp-d  ::=      "("  exp  ")"  :        exp  /
                "<"  exp*-","  ">"  /
                exp-d  rep-op  /
                exp-ide  :      exp-ide  /
                atom  :         atom  ;

exp-ide  ::=    "ID"  q  /
                "ID"  q  "#"  number  ;

atom  ::=       number  :       number  /
                string  :       string  /
                "TT"  :         TT  /
                "FF"  :         FF  /
                "?"  :          ?  ;

number  ::=     "NM"  n  :      n  ;

string  ::=     "ST"  q  :      q  ;

seq-op  ===     ";"  /  "CIRC"  /  "STAR"  ;

di-op  ===      "AND"  /  "OR"  /  "EQ"  /  "NE"  /
                "LS"  /  "GR"  /  "LE"  /  "GE"  /
                "PLUS"  /  "MINUS"  /  "MULT"  /  "DIV"  /  "REM"  /
                "CAT"  /  "AUG"  /  "PRE"  /  "EL"  ;

mon-op  ===     "NOT"  /  "NUMBER"  /  "QUOTE"  /  "TRUTH"  /  "CONC"  /
                "CC"  /  "SIZE"  /  "VAL"  /  "SEG"  /  "ACTIVATE"  ;

rep-op  ===     "*"  /  "+"  ;
```

```
DOMAINS

segment, exp, exp-a, exp-b, exp-c, exp-d  :     Exp  ;
exp-ide  :       Ide  ;
seq-op  :        Di-op  ;
number  :        N  ;
string  :        Q  ;


LEXIS

segment ::=      word+  :        CONC word+  ;

word  ::=        bold-ident  :   <OUT "ID", bold-ident>  /
                 ident  :        <OUT "ID", ident>  /
                 ident decor  :  <OUT "ID", QUOTE<ident,decor>>  /
                 numeral  :      <OUT "NM", numeral>  /
                 string  :       <OUT "ST", string>  /
                 layout+  :      <>  /
                 "!"  comment*  :        <>  ;

bold-ident  ::= upper  lower-dash*  :   QUOTE(upper PRE lower-dash*)  ;

ident  ::=       lower  lower-dash*  :   QUOTE(lower PRE lower-dash*)  ;

decor  ::=       digit+  prime*  :       QUOTE(digit+ CAT prime*)  /
                 prime+  digit*  :       QUOTE(prime+ CAT digit*)  ;

numeral  ::=     digit+  :               NUMBER digit+  ;

string  ::=      CC"Q"  string-ch*  CC"Q"  :     QUOTE  string-ch*  ;

upper  ===       "A"..."Z"  ;

lower  ===       "a"..."z"  ;

lower-dash  === "a"..."z"  /  "-"  ;

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

prime  ===       "'"  ;

string-ch  =\=  CC"Q"  /  CC"C"  /  CC"L"  /  CC"P"  /  CC"T"  ;

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

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


END
```

B. GRAM Syntax

```
GRAM "GRAM"

SYNTAX

grammar ::=     "GRAM"  title  pass+  "END"  ;

title ::=       string  :                 string  ;

pass ::=        pass  "DOMAINS"  domain-dec+  /
                mode  prod-range+  ;

domain-dec ::=  ide+-","  ":"  ide  ";"  ;

mode ===        "SYNTAX"  /  "LEXIS"  /  "TRANSFORM"  ;

prod-range ::=  prod  :                   prod  /
                range  :                  range  ;

prod ::=        ide  "::="  altern+-"/"  ";"  ;

altern ::=      elem*  ":"  exp  /
                elem*  ;

elem ::=        item  sep-op  item  /
                item  rep-op  /
                item  :                   item  ;

item ::=        ide  /
                string  /
                control-char  ;

sep-op ===      "*-"  /  "+-"  ;

rep-op ===      "*"  /  "+"  ;

exp ::=         exp  di-op  exp-a  /
                exp  "NODE"  exp-a  /
                exp-a  :                  exp-a  ;

di-op ===       "CAT"  /  "AUG"  /  "PRE"  ;

exp-a ::=       mon-op  exp-a  /
                "OUT"  string  /
                exp-b  :                  exp-b  ;

mon-op ===      "NUMBER"  /  "QUOTE"  /  "CONC"  /  "CC"  ;
```

```
exp-b ::=         "("  exp  ")"  :          exp  /
                  "<"  exp*-","  ">"  /
                  "["  exp-c*  "]"  /
                  tup-ide  :                tup-ide  /
                  string  :                 string  /
                  numeral  :                numeral  /
                  "TT"  :                   TT  /
                  "FF"  :                   FF  /
                  "?"  :                    ?  ;

exp-c ::=         exp-b  ":"  tup-ide  /
                  tup-ide  :                tup-ide  /
                  string  :                 string  ;

tup-ide ::=       ide  rep-op  /
                  ide  ;

range ::=         ide  quality  spec+-"/"  ";"  ;

quality ===       "==="  /  "=\="  ;

spec ::=          string  "..."  string  /
                  string  /
                  control-char  ;

ide ::=           "ID"  q  :                q  ;

numeral ::=       "NM"  n  :                n  ;

string ::=        "ST"  q  :                q  ;

control-char ::="CC"  "ST"  q  :            (CC q)  ;
```

DOMAINS

```
exp, exp-a, exp-b, exp-c:        Exp  ;

tup-ide, ide:                    Ide  ;

string, control-char:            Term  ;
```

```
LEXIS

grammar ::=       word+  :                   (CONC word+)  ;

word ::=          layout-char+  :            <>  /
                  identifier  :              <OUT "ID", identifier>  /
                  numeral  :                 <OUT "NM", numeral>  /
                  string  :                  <OUT "ST", string>  /
                  "!"  comment-char*  :      <>  ;

layout-char === " "  /  CC"C"  /  CC"L"  /  CC"P"  /  CC"T"  ;

identifier ::=  letter  low-letter-dash*  :
                                             (QUOTE (letter PRE low-letter-dash*)) ;

letter ===        "A"..."Z"  /  "a"..."z"  ;

low-letter-dash ==="a"..."z"  /  "-"  ;

numeral ::=       digit+  :                  (NUMBER digit+)  ;

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

string ::=        CC"Q"  string-char*  CC"Q"  :
                                             (QUOTE string-char*)  ;

string-char =\= CC"Q"  /  CC"C"  /  CC"L"  /  CC"P"  /  CC"T"  ;

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

END
```

C. DSL Syntax

```
GRAM    "DSL-Parser"

SYNTAX

segment  ::=     "DSL"  title  exp  "END"  ;

title  ::=       string  :       string  ;

exp  ::=         defn-list+  "IN"  exp  /
                 "LAM"  exp-a  "."  exp  /
                 "FIXLAM"  exp-a  "."  exp  /
                 exp-a  "->"  exp  ","  exp  /
                 exp-a  "\"  exp  "<-"  exp  /
                 exp-a  seq-op  exp  /
                 exp-a  :        exp-a  ;

exp-a  ::=       exp-a  di-op  exp-b  /
                 exp-a  "&"  exp-ide  exp-b  /
                 exp-a  "NODE"  exp-b  /
                 exp-a  "IS"  exp-b  /
                 exp-a  ":"  dom-b  /
                 exp-b  :        exp-b  ;

exp-b  ::=       mon-op  exp-b  /
                 "&"  exp-ide  exp-b  /
                 exp-c  :        exp-c  ;

exp-c  ::=       exp-c  exp-d  /
                 exp-c  "@"  dom-b  /
                 exp-d  :        exp-d  ;

exp-d  ::=       "CASE"  exp-a  altern+  "ESAC"  /
                 "("  exp*-","  ")"  /
                 "<"  exp*-","  ">"  /
                 "["  exp-e*  "]"  /
                 exp-d  rep-op  /
                 exp-ide  :      exp-ide  /
                 atom  :         atom  ;

exp-e  ::=       exp-f  :        exp-f  /
                 atom  :         atom  ;

exp-f  ::=       exp-f  rep-op  /
                 exp-ide  :      exp-ide  ;

exp-ide  ::=     "ID"  q  /
                 "ID-DECOR"  q  q  ;

dom  ::=         dom-a+-"/"  ;

dom-a  ::=       dom-b  "->"  dom-a  /
                 dom-b  :        dom-b  ;
```

```
dom-b  ::=      "("  dom*-","  ")"  /
                "<"  dom*-","  ">"  /
                "["  dom-c*  "]"  /
                dom-b  rep-op  /
                dom-ide  :       dom-ide  /
                atom  :          atom  ;

dom-c  ::=      dom-d  :         dom-d  /
                atom  :          atom  ;

dom-d  ::=      dom-d  rep-op  /
                dom-ide  :       dom-ide  ;

dom-ide  ::=    "BOLD-ID"  q  ;

altern  ::=     "/"  exp-a+-"/"  "->"  exp  ;

defn-list  ::=  "DEF"  defn+-"WITH"  /
                "LET"  defn+-"ALSO"  /
                "DOMAINS"  dom-defn+  ;

defn  ::=       exp-a  "="  exp  /
                exp-ide  par+  ":"  dom  "="  exp  ;

par  ::=        "@"  dom-d  /
                ";"  exp-d  /
                exp-d  ;

dom-defn  ::=   exp-ide+-","  ":"  dom-ide  "="  dom  ";"  /
                exp-ide+-","  ":"  dom-ide  ";"  /
                exp-ide+-","  ":"  "="  dom  ";"  /
                dom-ide  "="  dom  ";"  ;

atom  ::=       number  :        number  /
                string  :        string  /
                "TT"  :          TT  /
                "FF"  :          FF  /
                "?"  :           ?  ;

number  ::=     "NM"  n  :       n  ;

string  ::=     "ST"  q  :       q  ;

seq-op  ===     ";"  /  "CIRC"  /  "STAR"  ;

di-op  ===      "AND"  /  "OR"  /  "EQ"  /  "NE"  /
                "LS"  /  "GR"  /  "LE"  /  "GE"  /
                "PLUS"  /  "MINUS"  /  "MULT"  /  "DIV"  /  "REM"  /
                "CAT"  /  "AUG"  /  "PRE"  /  "EL"  ;

mon-op  ===     "NOT"  /  "NUMBER"  /  "QUOTE"  /  "TRUTH"  /  "CONC"  /
                "CC"  /  "SIZE"  /  "VAL"  /  "SEG"  /  "ACTIVATE"  ;

rep-op  ===     "*"  /  "+"  ;
```

```
DOMAINS

segment,
exp, exp-a, exp-b, exp-c, exp-d, exp-e, exp-f,
dom, dom-a, dom-b, dom-c, dom-d  :                Exp  ;

exp-ide, dom-ide  :              Ide  ;

seq-op  :        Di-op  ;

number  :        N  ;

string  :        Q  ;


LEXIS

segment ::=      word+  :        CONC word+  ;

word  ::=        bold-ident  :   <OUT "BOLD-ID", bold-ident>  /
                 ident  :        <OUT "ID", ident>  /
                 ident decor  :  <OUT "ID-DECOR", ident, decor>  /
                 numeral  :      <OUT "NM", numeral>  /
                 string  :       <OUT "ST", string>  /
                 layout+  :      <>  /
                 "!"  comment*  :        <>  ;

bold-ident  ::= upper  lower-dash*  :   QUOTE(upper PRE lower-dash*)  ;

ident  ::=       lower  lower-dash*  :   QUOTE(lower PRE lower-dash*)  ;

decor  ::=       digit+  prime*  :       QUOTE(digit+ CAT prime*)  /
                 prime+  digit*  :       QUOTE(prime+ CAT digit*)  ;

numeral  ::=     digit+  :               NUMBER digit+  ;

string  ::=      CC"Q"  string-ch*  CC"Q"  :     QUOTE  string-ch*  ;

upper  ===       "A"..."Z"  ;

lower  ===       "a"..."z"  ;

lower-dash  ===  "a"..."z"  /  "-"  ;

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

prime  ===       "'"  ;

string-ch  =\=  CC"Q"  /  CC"C"  /  CC"L"  /  CC"P"  /  CC"T"  ;

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

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

END
```

D. DSL Semantics

```
DSL     "DSL-Semantics"

DOMAINS          ! Syntactic:

segment:Segment  =      ["DSL" Title Exp "END"]  ;

title:  Title  =        Q  ;

exp:    Exp  =          [Defn-list+ "IN" Exp]  /
                        ["LAM" Exp "." Exp]  /
                        ["FIXLAM" Exp "." Exp]  /
                        [Exp "->" Exp "," Exp]  /
                        [Exp "\" Exp "<-" Exp]  /
                        [Exp Di-op Exp]  /
                        [Exp "%" Ide Exp]  /
                        [Exp "NODE" Exp]  /
                        [Exp "IS" Exp]  /
                        [Exp ":" Exp]  /
                        [Mon-op Exp]  /
                        ["%" Ide Exp]  /
                        [Exp Exp]  /
                        [Exp "@" Exp]  /
                        ["CASE" Exp Altern+ "ESAC"]  /
                        ["(" Exp* ")"]  /
                        ["<" Exp* ">"]  /
                        ["[" Exp* "]"]  /
                        [Exp Rep-op]  /
                        [Exp+]  /
                        [Exp "->" Exp]  /
                        Ide  /  Atom  ;

ide:    Ide  =          ["ID" Q]  /
                        ["ID-DECOR" Q Q]  /
                        ["BOLD-ID" Q]  /
                        ["ID" Q "#" N]  ;       ! LAMB only

atom:   Atom  =         N  /  Q  /  T  /  ?  ;

defn-list:Defn-list  =  ["DEF" Defn+]  /
                        ["LET" Defn+]  /
                        ["DOMAINS" Dom-defn+]  ;

defn:   Defn  =         [Exp "=" Exp]  /
                        [Ide Par+ ":" Exp "=" Exp]  ;

par:    Par  =          ["@" Exp]  /
                        [";" Exp]  /
                        [Exp] ;

dom-defn:Dom-defn  =    [Ide+ ":" Ide "=" Exp ";"]  /
                        [Ide+ ":" Ide ";"]  /
                        [Ide+ ":" "=" Exp ";"]  /
                        [Ide "=" Exp ";"]  ;

altern: Altern  =       ["/" Exp+ "->" Exp]  ;
```

```
di-op:  Di-op  =        ";"  /  "CIRC"  /  "STAR"  /  "AND"  /  "OR"  /
                        "EQ"  /  "NE"  /  "LS"  /  "GR"  /  "LE"  /  "GE"  /
                        "PLUS"  /  "MINUS"  /  "MULT"  /  "DIV"  /  "REM"  /
                        "CAT"  /  "AUG"  /  "PRE"  /  "EL"  ;

mon-op: Mon-op  =       "NOT"  /  "NUMBER"  /  "QUOTE"  /  "TRUTH"  /
                        "CONC"  /  "CC"  /  "SEG"  /  "ACTIVATE"  /  "VAL"  ;

rep-op: Rep-op  =       "*"  /  "+"  ;


DOMAINS         ! Semantic:

n  :    N  ;                        ! numbers
q  :    Q  ;                        ! quotations
r  :    R  =    Q -> Q  ;           ! environments


DOMAINS         ! Functions:

ee  :=          Exp -> R -> Exp  ;

ee-s  :=        Exp* -> R -> Exp*  ;

ee-qs  :=       Exp* -> R -> Q*  ;

ee-bs  :=       Exp* -> R -> Exp*  ;

dd-lists  :=    Defn-list* -> Exp -> R ->  Exp  ;

round-map  :=   (Defn ->R ->Exp) -> Defn+ -> R -> Exp  ;

map  :=         (Defn ->R ->Exp) -> Defn* -> R -> Exp*  ;

dd-l  :=        Defn -> R -> Exp  ;

dd-r  :=        Defn -> R -> Exp  ;

pp-s  :=        Par* -> Exp -> R -> Exp  ;

dd-doms  :=     Dom-defn* -> R -> R  ;

dd-dom  :=      Dom-defn -> R -> R  ;

lay-ides  :=    R -> Ide* -> Ide ->R  ;

ii-q  :=        Ide -> Q  ;

aa-s  :=        Altern* -> R -> Exp -> Exp  ;

aa-es  :=       Exp* -> Exp -> R -> Exp -> Exp -> Exp  ;

private-ide  := Exp  ;


! meaning of description  :=    Exp -> Exp  ;


! Note: LAMB is a sub-domain of Exp.
!       To help the reader, identifiers denoting LAMB-expressions
!       (exp) are decorated with primes (') below.
```

```
DEF     ee(exp)r  : Exp  =

CASE    exp

    /["DSL" title exp1 "END"] ->LET exp1' = ee(exp1)r IN
                                ["LAMB" title exp1' "END"]

    /[defn-list+ "IN" exp1] ->  dd-lists(defn-list+)(exp1)r

    /["LAM" exp1 "." exp2] ->   LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                ["LAM" exp1' "." exp2']

    /["FIXLAM" exp1 "." exp2] ->LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                ["FIXLAM" exp1' "." exp2']

    /[exp1 "->" exp2 "," exp3]->LET (exp1',exp2',exp3') =
                                        ee-s(exp1,exp2,exp3)r
                                IN  [exp1' "->" exp2' "," exp3']

    /[exp1 "\" exp2 "<-" exp3]->LET (exp1',exp2',exp3') =
                                        ee-s(exp1,exp2,exp3)r
                                LET exp10' = private-ide
                                LET exp20' =
                                    LET  exp21' =   LET di-op = "EQ" IN
                                                    [exp10' di-op exp2']
                                    ALSO exp22' = exp3'
                                    ALSO exp23' = [exp1' exp10'] IN
                                    [exp21' "->" exp22' "," exp23']
                                IN ["LAM" exp10' "." exp20']

    /[exp1 di-op exp2] ->       LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                [exp1' di-op exp2']

    /[exp1 "%" ide exp2] ->     LET  exp10' = ee(ide)r
                                ALSO exp20'* = ee-s(exp1,exp2)r
                                LET  exp20' = ["<" exp20'* ">"] IN
                                [exp10' exp20']

    /[exp1 "NODE" exp2] ->      LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                [exp1' "NODE" exp2']

    /[exp1 "IS" exp2] ->        LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                [exp1' "IS" exp2']

    /[exp1 ":" exp2] ->         ee(exp1)r

    /[mon-op exp1] ->           LET exp1' = ee(exp1)r IN
                                [mon-op exp1']

    /["%" ide exp1] ->          LET  exp10' = ee(ide)r
                                ALSO exp20' = ee(exp1)r IN
                                [exp10' exp20']

    /[exp1 exp2] ->             LET (exp1',exp2') = ee-s(exp1,exp2)r IN
                                [exp1' exp2']

    /[exp1 "@" exp2] ->         ee(exp1)r
```

```
    /["CASE" exp1 altern+ "ESAC"] ->
                                LET exp1' = ee(exp1)r
                                LET exp11' = private-ide
                                LET exp2' = aa-s(altern+)r exp11'
                                LET exp10' = ["LAM" exp11' "." exp2'] IN
                                [exp10' exp1']

    /["(" exp* ")"] ->          LET exp'* = ee-s(exp*)r IN
                                SIZE exp'* EQ 1 -> exp'* EL 1,
                                ["<" exp'* ">"]

    /["<" exp* ">"] ->          LET exp'* = ee-s(exp*)r IN
                                ["<" exp'* ">"]

    /["[" exp* "]"] ->          LET  q* = ee-qs(exp*)r
                                ALSO exp'* = ee-bs(exp*)r
                                LET  exp1' = QUOTE q*
                                ALSO exp2' = ["<" exp'* ">"] IN
                                [exp1' "NODE" exp2']

    /[exp1 rep-op] ->           LET exp1' = ee(exp1)r IN
                                [exp1' rep-op]

    /["ID" q1] ->               ["ID" q1]

    /["ID-DECOR" q1 q2] ->      LET q' = QUOTE <q1,q2> IN

                                ["ID" q']

    / NUMBER ?
    / QUOTE  ?
    / TRUTH  ? ->               exp

    / ? ->                      ?

ESAC


WITH    ee-s(exp*)r  : Exp*  =

CASE    exp*

    / <> ->              <>

    / exp1 PRE exp1* -> ee(exp1)r PRE ee-s(exp1*)r

ESAC


WITH    ee-qs(exp*)r  : Q*  =

CASE    exp*

    / <> ->              <>

    / exp1 PRE exp1* -> ee-q(exp1)r PRE ee-qs(exp1*)r

ESAC
```

```
        WITH    ee-q(exp)r  : Q  =

        CASE    exp

            /[exp1 rep-op] ->   QUOTE <ee-q(exp1)r, rep-op>

            /["ID" q1]
            /["ID-DECOR" q1 q2] ->      LET q' = r(q1) IN
                                        q' EQ ? -> "?", q'

            / NUMBER ? ->               "N"

            / QUOTE  ? ->               exp

            / TRUTH  ? ->               "T"

            / ? ->                      "?"

        ESAC


WITH    ee-bs(exp*)r  : Exp*  =

CASE    exp*

    / <> ->             <>

    / QUOTE ? PRE exp1* ->      ee-bs(exp1*)r

    / exp1    PRE exp1* ->      ee(exp1)r PRE ee-bs(exp1*)r

ESAC


WITH    dd-lists(defn-list*)(exp)r  : Exp  =

CASE    defn-list*

    / <> ->             ee(exp)r

    /["DEF" defn+] PRE
        defn-list1* ->  LET  exp1' = round-map(dd-l)(defn+)r
                        ALSO exp2' = round-map(dd-r)(defn+)r
                        ALSO exp' = dd-lists(defn-list1*)(exp)r
                        LET  exp10' = ["LAM" exp1' "." exp']
                        ALSO exp20' = ["FIXLAM" exp1' "." exp2']
                        IN [exp10' exp20']

    /["LET" defn+] PRE
        defn-list1* ->  LET  exp1' = round-map(dd-l)(defn+)r
                        ALSO exp2' = round-map(dd-r)(defn+)r
                        ALSO exp' = dd-lists(defn-list1*)(exp)r
                        LET  exp10' = ["LAM" exp1' "." exp']
                        IN [exp10' exp2']

    /["DOMAINS" dom-defn+] PRE
        defn-list1* ->  LET r' = dd-doms(dom-defn+)r
                        IN dd-lists(defn-list1*)(exp)r'

ESAC
```

```
WITH    round-map(f :(Defn ->R ->Exp))(defn+)r  : Exp  =

        SIZE defn+ EQ 1 -> f(defn+ EL 1)r,
        LET exp'* = map(f)(defn+)r IN
        ["<" exp'* ">"]


WITH    map(f :(Defn ->R ->Exp))(defn*)r  : Exp*  =

CASE    defn*
    / <> ->      <>
    / defn1 PRE defn1* -> f(defn1)r PRE map(f)(defn1*)r
ESAC


WITH    dd-l(defn)r  : Exp  =

CASE    defn

    /[exp1 "=" exp2] ->          ee(exp1)r

    /[ide par+ ":" exp1 "=" exp2] ->    ee(ide)r

ESAC


WITH    dd-r(defn)r  : Exp  =

CASE    defn

    /[exp1 "=" exp2] ->          ee(exp2)r

    /[ide par+ ":" exp1 "=" exp2] -> pp-s(par+)(exp2)r

ESAC


WITH    pp-s(par*)(exp)r  : Exp  =

CASE    par*

    / <> ->                <>

    /["@" exp1] PRE par1* ->    pp-s(par1*)(exp)r

    /[";" exp1] PRE par1*
    /[exp1]     PRE par1* ->    LET  exp1' = ee(exp1)r
                                ALSO exp' = pp-s(par1*)(exp)r
                                IN ["LAM" exp1' "." exp']

ESAC
```

```
WITH    dd-doms(dom-defn*)r  : R  =

CASE    dom-defn*

    / <> ->              r

    / dom-defn1 PRE dom-defn1* ->
                         LET r' = dd-dom(dom-defn1)r IN
                         dd-doms(dom-defn1*)r'

ESAC


WITH    dd-dom(dom-defn)r  : R  =

CASE    dom-defn

    /[ide+ ":" ide "=" exp ";"]
    /[ide+ ":" ide ";"]          ->       lay-ides(r)(ide+)(ide)

    /[ide+ ":" "=" exp ";"]
    /[ide "=" exp]               ->       r

ESAC


        WITH    lay-ides(r)(ide*)(ide)  : R  =

        CASE    ide*

            / <> ->              r

            / ide1 PRE ide1* -> LET r' = r \ ii-q(ide1) <- ii-q(ide)
                                IN lay-ides(r')(ide1*)(ide)

        ESAC


        WITH    ii-q(ide)  : Q  =

        CASE    ide
            /["ID" q1]
            /["ID-DECOR" q1 q2]
            /["BOLD-ID" q1]      ->       q1
        ESAC
```

```
WITH    aa-s(altern*)(r)(exp')  : Exp  =

CASE    altern*

    / <> ->              ?

    /["/" exp+ "->" exp] PRE altern1* ->
                         LET exp'' = aa-s(altern1*)(r)(exp') IN
                         aa-es(exp+)(exp)(r)exp'exp''

ESAC


        WITH    aa-es(exp*)(exp2)(r)exp'exp''  : Exp  =

        CASE    exp*

            / <> ->     exp''

            / exp1 PRE exp1* ->
                         LET (exp1',exp2') = ee-s(exp1,exp2)r
                         LET exp3' = aa-es(exp1*)(exp2)(r)exp'exp''
                         LET  exp10' = [exp1' "IS" exp']
                         ALSO exp20' =
                             LET exp21' = ["LAM" exp1' "." exp2']
                             IN [exp21' exp']
                         IN [exp10' "->" exp20' "," exp3']

        ESAC


WITH    private-ide  : Exp  =

        LET q = "#" IN
        ["ID" q]


IN

        LAM exp. ACTIVATE ( ee(exp)(LAM q. ?) )

END     ! of "DSL-Semantics"
```

E. LAMB Reduction Rules

The reduction rules for LAMB are basically the beta-rule of the lambda-calculus together with rules for operators acting on tuples, nodes and constants. The rules concerned with pattern-expressions may be regarded as "explaining away" this feature of LAMB in terms of the other features.

As in Chapter 2, the small letters n, q, t, p, e and f (possibly subscripted) will stand for LAMB-expressions with meanings in the corresponding domains, a suffixed '*' indicating a tuple. The letter 'x' will stand for an arbitrary identifier. However, in describing the reduction rules, it is more appropriate to consider the syntactic form of an expression, rather than just its meaning. Therefore n, q and t below will be restricted to denote literal constants (numerals, strings "..." and 'TT', 'FF'); and p, f and e* will denote expressions of the form e NODE e', LAM e. e' and <e,...> respectively. The letter 'a' will stand for a constant (n,q,t) or '?'.

An occurrence of the left-hand-side of a reduction rule is called a "redex". An expression is said to be "in normal form" when it contains no redexes. (This definition is actually recursive, as some of the reduction rules impose the condition that particular components of the redex be already in normal form.) The predicate 'is-norm(e)' is to be true for exactly those expressions e which are in normal form.

N.B. The following reduction rules do NOT attempt to define LAMB: they are merely consistent with the semantics of LAMB.

```
TT -> e1, e2                          =>      e1
FF -> e1, e2                          =>      e2

t1 AND t2                             =>      conjunction of t1 and t2
t1 OR  t2                             =>      disjunction of t1 and t2

a1 EQ a2                              =>      TT when a1 identical to a2
                                                      (leading zeros ignored)
                                              FF otherwise

(q1 NODE e1*) EQ (q2 NODE e2*)        =>      (q1 EQ q2) AND (e1* EQ e2*)

(LAM e1'. e1) EQ (LAM e2'. e2)        =>      FF

<e1,...em> EQ <e1',...en'>            =>      (e1 EQ e1') AND...(em EQ en') when m=n
                                              FF otherwise

a EQ p          =>      p EQ a        =>      FF
a EQ f          =>      f EQ a        =>      FF
a EQ e*         =>      e* EQ a       =>      FF
p EQ f          =>      f EQ p        =>      FF
p EQ e*         =>      e* EQ p       =>      FF
f EQ e*         =>      e* EQ f       =>      FF

e1 NE e2                              =>      NOT (e1 EQ e2)

n1 LS n2                              =>      TT when n1 less than n2
                                      =>      FF otherwise
n1 GR n2                              =>      TT when n1 greater than n2
                                      =>      FF otherwise
n1 LE n2                              =>      TT when n1 less than or equal to n2
                                      =>      FF otherwise
n1 GE n2                              =>      TT when n1 greater than or equal to n2
                                      =>      FF otherwise

n1 PLUS n2                            =>      n1 plus n2

n1 MINUS n2                           =>      n1 minus n2
                                              when n1 greater than or equal to n2
                                      =>      ? otherwise

n1 MULT n2                            =>      n1 times n2

n1 DIV n2                             =>      n1 divided by n2 (with truncation)
                                                      when n2 greater than 0
                                      =>      ? otherwise

n1 REM n2                             =>      n1 modulo n2 when n2 greater than 0
                                      =>      ? otherwise

<e1,...,em> CAT <e1',...,en'>         =>      <e1,...,em,e1',...,en'>

<e1,...,em> AUG e'                    =>      <e1,...,em,e'>

e' PRE <e1,...,em>                    =>      <e',e1,...,em>

<e1,...,em> EL n                      =>      en when 0 less than n
                                                      and n less than or equal to m
                                      =>      ? otherwise
```

```
NOT t                                =>     negation of t

NUMBER <q1,...,qn>                   =>     the numeral formed from the characters
                                            of q1,...,qn when they are all digits
                                     =>     ? otherwise

QUOTE <q1,...,qn>                    =>     the quotation formed from q1,...,qn

TRUTH <"T","T">                      =>     TT
TRUTH <"F","F">                      =>     FF

CONC <e1*,...,em*>                   =>     e1* CAT...em*

CC "Q"                               =>     quote mark "
CC "C"                               =>     carriage-return
CC "L"                               =>     line-feed
CC "T"                               =>     horizontal tab
CC "P"                               =>     page-throw
CC "E"                               =>     end-of-file

SIZE <e1,...,en>                     =>     n

SEG q                                =>     expression in the file identified by q

ACTIVATE e                           =>     expression represented by tree e
                                                    when is-norm(e)

e IS ?                               =>     TT
e IS x                               =>     TT
e IS a                               =>     e EQ a when a is not '?'

(q NODE e*) IS (e1 NODE e2)          =>     (q IS e1) AND (e* IS e2)

(LAM e'. e) IS (LAM ?. ?)            =>     TT

<e1,...,em> IS <e1',...,en'>         =>     (e1 IS e1') AND...(em IS en') when m=n
                                     =>     FF otherwise

<e1,...,em> IS e' *                  =>     (e1 IS e') AND...(em IS e')
<> IS e' *                           =>     TT

<e1,...,em> IS e' +                  =>     (e1 IS e') AND...(em IS e') when m>0
<> IS e' +                           =>     FF

<e1,...,em> IS (e1' PRE e2')         =>     (e1 IS e1') AND (<e2,...,em> IS e2')
                                                    when m>0

<e1,...,em> IS (e1' AUG e2')         =>     (<e1,...,em-1> IS e1') AND (em IS e2')
                                                    when m>0

n IS (NUMBER e)                      =>     q* IS e when NUMBER q* => n
q IS (QUOTE e)                       =>     q* IS e when QUOTE q* => q
t IS (TRUTH e)                       =>     q* IS e when TRUTH q* => t
q IS (CC q)                          =>     q' IS e when CC q' => q

e IS (VAL e')                        =>     e IS e' when is-norm(e)

e IS e'                              =>     FF in all other cases
                                            (e.g., n IS (LAM ?. ?) => FF )
```

```
(LAM x. e)(e')                    =>     e with e' substituted for x
                                         (beta-reduction, avoiding "captures")

(LAM ?. e)(e')                    =>     e

(LAM a. e)(e')                    =>     (e' IS a) -> e, ?

(LAM(e1 NODE e2). e)(q NODE e*)   =>     (LAM e1. LAM e2. e)(q)(e*)

(LAM(LAM?.?). e)(LAM e1'. e2')    =>     e

(LAM<e1,...,em>. e)<e1,...,en'>   =>     (LAM e1...LAM em. e)(e1')...(en')
                                                  when m=n

(LAM e1 *. e)(e')                 =>     (e' IS e1 *) ->
                                                  e with e' subst. for e1 *, ?
(LAM e1 +. e)(e')                 =>     (e' IS e1 +) ->
                                                  e with e' subst. for e1 +, ?

(LAM(e1 PRE e2). e)<e1',...,em'>=>       (LAM e1. LAM e2. e)(e1')<e2',...,em'>
                                                  when m>0
(LAM(e1 AUG e2). e)<e1',...,em'>=>       (LAM e1. LAM e2. e)<e1',...,em-1'>(em')
                                                  when m>0

(LAM(NUMBER e1). e)(n)            =>     (LAM e1. e)(q*) when NUMBER q* => n
(LAM(QUOTE e1). e)(q)             =>     (LAM e1. e)(q*) when QUOTE q* => q
(LAM(TRUTH e1). e)(t)             =>     (LAM e1. e)(q*) when TRUTH q* => t
(LAM(CC e1). e)(q)                =>     (LAM e1. e)(q') when CC q' => q

(LAM(VAL e1). e)(e')              =>     (LAM e1. e)(e') when is-norm(e')

(e1; e2)(e')                      =>     (e1(e2))(e')

(e1 CIRC e2)(e')                  =>     (LAM VAL x.
                                           (LAM VAL x1.
                                                  e2(x1) )(e1(x)) ) (e')

(e1 STAR e2)(e')                  =>     (LAM VAL x.
                                           (LAM VAL<x1,x2>.
                                                  e2(x1)(x2) )(e1(x)) ) (e')

FIXLAM x. e                       =>     (LAM x. e)(FIXLAM x. e)
                                         only in the following contexts []:
                                                  [] -> e1, e2
                                                  [] di-op []
                                                  [] NODE []
                                                  [] IS e'
                                                  mon-op []
                                                  [](e')

FIXLAM e1. e2                     =>     (LAM e1. e2)(
                                           (LAM x'. e')(FIXLAM e1. e2) )
                                         where e' is such that
                                                  (LAM e1. e1)(e') => e'
                                         and      (LAM x. e')(e2) => e2
```

F. LAMB Evaluator

SIS evaluates LAMB-expressions by applying the reduction rules of Appendix E in a particular order. The algorithm used, called 'red', is based on the "call-by-need" or "lazy evaluator" strategy [Vuillemin73] (due also to Chris Wadsworth). As with a "call-by-name" (normal order) strategy, the leftmost outermost redex is reduced at each step; so in general it is a "safe" strategy, not embarking on the reduction of a sub-expression which may later be "thrown away". (Actually, it is slightly unsafe on FIXLAM x. e, wen x is not used in e -- see Chapter 5.) The inefficiency usually associated with call-by-name is avoided by ensuring that redexes are kept linked together, so that parameters of abstractions are not reduced more than once. E.g.

```
        (LAM x. x PLUS x)(1 PLUS 2)      =>*      6
```

in 3 steps (as in a "call-by-value" or "applicative order" strategy) as opposed to the 4 steps taken with call-by-name. Unfortunately red is not completely optimal, as redexes can be "hidden" from being kept linked together by abstractions:

```
        (LAM f. f(1) PLUS f(2))(LAM x. x PLUS (3 PLUS 4))        =>*      17
```

takes 8 steps, with (3 PLUS 4) being reduced to 7 twice. It seems that this is not a real source of inefficiency in practice.

Rather than iteratively searching for and reducing redexes, red uses recursion to keep track of what to reduce next. Substitution is simulated by the use of environments, associating identifiers with "closures" (pairs of expressions and environments). The call-by-need strategy is effected by updating the environment after reducing the closure associated with an identifier.

The following description corresponds quite closely to the actual implementation in the current version of SIS (1.1), but the details of the auxiliary functions are omitted, as the main purpose is to specify the order of reduction of redexes.

```
DSL "LAMB-Reducer"

! N.B. This description has not been tested.

DOMAINS

! LAMB syntax:

segment :       Segment = ["LAMB" Title Exp "END"] ;
title   :       Title   = Q ;
exp     :       Exp     = ["LAM" Exp "." Exp]
                        / ["FIXLAM" Exp "." Exp]
                        / [Exp "->" Exp "," Exp]
                        / [Exp Di-op Exp]
                        / [Exp "NODE" Exp]
                        / [Exp "IS" Exp]
                        / [Mon-op Exp]
                        / [Exp Exp]
                        / ["<" Exp* ">"]
                        / [Exp Rep-op]
                        / Ide
                        / N / Q / T / ?
                        / <Exp,Env> ;   ! created by red - not in LAMB
ide     :       Ide     = ["ID" Q]
                        / ["ID" Q "#" N] ;
di-op   :       Di-op   = ";" / "CIRC" / "STAR" / "AND" / "OR"
                        / "EQ" / "NE" / "LS" / "GR" / "LE" / "GE"
                        / "PLUS" / "MINUS" / "MULT" / "DIV" / "REM"
                        / "CAT" / "AUG" / "PRE" / "EL" ;
mon-op  :       Mon-op  = "NOT" / "NUMBER" / "QUOTE" / "TRUTH"
                        / "CONC" / "CC" / "SEG" / "ACTIVATE" / "VAL" ;
rep-op  :       Rep-op  = "*" / "+" ;

! Environments
env     :       Env     = <Env,Exp,Cell> / ? ;

! States
state   :       State   = <Mem,Cell,N,Files> ;
mem     :       Mem     = Cell -> Exp ;
cell    :       Cell    = N ;
files   :       Files   = Q -> Exp ;

! Continuations
c       :       C       = State -> Exp ;
k       :       K       = Exp -> C ;
x       :       X       = Env -> C ;
y       :       Y       = <A,Exp> -> C ;
a       :       A       = T / ? ;

! Standard
n       :       N ;     ! Natural numbers
q       :       Q ;     ! Quotations
t       :       T ;     ! Truth values

! Main functions
m       :       M       = "norm" / "part" ;     ! Mode of reduction for red
red             :=      M -> Exp -> K -> C ;
red-list        :=      M -> Exp -> K -> C ;
match           :=      Exp -> Exp -> Y -> C ;
match-list      :=      Exp -> Exp -> Y -> C ;
```

```
! Identifier Handling
! -------------------
! Fresh identifiers are supplied by  modify-ides , which uses the N-component
! of  State  to remember the highest subscript so far.

LET modify-ides(exp);k : C = ? ! (omitted expressions are represented by '?'.)
        ! gives a result of the same shape as  exp , but with all identifiers
        ! having fresh subscripts.  Modifies the N-component of  State .


! Environments
! ------------
! To achieve the call-by-need/lazy evaluator effect, environments are directly
! updatable, as well as extensible.  The Mem-component of  State  remembers the
! currents contents of cells, and the Cell-component points to a fresh cell.

LET void : Env = ?       ! (Here '?' is actually the value wanted!)
        ! EQ will be used to test whether an env is  void  or not,
        ! which is the reason that  Env  is not just  Exp -> Cell.

LET bind(env)(exp,exp');x : C = ?
        ! exp should be an identifier -- perhaps with subscript and rep-ops --
        ! or else an expression with the same shape as exp'.  env gets extended
        ! by binding all identifiers in exp to the corresponding components of
        ! exp', and the resulting environment is passed to the continuation  x.

LET find(env)(exp);y : C = ?
        ! If the identifier exp is not bound in env, then the continuation y
        ! is applied to the pair <FF,?> (:<A,Exp>);  otherwise, y is applied to
        ! <TT, the current contents of the cell associated with exp in env>.

LET rebind(env)(exp,exp');c : C = ?
        ! updates the cell associated with (identifier) exp in env to be exp'.

LET fix(env,env');c : C = ?
        ! env must be an extension of env', by cells containing pure
        ! expressions only.  All the extra associations in env' are updated
        ! to contain closures formed from the original expressions and env'.


! Suspensions
! -----------
! These are (in general) compound closures, i.e. expressions with environments
! attached to sub-expressions.  (The domain of suspensions could have been made
! distinct from that of pure expressions, but the '[...]' notation would then
! insist on a lot of renamings to get the labels right!)

LET sus-exp(exp) : Exp =
        CASE exp
        /<exp',env>      -> exp'
        / ?     -> exp
        ESAC

LET sus-env(exp) : Env =
        CASE exp
        /<exp',env>      -> env
        / ?     -> void
        ESAC
```

```
! Control
! -------
! Initialisation, finalisation, result-passing

LET set-up(exp)(c)(files) : Exp = ?
        ! applies c to initial state made from exp and files.

LET stop(exp)(state) : Exp = exp

LET res(exp);k : C = k(exp)

LET res-a(a);y : C = y(a)
        ! It would be "nicer" to make res polymorphic:
        ! LET res@Z(z:Z)(f:(Z -> C)) : C = f(z)
        ! but then one has to specify a domain every time, e.g. res@Exp(exp);k.
        ! "Overloading" is what is wanted, but it is not in DSL (yet).


! Abbreviations
! -------------

LET is-ide(exp) : T =
        CASE exp
        /["ID" q] / ["ID" q "#" n]        -> TT
        /[exp1 rep-op]              -> is-ide(exp1)
        / ?                -> FF
        ESAC

LET is-basic(exp)  : T =
        CASE exp
        /<exp',env>      -> is-basic(exp')
        /["LAM" exp1 "." exp2]
        /[exp1 "NODE" exp2]
        /["<" exp* ">"]
        / NUMBER ? / QUOTE ? / TRUTH ?            -> TT
        / ?                           -> (exp EQ ? -> TT,FF)
        ESAC

LET is-norm(exp) : T = ?
        ! true when no redexes in exp.  Actually implemented by "tagging"
        ! expressions with a bit indicating whether they have been reduced.

LET lam-query-query : Exp =     ? ! tree of 'LAM ?. ?'
LET number-query : Exp =        ? ! tree of 'NUMBER ?'
LET quote-query : Exp =         ? ! tree of 'QUOTE ?'
LET truth-query : Exp =         ? ! tree of 'TRUTH ?'
LET query-star : Exp =          ? ! tree of '? *'
LET query-star-star : Exp =     ? ! tree of '? * *'
LET quote-query-star : Exp =    ? ! tree of '(QUOTE ?) *'

LET circ : Exp =        ? ! tree of 'LAM ##1. LAM ##2. LAM ##3. ##2(##1(##3))'

LET star : Exp =        ? ! tree of 'LAM ##1. LAM ##2. LAM ##3.
                          !             (LAM<##4,##5>.##2(##4)(##5))(##1(##3))'
```

```
! Monadic operators
! ----------------

LET mon-op-arg(mon-op) : Exp =
        CASE mon-op
        / "NOT"          -> truth-query
        / "NUMBER"
        / "QUOTE"
        / "TRUTH"        -> quote-query-star
        / "CONC"         -> query-star-star
        / "CC" / "SEG"   -> quote-query
        ESAC

LET mon-op-fn[mon-op exp] : Exp = ?
        ! the result of applying the mon-op to exp

LET mon-op-inv(mon-op,exp) : Exp = ?
        ! the result of inverting the mon-op on exp


! Diadic operators
! ----------------

LET di-op-arg(di-op,n) : Exp =  ! n is 1 or 2
        CASE di-op
        / "AND" / "OR"              -> truth-query
        / "LS" / "GR" / "LE" / "GE"      -> number-query
        / "PLUS" / "MINUS" / "MULT"
        / "DIV" / "REM"                  -> number-query
        / "CAT"          -> query-star
        / "AUG"          -> n EQ 1 -> query-star, ?
        / "PRE"          -> n EQ 2 -> query-star, ?
        / "EL"           -> n EQ 1 -> query-star, number-query
        ESAC

LET di-op-fn[exp1 di-op exp2] : Exp =
        CASE di-op
        / ";"            -> [exp1 exp2]
        / "CIRC"         -> ? ! tree of 'circ(exp1)(exp2)'
        / "STAR"         -> ? ! tree of 'star(exp1)(exp2)'
        / ?              -> ? ! the result of applying the di-op to exp1, exp2
        ESAC

LET di-op-inv(di-op,n,exp) : Exp = ? ! n is 1 or 2
        ! the result of inverting the di-op on exp and taking the nth component


! IS-operator
! -----------

LET is-fn[exp1 "IS" exp2] : Exp = ?
        ! see the reduction rules


! Application
! -----------

LET app-fn[exp1 exp2] : Exp = ?
        ! see the reduction rules
```

```
! Reducer
! -------
! red(m)(exp0);k  reduces the expression (suspension) exp0 either to
! normal form, or to a basic form -- abstraction, tuple, node or atom.
! The parameter m specifies which ("norm" or "part").  If m is "part"
! but no basic form is found, the normal form is given as result.

DEF red(m)(exp0);k : C =

        LET (exp, env) = (sus-exp(exp0), sus-env(exp0)) IN

        is-norm(exp) AND (env EQ void) -> res(exp); k,

        CASE exp

        /<exp',env'> ->
                env EQ void -> red(m)<exp',env'>; k,
                env' EQ void -> red(m)<exp',env>; k, ?

        /["LAMB" title exp1 "END"] ->
                red(m)<exp1,env>; k

        /["LAM" exp1 "." exp2] ->
                m NE "norm" -> res<exp,env>; k,
                modify-ides(exp1); LAM exp1'.
                bind(env)(exp1,exp1'); LAM env'.
                red"norm"<exp2,env'>; LAM exp2'.
                res["LAM" exp1' "." exp2']; k

        /["FIXLAM" exp1 "." exp2] ->
                m NE "norm" ->
                        bind(env)(exp1,exp2); LAM env'.
                        fix(env,env');
                        red(m)<exp2,env'>; k,
                modify-ides(exp1); LAM exp1'.
                bind(env)(exp1,exp1'); LAM env'.
                red"norm"<exp2,env'>; LAM exp2'.
                res["FIXLAM" exp1' "." exp2']; k

        /[exp1 "->" exp2 "," exp3] ->
                match(truth-query)<exp1,env>; LAM<a,exp1'>.
                CASE a
                / TT -> CASE exp1'
                        / TT -> red(m)<exp2,env>; k
                        / FF -> red(m)<exp3,env>; k
                        ESAC
                / FF -> res(?); k
                / ? ->  red"norm"<exp2,env>; LAM exp2'.
                        red"norm"<exp3,env>; LAM exp3'.
                        res[exp1' "->" exp2' "," exp3']; k
                ESAC
```

```
/[exp1 di-op exp2] ->
        CASE di-op
        / ";" / "CIRC" / "STAR" ->
                LET (exp1', exp2') = (<exp1,env>, <exp2,env>) IN
                m NE "norm" -> red(m)(di-op-fn[exp1' di-op exp2']); k,
                red"norm"exp1'; LAM exp1''.
                red"norm"exp2'; LAM exp2''.
                res[exp1'' di-op exp2'']; k
        / "EQ" / "NE" ->
                red"part"<exp1,env>; LAM exp1'.
                red"part"<exp2,env>; LAM exp2'.
                is-basic(exp1') AND is-basic(exp2') ->
                        red(m)(di-op-fn[exp1' di-op exp2']); k,
                red"norm"exp1'; LAM exp1''.
                red"norm"exp2'; LAM exp2''.
                res[exp1'' di-op exp2'']; k
        / ? ->
                match(di-op-arg(di-op,1))<exp1,env>; LAM<a1,exp1'>.
                match(di-op-arg(di-op,2))<exp2,env>; LAM<a2,exp2'>.
                CASE (a1,a2)
                /(TT,TT) ->     red(m)(di-op-fn[exp1' di-op exp2']); k
                /(FF,? )
                /(?, FF) ->     res(?); k
                /(?, ? ) ->     red"norm"exp1'; LAM exp1''.
                                red"norm"exp2'; LAM exp2''.
                                res[exp1'' di-op exp2'']; k
                ESAC
        ESAC

/[exp1 "NODE" exp2] ->
        match(quote-query)<exp1,env>; LAM<a1,exp1'>.
        match(query-star)<exp2,env>; LAM<a2,exp2'>.
        CASE (a1,a2,m)
        /(TT,TT,"part") ->      res[exp1' "NODE" exp2']; k
        /(FF, ?, ?     )
        /(?, FF, ?     ) ->     res(?); k
        /(?, ?,  ?     ) ->     red"norm"exp1'; LAM exp1''.
                                red"norm"exp2'; LAM exp2''.
                                res[exp1'' "NODE" exp2'']; k
        ESAC

/[exp1 "IS" exp2] ->
        match(exp2)<exp1,env>; LAM<a,exp1'>.
        CASE a
        / TT ->         red(m)(is-fn[exp1' "IS" exp2]); k
        / FF ->         res(FF); k
        / ?  ->         res[exp1' "IS" exp2]; k
        ESAC
```

```
/[mon-op exp1] ->
        CASE mon-op
        / "ACTIVATE" ->
                m NE "norm" -> res(?); k,
                red"norm"exp1; LAM exp1'.
                res(mon-op-fn[mon-op exp1']); k
        / ? ->
                match(mon-op-arg(mon-op))<exp1,env>; LAM<a1,exp1'>.
                CASE a1
                / TT ->         red(m)(mon-op-fn[mon-op exp1']); k
                / FF ->         res(?); k
                / ?  ->         res[mon-op exp1']; k
                ESAC
        ESAC
```

```
/[exp1 exp2] ->
        match(lam-query-query)<exp1,env>; LAM<a1,exp1'>.
        CASE a1
        / TT ->
                LET ["LAM" exp11 "." exp12] = sus-exp(exp1')
                LET env1 = sus-env(exp1') IN
                match(exp11)<exp2,env>; LAM<a2,exp2'>.
                CASE a2
                / TT -> is-ide(exp11) ->
                                bind(env1)<exp11,exp2'>; LAM env'.
                                red(m)<exp12,env'>; k,
                        red(m)(app-fn[exp1' exp2']); k
                / FF -> res(?); k
                / ?  -> red"norm"exp1'; LAM exp1'';
                        res[exp1'' exp2']; k
                ESAC
        / FF ->
                red"norm"<exp2,env>; LAM exp2'.
                res[exp1' exp2']; k
        ESAC

/["<" exp* ">"] ->
        LET exp'* = map-list(LAM exp'.<exp',env>)(exp*)
        LET exp' = ["<" exp'* ">"] IN
        m NE "norm" -> res(exp'); k,
        red-list"norm"(exp'); k

/[exp1 rep-op]
/["ID" q]
/["ID" q "#" n] ->
        find(env)(exp); LAM<a,exp'>.
        CASE a
        / TT ->         red(m)(exp'); LAM exp''.
                        rebind(env)(exp,exp'');
                        res(exp''); k
        / FF ->         res(exp); k
        ESAC
```

```
        / NUMBER ?
        / QUOTE ?
        / TRUTH ? ->
                res(exp); k

        / ? ->
                exp EQ ? -> res(?); k, ?

        ESAC


WITH map-list(f:(Exp -> Exp))(exp*) : Exp* =
        CASE exp*
        / <> ->                 <>
        / exp1 PRE exp1* ->     f(exp1) PRE map-list(f)(exp1*)
        ESAC


WITH red-list(m)["<" exp* ">"]; k : C =
        CASE exp*
        / <> ->                 <>
        / exp1 PRE exp1* ->     red(m)(exp1); LAM exp1'.
                                red-list(m)["<" exp1* ">"];LAM["<" exp1'* ">"].
                                LET exp'* = exp1' PRE exp1'* IN
                                res["<" exp'* ">"]; k
        ESAC


! Matcher
! -------
! match(exp0)(exp);y  tries to make exp match the shape of exp0, by reducing
! (as little as possible) and by inverting operators.  It is basically 1-level,
! although [exp rep-op] does not count in this respect.  The A-component of the
! value passed to the continuation y is
!       TT if the match was successful
!       FF if the match failed (i.e. was impossible)
!       ?  if the normal form of exp is not a basic expression, thus containing
!          potential redexes and free variables.

WITH match(exp0)(exp); y : C =

        (exp0 EQ ?) OR (exp0 IS ["ID" q]) OR (exp0 IS ["ID" q "#" n]) ->
                res-a<TT,exp>; y,

        red"part"exp; LAM exp'.
        NOT is-basic(exp') ->
                res-a<?,exp'>; y,

        CASE exp0

        /["LAM" exp1 "." exp2] ->
                CASE exp'
                / <["LAM" exp1' "." exp2'], env> ->
                        res-a<TT,exp'>; y
                / ? ->
                        res-a<FF,?>; y
                ESAC
```

```
/[exp1 di-op exp2] ->
        CASE (di-op,exp')
        /("AUG", ["<" exp'* ">"])
        /("PRE", ["<" exp'* ">"]) ->
                SIZE exp1'* EQ 0 -> res-a<FF,?>; y,
                LET exp1' = di-op-inv(di-op,1,exp')
                LET exp2' = di-op-inv(di-op,2,exp') IN
                res-a<TT, [exp1' di-op exp2']>; y
        / ? ->
                res-a<FF, ?>; y
        ESAC

/[exp1 "NODE" exp2] ->
        CASE exp'
        /[exp1' "NODE" exp2'] ->
                res-a<TT,exp'>; y
        / ? ->
                res-a<FF,?>; y
        ESAC

/[mon-op exp1] ->
        CASE (mon-op,exp')
        /("NUMBER", NUMBER ?)
        /("QUOTE",  QUOTE ? )
        /("TRUTH",  TRUTH ? )
        /("CC",     CC ?    ) ->
                LET exp1' = mon-op-inv(mon-op,exp') IN
                res-a<TT, [mon-op exp1']>; y
        ESAC

/["<" exp* ">"] ->
        CASE exp'
        /["<" exp'* ">"] ->
                SIZE exp* NE SIZE exp'* ->
                        res-a<FF,?>; y,
                res<TT,exp'>; y
        / ? ->
                res-a<FF,?>; y
        ESAC

/[exp1 rep-op] ->
        CASE (rep-op,exp')
        /("*", ["<" exp'* ">"]) ->
                match-list(exp1)["<" exp'* ">"]; y
        /("+", ["<" exp'* ">"]) ->
                SIZE exp'* EQ 0 -> res-a<FF,?>; y,
                match-list(exp1)["<" exp'* ">"]; y

/ NUMBER ?
/ QUOTE ?
/ TRUTH ? ->
        exp0 EQ exp' -> res-a<TT,exp'>; y,
        res-a<FF,?>; y

ESAC
```

```
WITH match-list(exp0)["<" exp* ">"]; y : C =

        CASE exp*

        / <> ->
                res-a<TT, ["<" exp* ">"]>; y
        / exp1 PRE exp1* ->
                match(exp0)(exp1); LAM<a,exp1'>.
                match-list(exp0)["<" exp1* ">"]; LAM<a',exp'>.
                CASE (a,a',exp')
                /(TT,TT,["<" exp1'* ">"]) ->
                        LET exp'* = exp1' PRE exp1'* IN
                        res-a<TT, ["<" exp'* ">"]>; y
                /(FF,? ,?)
                /(?, FF,?) ->
                        res-a<FF,?>; y
                /(?, ?, ["<" exp1'* ">"]) ->
                        red"norm"exp1'; LAM exp1''.
                        red"norm"["<" exp1'* ">"]; LAM ["<" exp1''* ">"].
                        LET exp''* = exp1'' PRE exp1''* IN
                        res-a<?, ["<" exp1''* ">"]>; y
                ESAC

        ESAC


IN      (LAM exp. set-up(exp); red"norm"exp; stop) : (Files -> Exp -> Exp)

END
```

G. Loop Semantics (in LAMB)

```
LAMB "LOOP-Semantics"

(LAM <pp, cc, ee-list, ee, oo, repeat, update-list, initial-s,
    update, content>. pp)(
  FIXLAM
    < pp, cc, ee-list, ee, oo, repeat, update-list, initial-s,
      update, content>.
   < (LAM "Read-cmd;Cmd;Write-cmd"NODE<read-cmd, cmd, write-cmd>.
      LAM n*.
       (LAM <"READVar*"NODE<var*>, "WRITEExp+"NODE<exp+>>.
        (LAM s1.
         (LAM s2. ee-list(exp+)(s2))(
           cc(cmd)(s1)))(
         update-list(<var*, n*>)(initial-s)))(
        < read-cmd, write-cmd>)), (
     LAM cmd0.
      LAM s.
       (LAM ##0.
        ##0 IS ("Cmd;Cmd"NODE<cmd1, cmd2>) ->
          (LAM "Cmd;Cmd"NODE<cmd1, cmd2>.
           cc(cmd2)(cc(cmd1)(s)))(##0),
        ##0 IS ("Var:=Exp"NODE<var, exp>) ->
          (LAM "Var:=Exp"NODE<var, exp>.
           (LAM n. update(<var, n>)(s))(
             ee(exp)(s)))(##0),
        ##0 IS ("TOExpDOCmd"NODE<exp, cmd>) ->
          (LAM "TOExpDOCmd"NODE<exp, cmd>.
           (LAM n. repeat(n)(cc(cmd))(s))(
             ee(exp)(s)))(##0),
        ##0 IS ("(Cmd)"NODE<cmd>) ->
          (LAM "(Cmd)"NODE<cmd>. cc(cmd)(s))(##0),
        ?)(cmd0)), (
     LAM exp0+.
      LAM s.
       (LAM ##0.
        ##0 IS <exp> ->
          (LAM <exp>. <ee(exp)(s)>)(##0),
        ##0 IS (exp PRE exp+) ->
          (LAM exp PRE exp+.
           ee(exp)(s) PRE ee-list(exp+)(s))(##0),
        ?)(exp0+)), (
```

```
LAM exp0.
 LAM s.
  (LAM ##0.
   ##0 IS ("ExpOpExp"NODE<exp1, op, exp2>) ->
     (LAM "ExpOpExp"NODE<exp1, op, exp2>.
      (LAM <n1, n2>.
       oo(op)(<n1, n2>))(
        < ee(exp1)(s), ee(exp2)(s)>))(##0),
   ##0 IS ("Var"NODE<var>) ->
     (LAM "Var"NODE<var>. content(var)(s))(##0),
   ##0 IS ("Num"NODE<num>) ->
     (LAM "Num"NODE<num>. num)(##0),
   ?)(exp0)), (
LAM op.
 LAM <n1, n2>.
  (LAM ##0.
   ##0 IS "+" ->
     (LAM "+". n1 PLUS n2)(##0),
   ##0 IS "-" ->
     (LAM "-". n1 MINUS n2)(##0),
   ##0 IS "*" ->
     (LAM "*". n1 MULT n2)(##0),
   ##0 IS "/" ->
     (LAM "/". n1 DIV n2)(##0),
   ?)(op)), (
LAM n.
 LAM c.
  LAM s.
   n EQ ? -> ?,
   n EQ 0 -> s,
   repeat(n MINUS 1)(c)(c(s))), (
LAM <var0*, n0*>.
 LAM s.
  SIZE var0* EQ 0 -> s,
  (LAM <var PRE var*, n PRE n*>.
   update-list(<var*, n*>)(update(<var, n>)(s)))(
    < var0*, n0*>)), (LAM var. ?), (
LAM <var, n>.
 LAM s.
  LAM ##0.
   ##0 EQ var -> n,
   s(##0)), (LAM var. LAM s. s(var))>)

END
```

MD-30 SIS- REFERENCE MANUAL

TRYK: DAIMI/RECAU