Mathematical Semantics

and

Compiler Generation

by

Peter David Mosses

B.A., Oxon. (1970)

Submitted for the Degree of
Doctor of Philosophy
at the
University of Oxford

April 1975

Signature of Author: P D Mosses

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

ABSTRACT

The method of giving the mathematical (or denotational) semantics of a programming language, as developed by Dana Scott and Christopher Strachey, is taken as a starting-point. The main work is towards a general and portable compiler-generator system, which generates a compiler for any programming language from formal descriptions of its syntax and of its mathematical semantics.

The usual notation for describing a mathematical semantics is examined, and found to be lacking a formal description itself. A Mathematical Semantics Language (MSL), which is a simple variant of the notation of so-called semantic equations, is introduced and formally defined. MSL is for use with the compiler-generator, but it is argued that some rigorously-defined notation such as MSL is in any case desirable for use when giving definitive descriptions of programming languages. It is shown that MSL is adequate for describing the mathematical semantics of any (deterministic) programming language.

In the process of defining MSL, a useful extension LAMB of Scott's language LAMBDA is developed. Reduction rules for LAMB are given, and a reduction algorithm using the "call-by-need" technique is described. This algorithm always finds a normal form if one exists, and it is "usually" optimal in efficiency.

The compiler-generator system is described both informally and in MSL (which can be used as an expression-based programming language). The system uses LAMB as a universal code for its data and for generated compilers, and it operates entirely by evaluating applicative combinations of LAMB-expressions. It can be seen that a generated compiler correctly simulates the mathematical semantics of the described programming language, provided that the under-lying LAMB-reducer is implemented correctly.

The results of compiling and evaluating a few simple algorithms, expressed in a formally-defined mini-language, are given. In conclusion the present rudimentary state of the compiler-generator system is stressed, and suggestions are made as to how it might be developed into a useful research tool.

ACKNOWLEDGMENTS

To my supervisor Christopher Strachey, whose initial enthusiasm and constant encouragement were largely responsible for the successful development of the compiler-generator system from a wild hope into a reality;

To Dana Scott, whose joint work with Strachey provided a firm starting-point for this dissertation, and who can never be repaid for his patient explanations; and

To members of the Programming Research Group, Oxford, especially to George Ligler, for many useful conversations about this work.

Many grateful thanks also to Julia Bayman, who typed most of this dissertation and willingly coped with the vagaries of my manuscript.

The work reported in this dissertation was supported by a Research Studentship from the Science Research Council of Great Britain.

CONTENTS

LIST OF TABLES

"It is all very well to aim for a more 'abstract' and a 'cleaner' approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored. The reason is obvious: in the end the program must still be run on a machine - a machine which does not possess the benefit of 'abstract' human understanding, a machine that must operate with finite configurations. Therefore, a mathematical semantics, which will represent the first major segment of the complete, rigorous definition of a programming language, must lead naturally to an operational simulation of the abstract entities, which (if done properly) will establish the practicality of the language, and which is necessary for a full presentation."

Dana Scott, 1970, in *Outline of a Mathematical Theory of Computation.*

INTRODUCTION

This dissertation presents a compiler-generator system, which differs from previous compiler-generators(*) in many respects. Its main novelty is that it produces a compiler for a programming language from a formal description which consists of an unambiguous context-free grammar and a mathematical semantics. It is not claimed that generated compilers are very efficient, but it can be shown that they are "correct". The descriptive power of the notation used for mathematical semantics ensures complete generality; also this notation can be modified or extended using the system, giving a necessary flexibility. The system, the compilers it generates and the programs they compile are all run by the same easily-implemented interpreter, so the system is highly portable.

Our work also includes the first formal definition of a notation for mathematical semantics. Although the motivation for this comes mainly from its use in the compiler-generator system, it can be argued that a description of a mathematical semantics cannot be *guaranteed* unambiguous or precise in the absence of a formal definition of the notation used.

Three general areas of study in the theory of programming languages can be distinguished (after Morris [1]):

(i) Syntax, which concerns the form of the legal programs of a language.

(ii) Semantics, which deals with the relation between programs of a language and the abstract objects they denote.

(*) Such as those surveyed in [32].

(iii) Pragmatics, which treats the relation between a language and its users (human or mechanical).

Of course there is a lot of overlap between these areas - in particular there is a strong interaction between that part of pragmatics concerned with the design and definition of programming languages, and the general study of syntax and semantics. The latter aims to improve the conceptual basis of programming languages, so enabling clearer definitions and (ideally!) influencing their design; this in turn facilitates further study of syntax and semantics.

We see our work as contributing to the theory of programming languages in several ways:

(a) The compiler-generator will enable semantic descriptions to be checked empirically ("debugged"), increasing their accuracy.

(b) Language designers using mathematical semantics will be able to assess easily the effect of alternative choices.

(c) Language designers will perhaps be encouraged to use mathematical semantics by the provision of the compiler-generator. It is believed that many current languages would have been better-designed if an attempt at formulating their mathematical semantics had been made during the design process.

(d) Indirect benefit might come from a student of the theory acquiring familiarity with a wide range of languages. It is hoped that all the most interesting languages will soon have tested mathematical semantics, and the compiler-generator will provide portable compilers for them.

(e) The formal definition of a precise notation for mathematical semantics might allow machine-aided proofs of program equivalence and correctness, in conjunction with a system such as Milner's LCF [2].

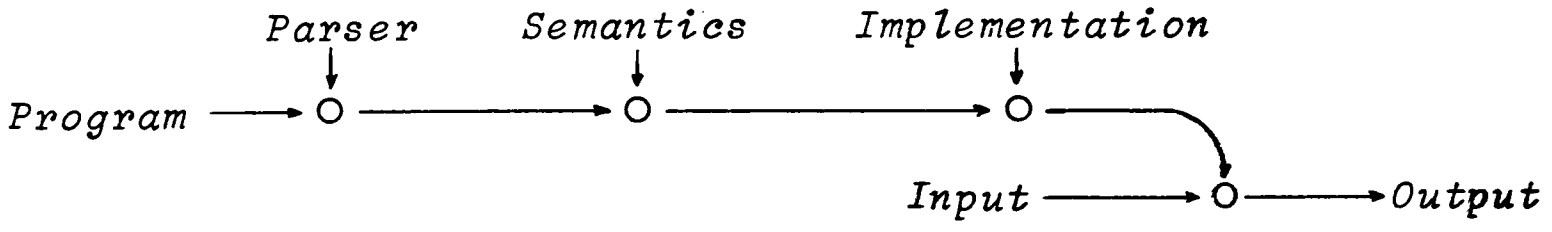
(f) The compiler produced from a mathematical semantics could be taken as a reference standard for hand-written compilers.

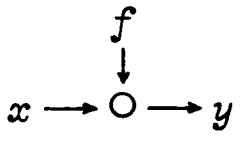
It is true that the value of our contribution depends entirely on the continuation of theoretical interest in the method of mathematical semantics. However a glance at the literature [3 , 4 , 5 , 6 , 7 , 8] gives no reason to fear that this interest will fade in the near future - on the contrary there is even the expectation that the other approaches of interest, such as the "axiomatic" method of Hoare [9 , 10] and the "operational" approach [11 , 12 , 13 , 14] will be justified by connecting them with mathematical semantics. For a detailed survey of all the current approaches, see Ligler [15], which includes an extensive bibliography.

Chapter 1 of this dissertation introduces relevant notions, notation and theorems for use in later chapters. Chapter 2 is mainly concerned with the rigorous definition of a notation called MSL (Mathematical Semantics Language) for mathematical semantics, but also introduces LAMB, our basic interpreted code. Chapter 3 discusses the properties of LAMB and also the correctness of its interpreter. The compiler-generator system is presented in detail in Chapter 4, and Chapter 5 gives an example of its use. To motivate the first three chapters, we now examine the way in which the compiler-generator uses MSL and LAMB.

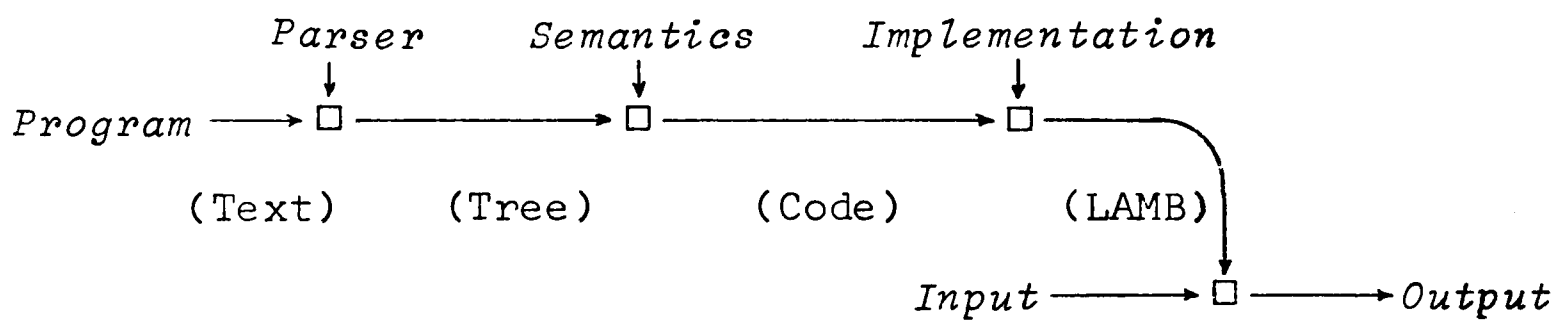
As stated in the literature [6 , 7 , 8 , 16 , 17] , a mathematical semantics gives a correspondence between programs and abstract mathematical entities. On further investigation we find that these programs are considered to be parse-trees ("annotated deduction trees" according to [6]); and that the mathematical entities (*values* for short) depend on some unspecified functions

whose provision constitutes an implementation of the language. (The convenience of such a factorization is perhaps one of the main reasons for the popularity of mathematical semantics.) Hence an abstract "compile-and-run" system could look like this:



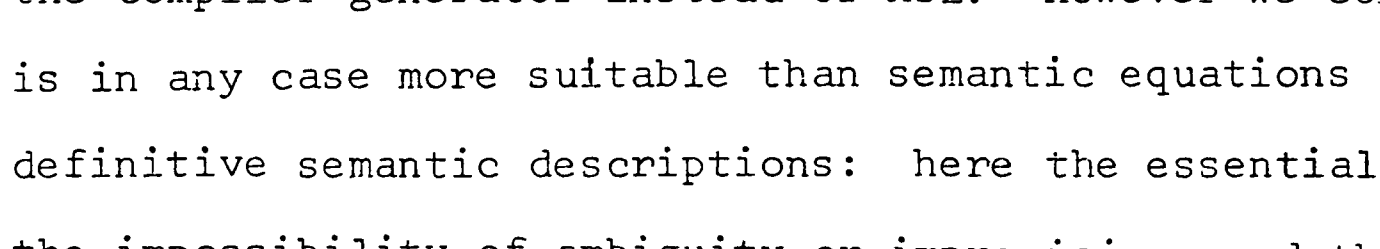
where  represents the application of an abstract function f to an abstract value x yielding y. *Parser* denotes an abstract function from text to parse-trees, *Semantics* yields values dependent on certain functions, and *Implementation* removes that dependence to give a function from input to output.

To achieve an operational simulation of these abstract entities on a (real) computer, we encode all the functions and values in LAMB, which is based on Scott's language LAMBDA [18], and implement a LAMB interpreter. If □ denotes the coded form of O above - it combines the code of a function with the code of a value and yields the code of the result - then we can sketch a concrete compile-and-run system as:



where all components (including *Program*) are denoted by expressions in LAMB.

The compiler-generator described in Chapter 4 produces the LAMB-code of the above components from their "high-level" descriptions: a BNF-like grammar for *Parser*, MSL expressions for the rest Here we discuss only *Semantics*. The formal description of MSL consists of a grammar and a LAMB-expression. The grammar yields the code of an *MSL-Parser* by the use of another part of the system; whereas the LAMB-expression *is* the code of the definitive *MSL-Semantics* (which incorporates *MSL-Implementation*). Therefore *Semantics* above can be produced by:



Only an initial version of *MSL-Semantics* needs to be hand-coded, as a "circular" description of MSL in MSL can be used to incorporate extensions or amendments. The complete system is sketched in Chapter 4.

The notation of so-called semantic equations has often been used to describe a mathematical semantics. Unfortunately some of its features prohibit a simple formal definition of its syntax and semantics (with present techniques), the main culprit being the double use of so-called meta-variables (ranging over phrases) on the left-hand side of a semantic equation: they name the sub-phrases of a phrase as well as indicating its form. In the absence of such a formal definition, semantic equations cannot be used in the compiler-generator instead of MSL. However we contend that MSL is in any case more suitable than semantic equations for giving definitive semantic descriptions: here the essential property is the impossibility of ambiguity or imprecision, and this can only be

guaranteed to hold when the notation has a formal definition. The attitude of authors of semantic equations seems to be that extra parentheses can always be inserted if a description is shown to be ambiguous - this is quite acceptable when the purpose of the semantics is description of an existing programming language, but surely has no place when defining a new language? An aggravating factor is the practice of adding to the basic notation of semantic equations, to obtain greater readability and/or compactness - necessarily this extra notation can be described only informally.

MSL is quite similar to the basic notation of semantic equations [6 , 7 , 8] , differing mainly in using ordinary variables instead of meta-variables, and in having definitions similar to PAL [19]. It is easy to translate semantic equations into equivalent MSL descriptions, and vice versa. We hope that MSL (or a modified version) will be used instead of semantic equations when programming languages are defined, although we expect the latter to remain popular for use in proofs due to the compactness of the notation of meta-variables.

We leave remarks about possible future development of MSL and the compiler-generator to the Conclusion.

CHAPTER 1

Prerequisites

This chapter introduces those definitions, notations and theorems from the literature which we shall need in later chapters. For motivation and proofs the reader is referred to [6 , 16 , 17 , 20] (lattices, λ-notation and mathematical semantics); [18](Pw, LAMBDA, retracts); and [6 , 7 , 8] (semantic equations and techniques).

Section j of Chapter i is referred to as "§$i.j$". The k-th Theorem or Lemma of a section may be referred to without the qualifying noun, i.e. as "$i.j.k$". The k-th equation is referred to (from within the same section) as "(k)".

Inside a formula a comment will be introduced by an exclamation mark "!", and will continue to the end of the line.

§1.1 Lattices

Definitions

A *lattice* consists of a set, D, structured by a partial ordering $\sqsubseteq$, such that any two elements of D have a unique least upper bound in D.

The *least upper bound* (*l.u.b.*) $\sqcup X$ of an arbitrary subset $X \subseteq D$ is uniquely characterized by

for all $y \in D, \sqcup X \sqsubseteq y$ iff for all $x \in X$, $x \sqsubseteq y$.

A *complete lattice* is a lattice in which $\sqcup X$ exists for all $X \subseteq D$.

A *domain* is a complete lattice which is continuous in the sense of [20].

Unless otherwise specified D, D', D_0, D_1, etc. will denote *domains*. In later chapters, more mnemonic names such as E, N, Exp, Env, etc., will also be assumed to denote domains.

$\perp_D$ denotes $\sqcup\ \emptyset$ in D, the *bottom* element of D. $\top_D$ denotes $\sqcup\ D$, the *top* element. The subscripts will often be omitted.

A set $X \subseteq D$ is said to be *directed* iff

$\{x,y\} \subseteq X$ implies $\sqcup\{x,y\} \in X$.

A function f from D_0 to D_1 is said to be *continuous* iff whenever $X \subseteq D_0$ is directed then

$f(\sqcup X) = \sqcup\{f(x) \mid x \in X\}$.

Theorem 1.1.1 (Tarski)

If f is a continuous function from D into itself, then f has a *least fixed point*, which we denote by $fix(f)$; i.e. $f(fix(f)) = fix(f)$.

Also $fix(f) = \sqcup\{f^n(\perp) \mid n \in \omega\}$,

where ω is the set of non-negative integers,

and f^n is the n-fold composition of f with itself.

Definitions

$D_0 \times D_1$ denotes the cartesian product of D_0 and D_1; i.e. the set of pairs $\{\langle x,y\rangle \mid x \in D_0, y \in D_1\}$, ordered by $\langle x,y\rangle \sqsubseteq \langle x',y'\rangle$ iff $x \sqsubseteq x'$ and $y \sqsubseteq y'$.

$D_0 \to D_1$ denotes the domain of continuous functions from D_0 to D_1, ordered point-wise; i.e. if $f,g \in D_0 \to D_1$ then $f \sqsubseteq g$ iff for all $x \in D_0$, $f(x) \sqsubseteq g(x)$.

$D_0 + D_1$ denotes the *separated sum* of D_0 and D_1, i.e. the set $\{\langle 0,x\rangle \mid x \in D_0\} \cup \{\langle 1,y\rangle \mid y \in D_1\} \cup \{\bot,\top\}$ ordered by $z \sqsubseteq z'$ iff

$z = \bot$ or $z' = \top$, or

$z = \langle 0,x\rangle$ and $z' = \langle 0,x'\rangle$ and $x \sqsubseteq x'$, or

$z = \langle 1,y\rangle$ and $z' = \langle 1,y'\rangle$ and $y \sqsubseteq y'$.

$D_0 \times D_1 \times \ldots \times D_{n-1}$ denotes the natural extension of $D_0 \times D_1$ to n arguments (rather than the iterated binary product).

D^n denotes $D \times D \times \ldots \times D$ (n times, $n \geq 2$)

D^1 denotes $\{\langle x\rangle \mid x \in D\}$ which is isomorphic to, but not identical with, D.

D^0 denotes $\{\langle\rangle\}$, the one-point domain.

$D_0 + D_1 + \ldots + D_{n-1}$ denotes the natural extension of $D_0 + D_1$ to n arguments.

D^* denotes $D^0 + D^1 + \ldots + D^n + \ldots$

Theorem 1.1.2 (Scott)

$D_0 \times D_1$, $D_0 \to D_1$, $D_0 + D_1$, D^* are all domains.

The following theorem enables the easy description of a wide variety of compound domains.

Theorem 1.1.3 (Scott)

There exists a unique minimal solution (up to isomorphism) to any system of equations defining domains recursively by expressions involving the operators $\times$, $\to$, $+$ and $*$.

(The statement of this theorem is made more precise in §1.3, using retracts).

Definitions

The *proper* elements of a domain are those elements not equal to $\top$ or $\bot$.

A domain is said to be *flat* or *primitive* if all distinct proper elements are incomparable.

Bool denotes the four-point flat domain $\{\bot, \top,$ *true*, *false*$\}$

Int denotes the flat domain of non-negative integers:

$\{\bot, \top\} \cup \{0, 1, 2, \ldots\}$

§1.2 λ-notation

λ-*notation* should be distinguished clearly from the λ-*calculus* of Church [21]. An expression in the former denotes an element of a domain (perhaps dependent on the values denoted by some variables), whereas an expression in the λ-calculus has no specified denotation - in fact it may be given various interpretations, e.g. as in [22]. We shall use an extended version of λ-notation, allowing tuples to be denoted as well as functions. Note that not every expression in λ-notation has a meaning, e.g. 1(2). As usual D, D′ etc. denote arbitrary domains.

Definitions

$(\lambda x{:}D.M)$ denotes the function from D to D′ given by M as x varies over elements of D, where D′ is determined by M.

If M denotes a function f from D to D′, and N denotes a value b in D, then $M(N)$ denotes the value of f at b, in D′.

If M denotes an element of the domain Bool, and N, N' denote elements d, d' both in the same domain D, then $(M \to N, N')$ denotes

d if b is *true*

d' if b is *false*

$\perp_D$ if b is $\perp_{Bool}$

$\top_D$ if b is $\top_{Bool}$

If M_i denotes an element $d_i \in D_i$ for i = 0, 1, ... , n-1, then $\langle M_0, M_1, \ldots, M_{n-1}\rangle$ denotes the element $\langle d_0, d_1, \ldots, d_{n-1}\rangle$ of $D_0 \times D_1 \times \ldots \times D_{n-1}$.

If M denotes an element $\langle d_0, d_1, \ldots, d_{n-1}\rangle$ of $D_0 \times D_1 \times \ldots \times D_{n-1}$, and K denotes an element k of Int such that $k \in \{0, 1, \ldots, n-1\}$, then $(M{\downarrow}K)$ denotes d_k.

Numerals and the usual arithmetic operations are allowed.

Conventions

Parentheses may be omitted from expressions when it is clear how they may be re-inserted. By convention $(M(N))(K)$ may be written as $M\ N\ K$; and an abstraction $\lambda x{:}D.\ M$ "continues as long as possible", e.g. $\lambda x{:}D.\ f(g)$ is equivalent to $(\lambda x{:}D.\ f(g))$.

§1.3 Pw

We shall make use of a rather special domain called Pw. It is the set of all subsets of ω (the set {0, 1, 2, ... } of non-negative integers), ordered by set inclusion. The details of the topology with which Scott has made Pw a so-called continuous lattice need not concern us here - they can be found in [18].

Definitions

$\{e_n \mid n \in \omega\}$ is the standard enumeration of the finite subsets of ω, where

$$e_n = \{k_0, k_1, \ldots, k_{m-1}\}$$

provided that $k_0 < k_1 < \ldots < k_{m-1}$ and $n = \Sigma\ 2^{k_i}$.

Properties

$k \in e_n$ implies $k < n$.

The relations $k \in e_n$, $e_n \subseteq e_m$, $e_n = e_m \cup e_k$ are all (primitive) recursive in k, m, n.

For all $x \in \mathsf{Pw}$, $x = \cup \{e_n \mid e_n \subseteq x\}$.

Definitions

A function f from Pw to Pw is said to be *continuous* iff for all $x \in \mathsf{Pw}$ we have

$$f(x) = \cup \{f(e_n) \mid e_n \subseteq x\}.$$

(n,m) is the standard coding of the pairs of integers as single integers, where $(n,m) = \frac{1}{2}(n+m)(n+m+1) + m$.

The *graph* of a continuous function $f \in \mathsf{Pw} \to \mathsf{Pw}$ is defined by

$$Graph(f) = \{(n,m) \mid m \in f(e_n)\}.$$

The *function* determined by any set $u \subseteq \omega$ is defined by

$$Fun(u)(x) = \{m \mid \exists e_n \subseteq x.\ (n,m) \in u\}$$

Theorem 1.3.1 (Scott) (The Graph Theorem)

Every continuous function is uniquely determined by its graph, in the sense that:

(1) $Fun(Graph(f)) = f$.

Conversely every set of integers determines a continuous function and we have:

(2) $u \subseteq Graph(Fun(u))$

where equality holds just in case u satisfies:

(3) whenever $(k,m) \in u$ and $e_k \subseteq e_n$ then $(n,m) \in u$.

Definition

A function f over several variables ranging over Pw is said to be continuous iff it is continuous in each variable separately. (For Scott's topology for Pw , this is equivalent to being continuous in the variables jointly.)

Theorem 1.3.2 (Scott) (The Substitution Theorem)

Continuous functions of several variables on Pw are closed under substitution.

Theorem 1.3.3 (Scott) (The Fixed-Point Theorem)

Every continuous function f: Pw → Pw has a least fixed point $fix(f)$ given by the formula

$$fix(f) = \cup\{f^n(\emptyset) \mid n \in \omega\}$$

where $\emptyset$ is the empty set and f^n is the n-fold composition of f with itself.

Theorem 1.3.4 (Scott) (The Extension Theorem)

Let X and Y be arbitrary topological spaces where $X \subseteq Y$ as a subspace. Then every continuous function $f: X \to Pw$ can be extended to a continuous function $\bar{f}: Y \to Pw$.

Theorem 1.3.5 (Scott) (The Embedding Theorem)

Every T_0-space X with a countable basis for its topology can be continuously embedded in Pw.

§1.4 LAMBDA

LAMBDA is a primitive yet powerful language due to Scott [18]. It is an extension of the λ-calculus to include integers and conditionals, and its semantics gives every term a denotation in Pw.

Here we follow Scott in giving the syntax of LAMBDA informally with the semantics.

Syntax and Semantics of LAMBDA

$$0 = \{0\}$$

$$x + 1 = \{n+1 \mid n \in x\}$$

$$x - 1 = \{n \mid n+1 \in x\}$$

$$z \supset x, y = \{n \in x \mid 0 \in z\} \cup \{m \in y \mid \exists k.\ k+1 \in x\}$$

$$u(x) = \{m \mid \exists e_n \subseteq x.\ (n,m) \in u\}$$

$$\lambda x.\tau = \{(n,m) \mid m \in \tau[e_n/x]\}$$

The syntax is indicated on the left-hand side, and the meanings of the combinations are shown on the right-hand side as subsets of Pw. There is one primitive constant (0) and two unary functions ($x+1$, $x-1$). The ternary function $z \supset x, y$ is a conditional, testing for zero, and could also be defined by cases:

$$z \supset x, y = \perp, \quad \text{if } z = \perp ;$$
$$= x, \quad \text{if } z = \{0\} ;$$
$$= y, \quad \text{if } 0 \notin z \neq \perp ;$$
$$= x \cup y, \quad \text{if } 0 \in z \neq \{0\}.$$

The binary function $u(x)$ defines application (u is treated as a graph and x as a set). The variable binding operator $\lambda x.\tau$ is functional abstraction; $\tau[e_n/x]$ denotes the value of the term τ when the variable x is given the value e_n. A LAMBDA-term defines a function of its free variables.

Theorem 1.4.1 (Scott) (The Continuity Theorem)

All LAMBDA-definable functions are continuous.

Theorem 1.4.2 (Scott) (The Conversion Theorem)

The three basic principles (α), (β), (ξ) of λ-conversion (Church [21]) are all valid in LAMBDA.

Definition (The Paradoxical Combinator) (in LAMBDA)

$$Y = \lambda u.\ (\lambda x.\ u(x(x)))(\lambda x.\ u(x(x)))$$

Theorem 1.4.3 (Scott) (The First Recursion Theorem)

If u is the graph of a continuous function f, then $Y(u) = fix(f)$, the least fixed point of f.

Convention

A continuous function from Pw to Pw is identified with its graph.

Definition

A continuous function f of k variables is said to be *computable* iff the relationship

$$m \in f(e_{n_0})(e_{n_1}) \ldots (e_{n_{k-1}})$$

is recursively enumerable in the integer variables $m, n_0, n_1, \ldots, n_{k-1}$.

Theorem 1.4.4 (Scott) (The Definability Theorem)

For a k-ary continuous function the following are equivalent:

(1) f is computable;

(2) $\lambda x_0.\ \lambda x_1.\ \ldots\ \lambda x_{k-1}.\ f(x_0)(x_1) \ldots (x_{k-1})$ as a set is r.e.;

(3) $\lambda x_0.\ \lambda x_1.\ \ldots\ \lambda x_{k-1}.\ f(x_0)(x_1) \ldots (x_{k-1})$ is LAMBDA-definable.

Definitions (in LAMBDA)

$\perp = (\lambda x.\ x(x))(\lambda x.\ x(x))$

$x \cup y = (\lambda z.\ 0) \supset x, y$

$\top = Y(\lambda f.\ \lambda x.\ x \cup f(x+1))(0)$

$z \bar{\supset} x, y = z \supset (z \supset x, \top), (z \supset \top, y)$

which is $\top$ when 0 and $k+1$ are both in z.

$u \circ v = \lambda x.\ u(v(x))$

$\langle\rangle = \perp$

$\langle x \rangle = \lambda z.\ z \supset x, \perp$

$\langle x, y \rangle = \lambda z.\ z \supset x, (z-1 \supset y, \perp)$

$\langle x_0, x_1, \ldots, x_{n-1} \rangle = \lambda z.\ z \supset x_0, \langle x_1, \ldots, x_{n-1} \rangle (z-1)$

$u_x = u(x)$

1.5 Retracts

It is sometimes more convenient to define domains directly (as sub-spaces of Pw) as the ranges of retracts.

Definitions

An element $a \in [Pw \to Pw]$ is called a *retract* iff it satisfies

$a = a \circ a$

If a is a retract, we write $u{:}a$ for $u = a(u)$, and $\lambda x{:}a.\tau$ for $\lambda x.\ \tau[a(x)/x]$

Theorem 1.5.1 (Scott) (The Lattice Theorem)

The fixed points of any continuous function on Pw form a complete lattice (under $\subseteq$); while those of a retract form a continuous lattice (in the sense of [20]).

Definitions (in LAMBDA)

$fun = \lambda u.\ \lambda x.\ u(x)$

$$pair = \lambda u.\ \langle u_0,\ u_1\rangle$$

$$bool = \lambda u.\ u\ \supset\ 0,\ 1$$

$$int = \lambda u.\ u\ \supset\ 0,\ int(u\text{-}1)\ \supset\ u,\ u$$

(Note that *int* was defined recursively. A non-recursive form could be given by the use of *Y*, see 1.5.3.)

$$a \circ\!\!\to b = \lambda u.\ b \circ u \circ a$$

$$a \otimes b = \lambda u.\ \langle a(u_0),\ b(u_1)\rangle$$

$$a \oplus b = \lambda u.\ u_0\ \supset\ \langle 0,\ a(u_1)\rangle\ ,\ \langle 1,\ b(u_1)\rangle$$

Theorem 1.5.2 (Scott)

Suppose a and b are retracts. Then

(1) $a \circ\!\!\to b$ is a retract, and

$u{:}a\circ\!\!\to b$ iff $u = x{:}a.u(x)$ and for all $x{:}a$, $u(x){:}b$;

(2) $a \otimes b$ is a retract, and

$u{:}a\otimes b$ iff $u = \langle u_0, u_1\rangle$ and $u_0{:}a$ and $u_1{:}b$;

(3) $a \oplus b$ is a retract, and

$u{:}a\oplus b$ iff $u = \bot$ or $u = \top$ or

$u = \langle 0, u_1\rangle$ and $u_1{:}a$ or

$u = \langle 1, u_1\rangle$ and $u_1{:}b$.

Theorem 1.5.3 (Scott) (The Limit Theorem)

Suppose F is a continuous function that maps retracts to retracts; then $Y(F)$ is a retract.

Definition

If a is a retract, then *range(a)* denotes its range, which is the same as the set of its fixed points.

§1.6 Mathematical Semantics

(We attempt only a brief outline of the method.) A mathematical semantics for a programming language gives a correspondence between programs and abstract mathematical entities. We discuss first what a program is taken to be.

So as not to get involved in problems of ambiguity and syntax analysis, it is assumed that there is some means of forming parse-trees ("annotated deduction trees") from program texts, according to some unambiguous context-free grammar. The annotations or labels at the nodes of a parse-tree often contain semantically-irrevelant information, such as whether an expression is a summand or a multiplicand - this is conveniently hidden by choosing one label for each class of semantically-equivalent labels, and by eliminating irrelevant "chain-reduction" nodes. The resulting parse-trees can then be described by a (usually very ambiguous) grammar, which is in general much smaller than the original one. The names of the syntactic categories which this grammar defines are often used as variables ranging over corresponding syntactic *domains*.

The abstract mathematical entities, referred to simply as (semantic) values, are always considered to be elements of domains. In order to build up the value of a program from the values of its parts, a mathematical semantics gives a value not only to complete programs but also to all possible phrases. Basic phrases such as numerals or booleans usually have values in a primitive domain such as **Int** or **Bool**; whereas commands and compound expressions have values in compound domains of functions and tuples. In particular the value of a phrase is usually a function of a so-called *environment*, which is itself a function from identifiers to the values associated with them (by declarations in the surrounding context).

Sometimes the value of a command or expressions is also a function of a so-called *continuation* - **goto** commands can then ignore the value of the continuation, using instead the value of a label in the environment, whereas other types of command would use the value of the continuation. (There are other applications of this technique, which need not concern us here.) However the fundamental technique is to make the value of a phrase a function of an abstract store or state, which acts like a computer memory. It is often the case that identifiers can (notionally) be associated with fixed locations or addresses in a store, and that assignment statements change *not* this association but rather the "contents" of a location in the store. In brief, the environment, continuation and store are the principal components of the semantic context of a phrase.

A mathematical semantics tries to avoid arbitrary choices of representation; in particular it avoids giving a full implementation of a language by allowing the semantic value of a program to depend on various unspecified functions, such as ones for accessing and updating locations in the store.

To specify a mathematical semantics, the notation of so-called *semantic equations* is used. On the left-hand side of each equation, the application of a semantic function to a compound phrase is denoted. The function is usually identified by a script letter such as $\mathcal{E}$ or $\mathcal{C}$, the phrase is denoted by the terminal and non-terminal symbols of an alternative from the reduced BNF grammar mentioned earlier. The non-terminal symbols (subscripted if necessary) serve to name the corresponding sub-phrases of the compound phrase. On the right-hand side, λ-notation (or LAMBDA) is used to denote an element of a domain dependent on the semantic values of the sub-phrases - this is the specified semantic value of the compound phrase Usually it is convenient to specify several semantic functions of

different functionalities, which together give the complete correspondence between phrases and values - then the functions are understood to be mutually recursive.

For example, supposed the reduced grammar contains

(1) (Cmd) $\gamma ::= \ldots \mid \gamma \ ; \ \gamma \mid \ldots$

so that a command γ may be a sequence of two other commands. If $\mathcal{F}$ denotes the semantic function for commands, then $\mathcal{F}[\![\gamma]\!]$ denotes the semantic value of the phrase denoted by γ. Considering a command to specify a transformation on a state σ belonging to a domain S, we might want:

(2) $$\mathcal{F}[\![\gamma]\!] \in [Env \to [S \to S]]$$

and then, ignoring the possibility of **goto** commands, errors and non-termination, the natural meaning of $\gamma_1 \ ; \ \gamma_2$ would be the composition of $\mathcal{F}[\![\gamma_1]\!]\rho$ with $\mathcal{F}[\![\gamma_2]\!]\rho$. The semantic equation for this is

(3) $$\mathcal{F}[\![\gamma_1 \ ; \ \gamma_2]\!] = \lambda\rho{:}Env.\ \lambda\sigma{:}S.\ \mathcal{F}[\![\gamma_2]\!]\rho(\mathcal{F}[\![\gamma_1]\!]\rho\sigma)$$

If a phrase can consist of a list of an arbitrary number of other phrases, it is often more natural to denote it as such than to choose left or right recursion in the grammar. The "..." convention is commonly used. For example if a possible expression is a tuple of one or more other expressions, this is denoted in the grammar by

(4) $$\varepsilon ::= \ldots \mid \langle \varepsilon \ , \ldots , \ \varepsilon \rangle \mid \ldots$$

and a possible semantic equation is

(5) $$\mathcal{E}[\![\langle \varepsilon_1, \ldots, \ \varepsilon_n \rangle]\!] = \lambda\rho{:}Env\ .\ \langle \mathcal{E}[\![\varepsilon_1]\!]\rho, \ldots, \ \mathcal{E}[\![\varepsilon_n]\!]\rho \rangle$$

where $\mathcal{E}[\![\varepsilon]\!] \in E$ and $E = E^* + \ldots$.

Note that here n is also a variable.

The equations (3) and (5) are atypical in their simplicity, and various abbreviations and conventions are needed to keep readability in a description of a "real" programming language. These include:

(a) global restriction of variables to domains, enabling domains to be omitted from λ-abstractions;

(b) introduction of infix operators (e.g. ∘, *) to eliminate λs and parentheses;

(c) definition of auxiliary constants and functions, and "let" and "where" definitions; and

(d) the frequent omission of the injections and projections between sum domains and their summands;

As regards (d) it should be noted that in general an element of $D_0 + D_1$ is not an element of D_0 or of D_1 (and vice versa), and in particular an element of D^n is not an element of D^*, so there would be a considerable amount of "book-keeping" necessary without this convention. In fact the present notation for injections (δ in $[D_0 + D_1]$) and projections ($\delta \mid D_0$) is neither convenient nor general enough - consider for example D + D - and the lack of a precise explanation of its connection with the domain definitions makes it difficult to improve on it.

CHAPTER 2

Definition of MSL

Our intention is to give a rigorous definition of MSL - a notation for mathematical semantics. First we shall present our means of definition: §2.1 will give a broad outline, and §§2.2-2.4 will provide the details. Then we shall describe MSL informally, giving an example of its use, in §2.5. The formal definition of MSL is contained in Appendices B and C.

§2.1 Outline of Method

MSL is a notation for describing the mathematical semantics of programming languages, so to define MSL itself we need to specify a correspondence between MSL texts and abstract semantic functions. Naturally, part of our definition will consist of an unambiguous grammar for MSL, enabling us to form parse-trees from MSL texts; but how shall we define the mapping from parse-trees to semantic functions? Our approach will be mathematical, hence it will be convenient if arbitrary semantic functions are embedded in a mathematical domain; in fact we shall accomplish this by embedding arbitrary parse-trees in a domain in such a way that any semantic function becomes an element of a domain of continuous functions.

To define a mapping from parse-trees of MSL to functional values, several notations might be considered suitable, namely semantic equations, λ-notation and Scott's language LAMBDA. Here we choose LAMBDA as the basis for our notation, for two reasons: a theorem by Scott shows that a function is "computable" iff it is LAMBDA-definable; and the semantics of LAMBDA are very simple. Of course LAMBDA-terms denote elements of Pw, rather than values in arbitrary recursively-defined domains, but it is quite easy to embed parse-trees and semantic functions in Pw, as we shall show in §2.4.

We could give a rigorous definition of the semantics of MSL using LAMBDA. Scott calls LAMBDA a "high-level" programming language for recursive function theory, but for purposes here it seems that the level is not quite high enough, as the definition of MSL would take many pages and be difficult to read. What we shall do is define another language called LAMB, based on LAMBDA and having the same expressive power but including tuples and various operators. LAMB is simple enough to be implemented directly, as we shall show

in Chapter 3, but it is much more compact than LAMBDA and can be used to give a manageable defintion of MSL.

In fact we shall base MSL on LAMB, and define the semantics of MSL as the composition of the semantics of LAMB with a syntactic mapping from MSL to LAMB. This is to facilitate the use of MSL in the compiler-generator.

We shall make it possible to distinguish three "types" in MSL: atoms (integers, strings, booleans), non-atoms (functions, tuples) and "error" values. The inclusion of this feature in LAMB will make it easy to show that MSL and LAMB have the same expressive power. We should like to define LAMB as a syntactic mapping into LAMBDA, but it will be easier to show the equivalence of LAMB and LAMBDA if we first extend LAMBDA to a "typed" language called LAMA, and then define LAMB in terms of LAMA. We shall need to introduce a new semantics for LAMBDA before we can extend it to LAMA.

Our definitions of LAMA and LAMB will use the informally-defined notation of semantic equations - we hope that this will be rigorous enough to give LAMB a precise interpretation in Pw. We could give the LAMBDA-expressions corresponding to these definitions, as they are quite short, but we see no need for this here.

§2.2 Typed LAMBDA and LAMA

We shall give a new semantics for LAMBDA, and compare it to the original one given by Scott (quoted in §1.4). To make this comparison easier, we shall express both semantics in the form of semantic equations, using λ-notation to denote elements of arbitrary domains. The operators of LAMBDA will occur on both sides of the equations: on the left, as parts of the LAMBDA-terms whose meanings are being specified; and on the right, as symbols in λ-notation denoting functions on elements of Pw (as defined in §1.4) - in fact only 0, x+1, x-1, and $(x \supset y,z)$ will be used, as $x(y)$ and $\lambda x.\tau$ might be confused with application and abstractions in λ-notation.

Our concept of types in LAMBDA will be made precise by the semantic function $\mathcal{E}$ below, but we can explain it informally as follows:

There are three types: integers, functions and "puns".

0 denotes an integer.

If x denotes an integer then so do x+1, x-1 (and even 0-1 (=⊥))

$x \supset y,z$ denotes a pun unless x denotes an integer.

$x(y)$ denotes a pun unless x denotes a function.

$\lambda x.\tau$ denotes a function.

The main purpose is the separation of integers and functions - in the original semantics an integer can be weaker than a function, making it impossible to distinguish them continuously. In practice puns are used rather rarely, but they do seem to be necessary for the Definability Theorem (1.4.4) to hold. Their values are dependent on the properties of the eunumerations e_n and (n,m), defined in §1.3.

We shall denote the "typed" semantic function for LAMBDA by $\mathcal{E}$ Its semantic equations are given in Table 1, where we give also a

descriptive syntax for LAMBDA, and a description of the domains used. The functions *Fun* and *Graph* are as defined in §1.3. Note that $x = 0$ is a continuous function on **Int** (but not on **Pw**).

Definition

We say that $x \in$ **Pw** is $\mathcal{E}$-*LAMBDA-definable* iff there is a LAMBDA-term ε such that

$$x = (Bare \circ \mathcal{E})[\![\varepsilon]\!](\perp_{Env}).$$

The above definition reflects the fact that our main interest lies in elements of **Pw**, rather than of **D**. Before proceeding to extend LAMBDA, we wish to show that $\mathcal{E}$-LAMBDA-definability is equivalent to LAMBDA-definability in the ordinary sense.

Table 2 gives the semantic equations for $\mathcal{D}$, which is equivalent to the original semantics for LAMBDA (as given in §1.4). We shall consistently use a prime (') to distinguish elements of $\mathcal{D}$-domains from elements of $\mathcal{E}$-domains. As in Table 1, 0, $x+1$, $x-1$ and $x \supset y,z$ are used in λ-notation to denote elements of **Pw**.

Definition

We say that $x \in$ **Pw** is $\mathcal{D}$-*LAMBDA-definable* iff there is a LAMBDA-term ε such that $x = \mathcal{D}[\![\varepsilon]\!](\perp_{Env'})$.

Clearly $\mathcal{D}$ does not say anything new, and $\mathcal{D}$-LAMBDA-definability is the same as ordinary LAMBDA-definability of elements of **Pw**. The purpose of giving $\mathcal{D}$ in semantic equations is to facilitate the analysis of $\mathcal{E}$.

Intuitively it is quite obvious that $\mathcal{D}$- and $\mathcal{E}$-LAMBDA are equivalent. Unfortunately the proof of this seems to need some rather high-powered results due to Milne [4] and (independently) Reynolds [23]. However we take comfort from the fact that

TABLE 1: $\mathcal{E}$-LAMBDA

Syntax:

$\varepsilon \in \mathbf{Exp}$ $\quad \varepsilon ::= \iota \mid 0 \mid \varepsilon + 1 \mid \varepsilon - 1 \mid$

$\varepsilon \supset \varepsilon, \varepsilon \mid \varepsilon(\varepsilon) \mid \lambda\iota.\ \varepsilon$

$\iota \in \mathbf{I}$ $\quad$ identifiers $\quad$ (variables of LAMBDA)

$\mathcal{E}$-Domains:

$\delta \in \mathbf{D} = \mathbf{Int} + [\mathbf{D} \to \mathbf{D}] + \mathbf{Pw}$

$\nu \in \mathbf{Int} = \mathit{range(int)} \quad (\subseteq \mathbf{Pw})$ $\quad$ (defined in §1.5)

$\rho \in \mathbf{Env} = [\mathbf{I} \to \mathbf{D}]$

$\mathcal{E}$-Semantics: $\quad \mathcal{E} \in \mathbf{Exp} \to \mathbf{Env} \to \mathbf{D}$

$$\mathcal{E}[\![\iota]\!]\rho = \rho[\![\iota]\!]$$

$$\mathcal{E}[\![0]\!]\rho = \mathit{InN}(0)$$

$$\mathcal{E}[\![\varepsilon + 1]\!]\rho = \mathit{IsN}(\mathcal{E}[\![\varepsilon]\!]\rho) \to \mathit{InN}(\mathit{OnN}(\mathcal{E}[\![\varepsilon]\!]\rho) + 1), \mathit{InP}(\mathit{Bare}(\mathcal{E}[\![\varepsilon]\!]\rho) + 1)$$

$$\mathcal{E}[\![\varepsilon - 1]\!]\rho = \mathit{IsN}(\mathcal{E}[\![\varepsilon]\!]\rho) \to \mathit{InN}(\mathit{OnN}(\mathcal{E}[\![\varepsilon]\!]\rho) - 1), \mathit{InP}(\mathit{Bare}(\mathcal{E}[\![\varepsilon]\!]\rho) - 1)$$

$$\mathcal{E}[\![\varepsilon_1 \supset \varepsilon_2, \varepsilon_3]\!]\rho = \mathit{IsN}(\mathcal{E}[\![\varepsilon_1]\!]\rho) \to ((\mathit{OnN}(\mathcal{E}[\![\varepsilon_1]\!]\rho)=0) \to \mathcal{E}[\![\varepsilon_2]\!]\rho, \mathcal{E}[\![\varepsilon_3]\!]\rho), \mathit{InP}(\mathit{Bare}(\mathcal{E}[\![\varepsilon_1]\!]\rho) \supset \mathit{Bare}(\mathcal{E}[\![\varepsilon_2]\!]\rho), \mathit{Bare}(\mathcal{E}[\![\varepsilon_3]\!]\rho))$$

$$\mathcal{E}[\![\varepsilon_1(\varepsilon_2)]\!]\rho = \mathit{IsF}(\mathcal{E}[\![\varepsilon_1]\!]\rho) \to \mathit{OnF}(\mathcal{E}[\![\varepsilon_1]\!]\rho)(\mathcal{E}[\![\varepsilon_2]\!]\rho), \mathit{InP}(\mathit{Fun}(\mathit{Bare}(\mathcal{E}[\![\varepsilon_1]\!]\rho))(\mathit{Bare}(\mathcal{E}[\![\varepsilon_2]\!]\rho)))$$

$$\mathcal{E}[\![\lambda\iota.\ \varepsilon]\!]\rho = \mathit{InF}(\lambda\delta.\ \mathcal{E}[\![\varepsilon]\!]\rho[\delta/\iota])$$

TABLE 1 (ctd.)

The primitive functions:

$$IsN \in D \to Bool, \quad IsF \in D \to Bool, \quad IsP \in D \to Bool,$$
$$OnN \in D \to Int, \quad OnF \in D \to [D \to D], \quad OnP \in D \to Pw,$$
$$InN \in Int \to D, \quad InF \in [D \to D] \to D, \quad InP \in Pw \to D,$$

are such that, if I_A denotes the identity function on any domain A, then we have

$$OnN \circ InN = I_{Int}, \quad OnF \circ InF = I_{D \to D}, \quad OnP \circ InP = I_{Pw},$$

and

$$(\lambda\delta.\ IsN(\delta) \to (InN \circ OnN)(\delta),$$
$$IsF(\delta) \to (InF \circ OnF)(\delta),$$
$$IsP(\delta) \to (InP \circ OnP)(\delta),\ \perp) = I_D.$$

(The existence of these functions comes from the definition of the separated sum of domains in §1.1.)

The function $Bare \in D \to Pw$ is defined recursively by

$$Bare(\delta) = IsN(\delta) \to OnN(\delta),$$
$$IsF(\delta) \to Graph(Bare \circ OnF(\delta) \circ InP),$$
$$IsP(\delta) \to OnP(\delta),$$
$$\perp_{Pw}.$$

TABLE 2: $\mathcal{D}$-LAMBDA

Syntax:

$\varepsilon \in Exp \quad \varepsilon ::= \iota \mid 0 \mid \varepsilon + 1 \mid \varepsilon - 1 \mid$

$\varepsilon \supset \varepsilon, \varepsilon \mid \varepsilon(\varepsilon) \mid \lambda\iota.\ \varepsilon$

$\iota \in I$ identifiers (variables of LAMBDA)

$\mathcal{D}$-Domains:

$\delta' \in D' = Pw$

$\rho' \in Env' = [I \to D']$

$\mathcal{D}$-Semantics: $\mathcal{D} \in Exp \to Env' \to D'$

$\mathcal{D}[\![\iota]\!]\rho' = \rho'[\![\iota]\!]$

$\mathcal{D}[\![0]\!]\rho' = 0$

$\mathcal{D}[\![\varepsilon + 1]\!]\rho' = \mathcal{D}[\![\varepsilon]\!]\rho' + 1$

$\mathcal{D}[\![\varepsilon - 1]\!]\rho' = \mathcal{D}[\![\varepsilon]\!]\rho' - 1$

$\mathcal{D}[\![\varepsilon_1 \supset \varepsilon_2, \varepsilon_3]\!]\rho' = \mathcal{D}[\![\varepsilon_1]\!]\rho' \supset \mathcal{D}[\![\varepsilon_2]\!]\rho', \mathcal{D}[\![\varepsilon_3]\!]\rho'$

$\mathcal{D}[\![\varepsilon_1(\varepsilon_2)]\!]\rho' = Fun(\mathcal{D}[\![\varepsilon_1]\!]\rho')(\mathcal{D}[\![\varepsilon_2]\!]\rho')$

$\mathcal{D}[\![\lambda\iota.\ \varepsilon]\!]\rho' = Graph(\lambda\delta'.\ \mathcal{D}[\![\varepsilon]\!]\rho'[\delta'/\iota])$

Reynolds' use of the results in [23] shows the equivalence of "direct" and "continuation" semantics for the λ-calculus (augmented with "call-by-value" abstractions) - that equivalence is just as intuitively obvious as ours.

Definitions

D $-\!\!\times$ D' denotes the *set of relations* between the domains D and D', i.e. the power set of D $\times$ D'.

η: $x \to x'$ denotes $\langle x, x' \rangle \in \eta$, where $\eta \in$ D $-\!\!\times$ D'.

A relation $\eta \in$ D $-\!\!\times$ D' is said to be *directed complete* iff η: $x \to x'$ whenever x and x' are the least upper bounds of two directed sequences $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ and $x'_0 \sqsubseteq x'_1 \sqsubseteq \ldots$ such that η: $x_n \to x'_n$ for all n.

Lemma 2.2.1

There exists a unique directed complete relation $\eta \in$ D $-\!\!\times$ D' such that η: $x \to x'$ iff

(1) $\mathit{Bare}(x) = x'$, and either

(2) $\mathit{IsN}(x)$, and $\mathit{OnN}(x) = x' \neq \top_{\mathrm{Int}}$, or

(3) $\mathit{IsF}(x)$, and for all $y \in$ D, $y' \in$ D'

η: $y \to y'$ implies η: $\mathit{OnF}(x)(y) \to \mathit{Fun}(x')(y')$, or

(4) $\mathit{IsP}(x)$, and $\mathit{OnP}(x) = x'$, or

(5) $x = \bot_D$, and $x' = \bot_{D'}$.

Proof

The proof of this Lemma is a straightforward application of the results given by Reynolds in [23], on noting that *Fun* and *Graph* give a correspondence between Pw and [Pw $\to$ Pw]. We omit the proof here, as it would involve us in much extra notation. □

The following Proposition establishes the relationship betweer $\mathcal{D}$ and $\mathcal{E}$, using the definitions in Tables 1 and 2.

Proposition 2.2.2

For any LAMBDA-term ε, $\mathcal{P}[\![\varepsilon]\!]$ holds; where $\mathcal{P}[\![\varepsilon]\!]$ iff η: $\mathcal{E}[\![\varepsilon]\!]\rho \to \mathcal{D}[\![\varepsilon]\!]\rho$ whenever $\rho \in Env$ and $\rho' \in Env'$ are such that η: $\rho[\![\iota]\!] \to \rho'[\![\iota]\!]$ for all $\iota \in I$.

Proof

By structural induction on ε. Let ε_0, ρ and ρ' be given.

(6) Suppose $\mathcal{P}[\![\varepsilon]\!]$ for all components ε of ε_0

(7) Suppose that η: $\rho[\![\iota]\!] \to \rho'[\![\iota]\!]$ for all $\iota \in I$.

Let a, a' denote $\mathcal{E}[\![\varepsilon_0]\!]\rho$, $\mathcal{D}[\![\varepsilon_0]\!]\rho'$ respectively. We show

(8) η: $a \to a'$, by cases.

Case $[\![\varepsilon_0]\!] = [\![\iota]\!]$: Then $a = \rho[\![\iota]\!]$, $a' = \rho'[\![\iota]\!]$, and (8) follows from (7).

Case $[\![\varepsilon_0]\!] = [\![0]\!]$: Then $a = InN(0)$, $a' = 0$, and (8) follows from $Bare(a) = 0$ and (2) above.

Case $[\![\varepsilon_0]\!] = [\![\varepsilon+1]\!]$: Let $x, x' = \mathcal{E}[\![\varepsilon]\!]\rho, \mathcal{D}[\![\varepsilon]\!]\rho'$.

Subcase $IsN(x) = \perp_{Bool}$: Then $x = \perp_D$, so $x' = \perp_D$, by (6) and (5), hence $a = \perp_D$, $a' = \perp_D$, and (8) follows from $Bare(a) = a'$ and (5).

Subcase $IsN(x) = true$: Then $OnN(x) = Bare(x) = x'$, so $a = InN(x'+1)$, $a' = x'+1$, and (8) follows since $x'+1 \neq \top$.

Subcase $IsN(x) = false$: Then $a = InP(Bare(x) + 1)$, $a' = x' + 1$, so (8) follows from (6).

Subcase $IsN(x) = \top_{Bool}$ cannot occur, since η: $x \to x'$.

Case $[\![\varepsilon_0]\!] = [\![\varepsilon - 1]\!]$: Similar to $[\![\varepsilon + 1]\!]$ above.

Case $[\![\varepsilon_0]\!] = [\![\varepsilon_1 \supset \varepsilon_2, \varepsilon_3]\!]$: Let $x_i, x_i' = \mathcal{E}[\![\varepsilon_i]\!]\rho, \mathcal{D}[\![\varepsilon_i]\!]\rho'$ $(i = 1,2,3)$

Subcase $IsN(x_1) = \perp$: As for $[\![\varepsilon + 1]\!]$.

Subcase $IsN(x_1) = true$: Then $a = ((OnN(x_1) = 0) \to x_2, x_3)$, $a' = (x_1' \supset x_2', x_3')$. From η: $x_1 \to x_1'$ we get $x_1' \neq \top_{Int}$, so

$a' = \perp_D$, x_2' or x_3' depending on x_1'. Hence (8) follows from (5) if $a' = \perp_D$, otherwise from (6).

Subcase $IsN(x_1) = false$: Then $a = InP(Bare(x_1) \supset Bare(x_2), Bare(x_3))$, so (6) gives $a = InP(x_1' \supset x_2', x_3')$, therefore $Bare(a) = (x_1' \supset x_2', x_3') = a'$, and (8) follows from (4).

Case $[\![\varepsilon_0]\!] = [\![\varepsilon_1(\varepsilon_2)]\!]$: This is the case which uses the recursive natur of η. Let $x_i, x_i' = \mathcal{E}[\![\varepsilon_i]\!]\rho, \mathcal{D}[\![\varepsilon_i]\!]\rho'$ $(i = 1,2)$.

Subcase $IsF(x) = \perp$: Similar to $IsN(x) = \perp$ in case $[\![\varepsilon+1]\!]$.

Subcase $IsF(x) = true$: Then $a = OnF(x_1)(x_2)$, and $a' = Fun(x_1')(x_2')$. By (6) $\eta: x_i \to x_i'$ $(i = 1,2)$, so by (3) we get $\eta: a \to a'$.

Subcase $IsF(x) = false$: Then $a' = InP(Fun(Bare(x_1))(Bare(x_2)))$, and $a' = Fun(x_1')(x_2')$, so (8) follows from (6) and (5).

Case $[\![\varepsilon_0]\!] = [\![\lambda\iota.\varepsilon]\!]$: $IsF(a) = true$, hence

$$Bare(a) = Graph(Bare \circ OnF(a) \circ InP)$$
$$= Graph(\lambda\delta'.Bare(\mathcal{E}[\![\varepsilon]\!]\rho[InP(\delta')/\iota]))$$

For any δ', $\eta: InP(\delta') \to \delta$ by (4), so we have

$\eta: (\rho[InP(\delta')/\iota])(\iota_0) \to (\rho[\delta'/\iota])(\iota_0)$ for all $\iota_0 \in I$,

so by (6) $\eta: \mathcal{E}[\![\varepsilon]\!]\rho[InP(\delta')/\iota] \to \mathcal{D}[\![\varepsilon]\!]\rho[\delta'/\iota]$,

giving $Bare(a) = Graph(\lambda\delta'.\ \mathcal{D}[\![\varepsilon]\!]\rho[\delta'/\iota]) = a'$.

To show (3) for a, a' let y, y' be such that $\eta: y \to y'$.

Then $OnF(a)(y) = \mathcal{E}[\![\varepsilon]\!]\rho[y/\iota]$,

$$Fun(a')(y') = Fun(Graph(\lambda\delta',\ \mathcal{D}[\![\varepsilon]\!]\rho'[\delta'/\iota]))(y')$$
$$= \mathcal{D}[\![\varepsilon]\!]\rho'[y'/\iota]$$

and (6) gives $\eta: \mathcal{E}[\![\varepsilon]\!]\rho[y/\iota] \to \mathcal{D}[\![\varepsilon]\!]\rho'[y'/\iota]$,

so (8) follows.

As there are no more cases, we conclude $\mathcal{P}[\![\varepsilon_0]\!]$. The principle of structural induction implies that $\mathcal{P}[\![\varepsilon]\!]$ for any ε. Q.E.D. □

Corollary 2.2.3

For any LAMBDA-term ε,

$$\eta\colon\ \mathcal{E}[\![\varepsilon]\!](\perp_{Env}) \to \mathcal{D}[\![\varepsilon]\!](\perp_{Env'}).$$

Proof Immediate. □

Corollary 2.2.4

For any LAMBDA-term ε,

$$Bare(\mathcal{E}[\![\varepsilon]\!](\perp_{Env})) = \mathcal{D}[\![\varepsilon]\!](\perp_{Env'}).$$

Proof Immediate. □

This last corollary is a statement of the equivalence of $\mathcal{E}$- and $\mathcal{D}$- LAMBDA-definability, and assures us that we do not lose any expressive power by using the $\mathcal{E}$-semantics for LAMBDA.

We are now able to extend LAMBDA to a language called LAMA, which includes operators for testing types. For convenience we include numerals ν_0, ν_1, ... in LAMA, and assume that ν_0 is written as 0, ν_1 as 1, etc. The syntax and semantics of LAMA are given in Table 3.

Proposition 2.2.5

For any LAMBDA-term ε, and $\rho \in Env$,

$$Bare(\mathcal{E}[\![\mathbf{isint}\ \varepsilon]\!](\rho)) = \begin{cases} 0 & \text{if } IsN(\mathcal{E}[\![\varepsilon]\!]\rho) = true, \\ 0+1 & \text{if } IsN(\mathcal{E}[\![\varepsilon]\!]\rho) = false, \\ \perp & \text{otherwise (when } \mathcal{E}[\![\varepsilon]\!]\rho = \perp_D). \end{cases}$$

Similar results hold for **isfun** and **ispun**.

Proof

By calculation from the definitions of $\mathcal{E}$ and *Bare*. □

We see that, in LAMA, "pun-less" integers and functions can be distinguished both from each other and from the value of **pun**.

TABLE 3: LAMA

Syntax:

$\varepsilon \in$ ExpA $\varepsilon ::=$ ι | ν_i | $\varepsilon + 1$ | $\varepsilon - 1$ |
$\varepsilon \supset \varepsilon, \varepsilon$ | $\varepsilon(\varepsilon)$ | $\lambda\iota.\ \varepsilon$ |
isint ε | isfun ε | ispun ε | pun

$\iota \in$ I identifiers

$\nu_i \in$ Num numerals 0,1,2, ...

Domains: As for $\mathcal{E}$-LAMBDA (Table 1).

Semantics:

As for $\mathcal{E}$-LAMBDA (Table 1), with the addition of:

$$\mathcal{E}[\![\nu_i]\!]\rho = InN(i)$$

$$\mathcal{E}[\![\text{isint } \varepsilon]\!]\rho = InN(\ IsN(\mathcal{E}[\![\varepsilon]\!]\rho) \rightarrow 0,\ (0+1)\ )$$

$$\mathcal{E}[\![\text{isfun } \varepsilon]\!]\rho = InN(\ IsF(\mathcal{E}[\![\varepsilon]\!]\rho) \rightarrow 0,\ (0+1)\ )$$

$$\mathcal{E}[\![\text{ispun } \varepsilon]\!]\rho = InN(\ IsP(\mathcal{E}[\![\varepsilon]\!]\rho) \rightarrow 0,\ (0+1)\ )$$

$$\mathcal{E}[\![\text{pun}]\!]\rho = InP(\ \perp_{pw}\ )$$

§2.3 LAMB

LAMB augments the generality of LAMA with convenient notation for tuples and commonly-needed operations. We shall define its semantics by giving a syntactic function $\mathcal{F}$ from LAMB to LAMA, and using the semantics of LAMA as described by $\mathcal{E}$ in the last section - thus the semantic function for LAMB will be $\mathcal{E} \circ \mathcal{F}$. In order to show the generality of LAMB we shall define another syntactic function $\mathcal{G}$ from LAMBDA to LAMB, such that $\mathcal{E} \circ \mathcal{F} \circ \mathcal{G}$ is equivalent to $\mathcal{E}$ on LAMBDA.

Unfortunately there does not seem to be a compact, readable and formally defined notation for specifying syntactic mappings. How, then, shall we define $\mathcal{F}$ and $\mathcal{G}$? We note that the notation of semantic equations is both compact and readable, and so we shall extend it to allow the specification of syntactic instead of semantic values - in fact one could argue the usefulness of this extension in the general context of mathematical semantics. We hope that the reader's familiarity with semantic equations will enable him to consider our definitions of $\mathcal{F}$ and $\mathcal{G}$ to be rigorous, for (as in the previous section) we shall not attempt to define our notation formally.

Our extension of semantic equations is quite simple: we allow compound phrases of a language to be denoted on the right-hand side of an equation by the notation already used on the left-hand side, i.e. by a sequence of terminal and non-terminal symbols from a descriptive BNF grammar for the language. As usual the non-terminal symbols ("meta-variables") occurring on the right-hand side of an equation denote components of the phrase specified on the left-hand side, and may be used in constructing a new phrase; but components of constructed phrases may also be compound - enclosed by "⟦" and

"⟧" to avoid ambiguity - or else denoted in λ-notation and enclosed by "{" and "}". For example, if ε_1, ε_2 are meta-variables and $\mathcal{A}⟦\varepsilon_3⟧$ denotes a phrase, then $⟦⟦\varepsilon_1 + \varepsilon_2⟧ - \{\mathcal{A}⟦\varepsilon_3⟧\}⟧$ could denote a compound phrase of a suitable language.

For the notation to be unambiguous we require that no two phrases of the grammar "reducible" to the same non-terminal become identical when all their non-terminal symbols are replaced by some unused mark. E.g. ε ::= ...| - ε | ε - ε | ... would be allowed, whereas ε ::= ... | φ(ε) | ε(ν) | φ | ν | ... would not.

Instances of primitive phrases of the language (such as identifiers and constants) may be denoted by themselves. To improve readability, the brackets ⟦ and ⟧ are omitted whenever they can be inferred (unambiguously) from the context by reference to the grammar. For example, if the grammar contains ε ::= ... | ε(ε) | ... , but *not* ε ::= ... | (ε) | ... , then $⟦⟦\varepsilon_1(\varepsilon_2)⟧(⟦\varepsilon_3(\varepsilon_2)⟧)⟧$ may be written as $⟦\varepsilon_1(\varepsilon_2)(\varepsilon_3(\varepsilon_2))⟧$.

Now to define LAMB. A descriptive grammar, and the specification of a syntactic mapping $\mathcal{F}$ from LAMB to LAMA, are given in Table 4. The following comments are intended to be read in parallel with the Table.

TABLE 4: LAMB

Syntax:

$\varepsilon \in$ ExpB $\varepsilon ::= \lambda\beta.\ \varepsilon \mid$ val $\beta.\ \varepsilon \mid$ fix $\beta.\ \varepsilon \mid \varepsilon \to \varepsilon, \varepsilon \mid$
$\varepsilon\ \omega\ \varepsilon \mid \omega\ \varepsilon \mid \varepsilon(\varepsilon) \mid \langle \varepsilon, \ldots, \varepsilon \rangle \mid \langle\rangle \mid$
$\iota_i \mid \nu_i \mid \sigma_i \mid$ true $\mid$ false $\mid$ err

$\beta \in$ Bvs $\beta ::= \iota_i \mid \langle \beta, \ldots, \beta \rangle \mid \langle\rangle$

$\omega \in Op_2$ $\omega ::= \underline{\circ} \mid \underline{*} \mid \| \mid + \mid - \mid x \mid$
$= \mid \leq \mid \vee \mid \wedge \mid$ cat $\mid \downarrow$

$\omega \in Op_1$ $\omega ::=$ pair $\mid \sim \mid$ dim $\mid$ isatom $\mid$ imp

$\iota_i \in$ I identifiers $\iota_5, \iota_6, \ldots$ of LAMA

$\nu_i \in$ Num numerals $\nu_0, \nu_1, \ldots$ of LAMA

$\sigma_i \in$ String strings $\sigma_0, \sigma_1, \ldots$

Domains:

$\alpha \in$ ExpA (as defined in Table 3)

Mapping: $\mathcal{F} \in$ ExpB $\to$ ExpA

$$\mathcal{F}[\![\lambda\beta.\ \varepsilon]\!] = \mathcal{B}[\![\beta]\!](\mathcal{F}[\![\varepsilon]\!])$$
$$\mathcal{F}[\![\text{val } \beta.\ \varepsilon]\!] = [\![\{Strict\}(\{\mathcal{B}[\![\beta]\!](\mathcal{F}[\![\varepsilon]\!])\})]\!]$$
$$\mathcal{F}[\![\text{fix } \beta.\ \varepsilon]\!] = [\![\{Fix\}(\{\mathcal{B}[\![\beta]\!](\mathcal{F}[\![\varepsilon]\!])\})]\!]$$
$$\mathcal{F}[\![\varepsilon_1 \to \varepsilon_2, \varepsilon_3]\!] = [\![\{\mathcal{F}[\![\varepsilon_1]\!]\} \supset \{\mathcal{F}[\![\varepsilon_2]\!]\}, \{\mathcal{F}[\![\varepsilon_3]\!]\}]\!]$$
$$\mathcal{F}[\![\varepsilon_1\ \omega\ \varepsilon_2]\!] = [\![\{\mathcal{O}_2[\![\omega]\!]\}(\{\mathcal{F}[\![\varepsilon_1]\!]\})(\{\mathcal{F}[\![\varepsilon_2]\!]\})]\!]$$
$$\mathcal{F}[\![\omega\ \varepsilon]\!] = [\![\{\mathcal{O}_1[\![\omega]\!]\}(\{\mathcal{F}[\![\varepsilon]\!]\})]\!]$$
$$\mathcal{F}[\![\varepsilon_1(\varepsilon_2)]\!] = [\![\{\mathcal{F}[\![\varepsilon_1]\!]\}(\{\mathcal{F}[\![\varepsilon_2]\!]\})]\!]$$

TABLE 4 (ctd.)

$$\mathcal{F}[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!] = [\![\lambda\iota_0.\ [\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_0)(\nu_0)]\!] \supset \nu_n,\ [\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_0)(\nu_1)]\!] \supset \{\mathcal{F}[\![\varepsilon_1]\!]\},\ \ldots\ [\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_0)(\nu_n)]\!] \supset \{\mathcal{F}[\![\varepsilon_n]\!]\},\ \text{pun}]\ldots]]]$$

$$\mathcal{F}[\![\langle\rangle]\!] = [\![\lambda\iota_0.\ [\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_0)(\nu_0)]\!] \supset \nu_0,\ \text{pun}]\!]$$

$$\mathcal{F}[\![\iota_i]\!] = [\![\iota_i]\!]$$

$$\mathcal{F}[\![\nu_i]\!] = [\![\nu_i]\!]$$

$$\mathcal{F}[\![\sigma_i]\!] = [\![\nu_i]\!]$$

$$\mathcal{F}[\![\text{true}]\!] = [\![\nu_0]\!]$$

$$\mathcal{F}[\![\text{false}]\!] = [\![\nu_1]\!]$$

$$\mathcal{F}[\![\text{err}]\!] = [\![\text{pun}]\!]$$

$$\mathcal{B} \in \text{Bvs} \to \text{ExpA} \to \text{ExpA}$$

$$\mathcal{B}[\![\iota_i]\!](\alpha) = [\![\lambda\iota_i.\{\alpha\}]\!]$$

$$\mathcal{B}[\![\langle \beta_1,\ldots,\beta_n\rangle]\!](\alpha) = [\![\lambda\iota_0.\ [\![\{\mathcal{B}[\![\beta_1]\!](\mathcal{B}[\![\beta_2]\!](\ldots(\mathcal{B}[\![\beta_n]\!](\alpha))\ldots))\}(\iota_0(\nu_0))(\iota_0(\nu_1))\ldots(\iota_0(\nu_n))]\!]$$

$$\mathcal{B}[\![\langle\rangle]\!](\alpha) = [\![\lambda\iota_0.\ \{\alpha\}]\!]$$

$$\mathcal{O}_2 \in \text{Op}_2 \to \text{ExpA}$$

$$\mathcal{O}_2[\![\underline{\circ}]\!] = [\![\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\{Strict\}(\iota_1)(\{Strict\}(\iota_0)(\iota_2))]\!]$$

$$\mathcal{O}_2[\![\underline{*}]\!] = [\![\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\lambda\iota_3.\ \{Strict\}(\iota_1)(\iota_3(\nu_1))(\iota_3(\nu_2))]\!]\ (\{Strict\}(\iota_0)(\iota_2))]\!]$$

$$\mathcal{O}_2[\![\|]\!] = [\![\lambda\iota_0.\lambda\iota_1.\ [\![\iota_0(\iota_1)]\!]$$

TABLE 4 (ctd.)

$$\mathcal{O}_2[\![+]\!] = [\![\{Fix\}(\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\iota_2 \supset \iota_1,\ [\![\iota_0(\iota_1+1)(\iota_2-1)]\!]]\!])]\!]$$

$$\mathcal{O}_2[\![-]\!] = [\![\{Fix\}(\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\iota_2 \supset \iota_1,\ [\![\iota_0(\iota_1-1)(\iota_2-1)]\!]]\!])]\!]$$

$$\mathcal{O}_2[\![\times]\!] = [\![\lambda\iota_3.\ [\![[\![\{Fix\}(\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\iota_2 \supset \iota_1,\ [\![\iota_0(\{\mathcal{O}_2[\![+]\!]\}(\iota_1)(\iota_3))(\iota_2-1)]\!]]\!])]\!](\nu_0)]\!]]\!]$$

$$\mathcal{O}_2[\![=]\!] = [\![\lambda\iota_3.\lambda\iota_4.\ [\![[\![\text{ispun } \iota_3]\!] \supset [\![\text{ispun } \iota_4]\!],\ [\![[\![\text{ispun } \iota_4]\!] \supset \nu_1,\ [\![[\![\{Fix\}(\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\iota_2 \supset [\![\iota_1 \supset \nu_0,\ \nu_1]\!],\ [\![\iota_1 \supset \nu_1,\ [\![\iota_0(\iota_1-1)(\iota_2-1)]\!]]\!])]\!]\ (\iota_3)(\iota_4)]\!]]\!]$$

$$\mathcal{O}_2[\![\leq]\!] = [\![\{Fix\}(\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\ [\![\iota_2 \supset [\![\iota_1 \supset \nu_0,\ \nu_1]\!],\ [\![\iota_1 \supset \nu_0,\ [\![\iota_0(\iota_1-1)(\iota_2-1)]\!]]\!])]\!]$$

$$\mathcal{O}_2[\![\vee]\!] = [\![\lambda\iota_0.\lambda\iota_1.\ [\![\iota_0 \supset \nu_0,\ \iota_1]\!]]\!]$$

$$\mathcal{O}_2[\![\wedge]\!] = [\![\lambda\iota_0.\lambda\iota_1.\ [\![\iota_0 \supset \iota_1,\ \nu_1]\!]]\!]$$

$$\mathcal{O}_2[\![\text{cat}]\!] = [\![\lambda\iota_0.\lambda\iota_1.\ \lambda\iota_2.\ [\![\iota_2 \supset \{\mathcal{O}_2[\![+]\!]\}(\{\mathcal{O}_1[\![\text{dim}]\!]\}(\iota_0))(\{\mathcal{O}_1[\![\text{dim}]\!]\}(\iota_1)),\ [\![[\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_2)(\{\mathcal{O}_1[\![\text{dim}]\!]\}(\iota_0)]\!] \supset \iota_0(\iota_2),\ [\![\iota_1(\{\mathcal{O}_2[\![-]\!]\}(\iota_2)(\mathcal{O}_1[\![\text{dim}]\!](\iota_0)))]\!]]\!]]\!]$$

$$\mathcal{O}_2[\![\downarrow]\!] = [\![\lambda\iota_0.\lambda\iota_1.\ [\![\iota_0(\iota_1)]\!]]\!]$$

TABLE 4 (ctd.)

$\mathcal{O}_1 \in Op_1 \to ExpA$

$\mathcal{O}_1[\![\text{pair}]\!] = [\![\lambda\iota_0.\lambda\iota_1.$
$\lambda\iota_2.\ [\![[\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_2)(\nu_0)]\!] \supset \nu_2,$
$[\![[\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_2)(\nu_1)]\!] \supset \iota_0,$
$[\![[\![\{\mathcal{O}_2[\![\leq]\!]\}(\iota_2)(\nu_2)]\!] \supset \iota_1, \text{pun}]\!]]\!]]\!]$

$\mathcal{O}_1[\![\sim]\!] = [\![\lambda\iota_0.\ [\![\iota_0 \supset \nu_1, \nu_0]\!]]\!]$

$\mathcal{O}_1[\![\text{dim}]\!] = [\![\lambda\iota_0.\ \iota_0(\nu_0)]\!]$

$\mathcal{O}_1[\![\text{isatom}]\!] = [\![\lambda\iota_0.\ \text{isint}\ \iota_0]\!]$

$\mathcal{O}_1[\![\text{imp}]\!] \in ExpA$ denotes an unspecified function from integers to arbitrary values.

Fix $= [\![\lambda\iota_0.\lambda\iota_1.\ [\![\iota_0(\iota_1(\iota_1))]\!](\iota_0(\iota_1(\iota_1)))]\!]$

Strict $= [\![\lambda\iota_0.\lambda\iota_1.\ [\![\text{ispun}\ \iota_1]\!] \supset \text{pun}, [\![\iota_0(\iota_1)]\!]]\!]$

⟦λβ. ε⟧ denotes functional abstraction, as in LAMA, but β may be a tuple of βs, enabling the components of a tuple of known length to be named. Note that ⟦⟦λ⟨$\beta_1,\ldots,\beta_n$⟩. ε⟧(⟨$\varepsilon_1,\ldots,\varepsilon_m$⟩)⟧ is valid when $m \geq n$.

⟦ val β. ε⟧ denotes "call-by-value" abstraction. Hence ⟦⟦ val β. ε⟧(ε_0)⟧ denotes ⊥ whenever ε_0 denotes ⊥.

⟦ fix β. ε⟧ denotes the minimal fixed point of ε with respect to β. It is equivalent to $Y(\lambda\beta.\ \varepsilon)$ in λ-notation.

⟦$\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3$⟧: here "→" is used in preference to "⊃", to conform to current usage in semantic equations.

⟦$\varepsilon_1\ \omega\ \varepsilon_2$⟧: various diadic operations.

⟦$\omega\ \varepsilon$⟧: various monadic operations.

⟦$\varepsilon_1(\varepsilon_2)$⟧: functional application.

⟦⟨$\varepsilon_1,\ldots,\varepsilon_n$⟩⟧ denotes a tuple, which is represented as a function on the integers, whose value at i is n when $i = 0$, the value of ε_i when $1 \leq i \leq n$, and pun when $i > n$.

⟦⟨⟩⟧ denotes the null tuple.

⟦ι_i⟧: it is postulated that $i \geq 5$, to avoid accidental binding of variables in the mapping from ExpA to ExpB.

⟦ν_i⟧: integers (non-negative).

⟦σ_i⟧: strings, included for mnemonic purposes later.

⟦true⟧, ⟦false⟧: used in logical operations.

⟦err⟧: distinct from "pun-less" values.

⟦pair⟧ is included in LAMB for pragmatic reasons, explained in Chapter 4.

⟦~⟧ : logical negation.

⟦dim⟧ finds the dimension of a tuple.

⟦isatom⟧ will be used to test whether a value is an integer or a tuple, but it also distinguishes integers from functions and the value of ⟦err⟧.

⟦imp⟧ provides access to various functions which it will be convenient to use in MSL, but which are inessential for theoretical purposes. We defer further discussion of this to the comments on the definition of MSL.

⟦$\underline{\circ}$⟧ is the reverse of the usual composition operator $\circ$, and denotes a strict function (mapping $\perp$ to $\perp$) even if its operands do not. Its properties will be motivated when MSL is introduced in § 2.5.

⟦$\underline{*}$⟧ bears the same relation to * in semantic equations as $\underline{\circ}$ does to $\circ$.

⟦‖⟧ denotes ordinary application, and is included in LAMB solely for pragmatic reasons.

⟦+⟧, ⟦-⟧, ⟦×⟧: the usual arithmetic operations on (non-negative) integers.

⟦=⟧ is defined on the integers and on the value of ⟦err⟧.

⟦≤⟧ is defined on the integers.

⟦∨⟧: logical or.

⟦∧⟧: logical and.

⟦cat⟧: concatenation of tuples.

$[\![\downarrow]\!]$ denotes selection of a component of a tuple. It is used instead of the equivalent application so that the representation of tuples as functions may be forgotten.

{*Fix*} the LAMA-term for Curry's "paradoxical combinator" *y*.

{*Strict*}, as its name suggests, converts an arbitrary function into a strict one, i.e. mapping $\perp$ to $\perp$.

Intuitively it is clear that LAMB is in effect an extension of LAMBDA, and hence as powerful. In fact we can define a syntactic mapping $\mathcal{G}$ from LAMBDA to LAMB, which enables us to verify that our intuition is correct.

Let Exp be as defined in Table 1. We define $\mathcal{G}$ as follows:

$$\mathcal{G} \in \mathsf{Exp} \to \mathsf{ExpB}$$

$$\mathcal{G}[\![\iota_i]\!] = [\![\iota_{i+5}]\!]$$
$$\mathcal{G}[\![0]\!] = [\![\nu_0]\!]$$
$$\mathcal{G}[\![\varepsilon + 1]\!] = [\![\{\mathcal{G}[\![\varepsilon]\!]\} + \nu_1]\!]$$
$$\mathcal{G}[\![\varepsilon - 1]\!] = [\![\{\mathcal{G}[\![\varepsilon]\!]\} - \nu_1]\!]$$
$$\mathcal{G}[\![\varepsilon_1 \supset \varepsilon_2, \varepsilon_3]\!] = [\![\{\mathcal{G}[\![\varepsilon_1]\!]\} \supset \{\mathcal{G}[\![\varepsilon_2]\!]\}, \{\mathcal{G}[\![\varepsilon_3]\!]\}]\!]$$
$$\mathcal{G}[\![\varepsilon_1(\varepsilon_2)]\!] = [\![\{\mathcal{G}[\![\varepsilon_1]\!]\}\ (\{\mathcal{G}[\![\varepsilon_2]\!]\})]\!]$$
$$\mathcal{G}[\![\lambda\iota_i.\ \varepsilon]\!] = [\![\lambda\iota_{i+5}.\ \{\mathcal{G}[\![\varepsilon]\!]\}]\!]$$

A simple structural induction shows that for any LAMBDA term ε and environment $\rho \in \mathsf{Env}$, we have

(1) $$(\mathcal{E} \circ \mathcal{F} \circ \mathcal{G})[\![\varepsilon]\!]\rho = \mathcal{E}[\![\varepsilon]\!]\rho,$$

where $\mathcal{E}$, $\mathcal{F}$ and Env are as defined in Tables 1 and 4. If we declare a value $x \in \mathsf{Pw}$ to be LAMB-definable iff there exists a LAMB-term ε such that

(2) $$x = \mathit{Bare}((\mathcal{E} \circ \mathcal{F})[\![\varepsilon]\!](\perp_{\mathsf{Env}}))$$

with *Bare* as defined in Table 1, then 2.2.4 and (1) give us that all computable values in Pw are LAMB-definable, by the Definability Theorem (1.4.4) for LAMBDA.

We shall discuss the reduction rules and implementation of LAMB in Chapter 3. In the next sections we shall define MSL with the aid of LAMB, and to avoid the possibility of ambiguity we give an unambiguous grammar for LAMB in Appendix A - this grammar can easily be interpreted to yield the descriptive grammar of Table 4, thus enabling the reader to use the definitions of $\mathfrak{E}$ and $\mathfrak{F}$ to understand the denotation in Pw of any written LAMB-term.

§2.4 Embedding Semantic Functions in Pw

A semantic function is a mapping from phrases or parse-trees of programs to their semantic values in recursively-defined domains. As we remarked in §2.1, it will be convenient to embed semantic functions themselves in a domain, and we do this by embedding both parse-trees and values in Pw. We shall show that the induced mappings from Pw to Pw are continuous, and hence are in the domain [Pw → Pw], which can be identified (one-one) with Pw itself.

First we embed parse-trees in Pw. As we intend them to be easily denotable in LAMB, it is natural to use LAMB instead of LAMBDA in defining this embedding. We note that a parse-tree is either a terminal "leaf", or a node which consists of a label, indicating its form, and a number of component sub-trees. We identify labels with the values of strings in LAMB - usually the strings will be formed from the symbols of a descriptive grammar - and terminal leaves with the values of LAMB numerals. Then if the label of a node, say t, is identified with the value of string σ, and the component sub-trees are identified with the values of LAMB-terms $\varepsilon_1,\ldots,\varepsilon_n (n \geq 0)$, we identify t with the value of the LAMB-term $\langle \varepsilon_1,\ldots,\varepsilon_n, \sigma \rangle$. Thus the components of a node may be denoted in the same way as the elements of a tuple, using the operator ↓, and the operator isatom will distinguish a terminal leaf from a node.

Distinct parse-trees are embedded as incomparable elements of Pw, because the values of LAMB-tuples correspond to functions whose value at 0 is the length of the tuple. Note that this would not necessarily hold if parse-trees were embedded (in the same way) using Scott's tuples (described in §1.4).

By the Embedding Theorem (1.3.5) all our semantic domains can be

continuously embedded in Pw. However it is even simpler to translate a description of a domain (in the notation of §1.2) into a LAMBDA-definition of a retract of Pw, whose range (by 1.5.2) is isomorphic to the original domain.

Hence our embedding of a semantic function is a mapping from a set of incomparable elements of Pw, to Pw. The incomparable elements form a sub-domain of Pw when augmented by $\perp_{Pw}$ and $\top_{Pw}$, and clearly any monotonic mapping on this sub-domain must be continuous. By extending the embedding of a semantic function to take the values $\perp$, $\top$ at $\perp$, $\top$ respectively it therefore becomes a continuous function, and so by the Extension Theorem (1.3.4) it has a continuous extension ϵ [Pw → Pw]. The Graph Theorem (1.3.1) gives the identification of [Pw → Pw] with Pw, so our embedded semantic functions can be considered simply as elements of Pw.

Later we shall use LAMB-expressions to denote embedded semantic functions. As usual in mathematical semantics, we shall want to omit the injections and projections between sum domains and their summands - we are interested more in values than types. By doing this we shall be denoting not the semantic functions themselves but elements of Pw to which they can be continuously related (by stripping away the type information, assuming the type is known). However the omission of injections and projections cannot change "type-independent" properties of semantic functions,so no confusion should be caused by this.

So far in this Chapter we have defined LAMB, and embedded both parse-trees and semantic functions in Pw. Next we shall define and discuss the syntax of MSL, which is an extension of LAMB, and is formally defined using LAMB.

§2.5 MSL

Ideally MSL would be a precisely-defined version of the basic notation of semantic equations. However, for practical and philosophical reasons we wish to keep the semantics of MSL as simple as possible. Unfortunately it does not seem to be possible to give a sufficiently simple mathematical definition of one of the main features of semantic equations: its use of so-called *meta-variables* ranging over phrases of programs.

The difficulty in describing meta-variables mathematically comes from their double use. For example, in

(1) $$\mathcal{F}[\![\gamma_1 \; ; \; \gamma_2]\!] = \lambda\rho.\ \lambda\sigma.\ \mathcal{F}[\![\gamma_2]\!]\rho(\mathcal{F}[\![\gamma_1]\!]\rho\sigma)$$

the meta-variables γ_1 and γ_2 not only denote sub-phrases of a phrase, but also indicate that the equation only applies to phrases of a certain form (with reference to a grammar). To keep the semantics simple, MSL uses a primitive string to indicate the form of a phrase, and a list of ordinary variables (i.e. as in λ-notation) to denote its components. Thus $[\![\gamma_1; \gamma_2]\!]$ of (1) would correspond to "γ ; γ" $\langle c_1, c_2 \rangle$ in MSL.

There are two other major differences of MSL from semantic equations. The first is that semantic function definitions are more explicit, rather like those in PAL. The second is that the "..." convention is not allowed: instead operators are provided to select arbitrary components of phrases, and to iterate a function.

Apart from the above differences, MSL is very similar to semantic equations. In fact MSL even follows the informal conventions of omitting the domain specification ":D" from an abstraction "λ*x*:D. *e*", and of allowing the omission of all injections and projections between sum and summand domains - MSL is closer to LAMBDA than to λ-notation, and does not insist on a careful control of

domains. But sometimes in semantic equations there occurs a test for "membership" of a particular summand of, e.g., $D_0 + D_1$, written as $Is D_0(\delta)$ or $Is D_1(\delta)$. This is usually justified by pointing out the possibility of inserting all the relevant omitted projections and injections. MSL provides operators to do common tests, such as testing to which summand D^n of D^* a tuple belongs, or whether a value is a tuple or not, and these may be used without inserting any projections and injections.

Appendix B gives an unambiguous grammar for MSL, and Appendix C contains a LAMB-expression denoting a syntactic mapping from MSL to LAMB. Together they constitute the formal definition of MSL (relative to LAMB). In the rest of this section we shall give a less formal description of MSL, using the extension of semantic equations introduced for the description of LAMB in §2.3. We first give a descriptive grammar, which is derived from the unambiguous grammar in Appendix B. In fact Appendix B describes a "hardware representation" of MSL, rather than a "publication language" - however Appendix Z gives a correspondence between the limited character set of a computer and the character set normally used in mathematical semantics. Note that several styles of brackets may be used in MSL; we do not include them in Table 5, as they are semantically irrelevant.

The following comments are intended to be read in parallel with Table 5:

TABLE 5: MSL

Syntax:

$E \in$ Seg	$E ::=$	$\lambda\pi.\ E \mid$ def Δ E $\mid$ let Δ E $\mid$ result ε
$\Delta \in$ DefL	$\Delta ::=$	δ and ... and δ
$\delta \in$ Def	$\delta ::=$	$\iota_i\ \pi = \varepsilon \mid \beta = \varepsilon$
$\varepsilon \in$ Exp	$\varepsilon ::=$	def δ in $\varepsilon \mid$ let δ in $\varepsilon \mid \lambda\pi.\ \varepsilon \mid$ fix $\beta.\ \varepsilon \mid \varepsilon \to \varepsilon, \varepsilon \mid$ clauses ε § Γ § $\mid$ $\varepsilon\ \omega\ \varepsilon \mid \omega\ \varepsilon \mid \omega\ \varepsilon\ \varepsilon \mid \varepsilon(\varepsilon) \mid \varepsilon[\varepsilon/\varepsilon] \mid$ $\langle \varepsilon,\ldots,\varepsilon\rangle \mid \langle\rangle \mid \iota_i \mid \nu_i \mid \sigma_i \mid$ true $\mid$ false $\mid$ err
$\pi \in$ ParL	$\pi ::=$	$\beta\ \pi \mid \beta$
$\beta \in$ Bvs	$\beta ::=$	$\iota_i \mid \langle \beta,\ldots,\beta\rangle \mid \langle\rangle$
$\Gamma \in$ CaseL	$\Gamma ::=$	$\gamma\ \Gamma \mid \gamma$
$\gamma \in$ Case	$\gamma ::=$	case $\Sigma\ \beta$: $\varepsilon \mid$ case Σ: $\varepsilon \mid$ default: ε
$\Sigma \in$ StrL	$\Sigma ::=$	σ_i case $\Sigma \mid \sigma$
$\omega \in$ InOp	$\omega ::=$	$\underline{\circ} \mid \underline{*} \mid \| \mid + \mid - \mid \times \mid = \mid \leq \mid \neq \mid \vee \mid \wedge \mid \downarrow \mid$ aug $\mid$ cat $\mid$ pre
$\omega \in$ PreOp$_1$	$\omega ::=$	pair $\mid \sim \mid$ dim $\mid$ isatom $\mid$ seg $\mid$ chars $\mid$ string $\mid$ mapn $\mid$ mapt $\mid$ label $\mid$ spread
$\omega \in$ PreOp$_2$	$\omega ::=$	node
$\iota_i \in$ I	identifiers	$\iota_{10}, \iota_{11}, \ldots$ of LAMB
$\nu_i \in$ Num	numerals	$\nu_0, \nu_1, \ldots$ of LAMB
$\sigma_i \in$ String	strings	$\sigma_0, \sigma_1, \ldots$ of LAMB

TABLE 5 (ctd.)

Domains:

$\zeta \in$ ExpB (as defined in Table 4)

Mapping: $\mathcal{E} \in$ [Seg + DefL + Def + Exp] → ExpB

$$\mathcal{E}[\![\lambda\pi.\ E]\!] = \mathcal{P}[\![\pi]\!](\mathcal{E}[\![E]\!])$$

$$\mathcal{E}[\![\text{def } \Delta \text{ E}]\!] = [\![\lambda\{\mathcal{D}[\![\Delta]\!]\}.\{\mathcal{E}[\![E]\!]\}]\!](\text{fix}\{\mathcal{D}[\![\Delta]\!]\}.\ \{\mathcal{E}[\![\Delta]\!]\})]\!]$$

$$\mathcal{E}[\![\text{let } \Delta \text{ E}]\!] = [\![\lambda\{\mathcal{D}[\![\Delta]\!]\}.\{\mathcal{E}[\![E]\!]\}]\!](\{\mathcal{E}[\![\Delta]\!]\})]\!]$$

$$\mathcal{E}[\![\text{result } \varepsilon]\!] = \mathcal{E}[\![\varepsilon]\!]$$

$$\mathcal{E}[\![\delta_1 \text{ and} \ldots \text{and } \delta_n]\!] = [\![\langle\{\mathcal{E}[\![\delta_1]\!]\}, \ldots, \{\mathcal{E}[\![\delta_n]\!]\}\rangle]\!]$$

$$\mathcal{E}[\![\iota_i\ \pi = \varepsilon]\!] = \mathcal{P}[\![\pi]\!](\mathcal{E}[\![\varepsilon]\!])$$

$$\mathcal{E}[\![\beta = \varepsilon]\!] = \mathcal{E}[\![\varepsilon]\!]$$

$$\mathcal{E}[\![\text{def } \delta \text{ in } \varepsilon]\!] = [\![\lambda\{\mathcal{D}[\![\delta]\!]\}.\ \{\mathcal{E}[\![\varepsilon]\!]\}](\text{fix}\{\mathcal{D}[\![\delta]\!]\}.\ \{\mathcal{E}[\![\delta]\!]\})]\!]$$

$$\mathcal{E}[\![\text{let } \delta \text{ in } \varepsilon]\!] = [\![\lambda\{\mathcal{D}[\![\delta]\!]\}.\ \{\mathcal{E}[\![\varepsilon]\!]\}](\{\mathcal{E}[\![\delta]\!]\})]\!]$$

$$\mathcal{E}[\![\lambda\pi.\ \varepsilon]\!] = \mathcal{P}[\![\pi]\!](\mathcal{E}[\![\varepsilon]\!])$$

$$\mathcal{E}[\![\text{fix } \beta.\ \varepsilon]\!] = [\![\text{fix } \beta.\{\mathcal{E}[\![\varepsilon]\!]\}]\!]$$

$$\mathcal{E}[\![\varepsilon_1 \to \varepsilon_2, \varepsilon_3]\!] = [\![\{\mathcal{E}[\![\varepsilon_1]\!]\} \to \{\mathcal{E}[\![\varepsilon_2]\!]\}, \{\mathcal{E}[\![\varepsilon_3]\!]\}]\!]$$

$$\mathcal{E}[\![\text{clauses } \varepsilon\ \S\ \Gamma\ \$]\!] = \mathcal{C}[\![\Gamma]\!]([\![\{\mathcal{O}_1[\![\text{label}]\!]\}(\{\mathcal{E}[\![\varepsilon]\!]\})]\!])(\mathcal{E}[\![\varepsilon]\!])(\mathcal{E}[\![\text{err}]\!])$$

$$\mathcal{E}[\![\varepsilon_1\ \omega\ \varepsilon_2]\!] = [\![\{\mathcal{O}_2[\![\omega]\!]\}(\{\mathcal{E}[\![\varepsilon_1]\!]\})(\{\mathcal{E}[\![\varepsilon_2]\!]\})]\!]$$

$$\mathcal{E}[\![\omega\ \varepsilon]\!] = [\![\{\mathcal{O}_1[\![\omega]\!]\}(\{\mathcal{E}[\![\varepsilon]\!]\})]\!]$$

$$\mathcal{E}[\![\omega\ \varepsilon_1\ \varepsilon_2]\!] = [\![\{\mathcal{O}_2[\![\omega]\!]\}(\{\mathcal{E}[\![\varepsilon_1]\!]\})(\{\mathcal{E}[\![\varepsilon_2]\!]\})]\!]$$

TABLE 5 (ctd.)

$$\mathcal{E}[\![\varepsilon_1(\varepsilon_2)]\!] = [\![\{\mathcal{E}[\![\varepsilon_1]\!]\}(\{\mathcal{E}[\![\varepsilon_2]\!]\})]\!]$$

$$\mathcal{E}[\![\varepsilon_1[\varepsilon_2/\varepsilon_3]]\!] = [\![\lambda\iota_5.\ [\![\iota_5=\{\mathcal{E}[\![\varepsilon_3]\!]\}]\!] \to \{\mathcal{E}[\![\varepsilon_2]\!]\},\ [\![\{\mathcal{E}[\![\varepsilon_1]\!]\}(\iota_5)]\!]]\!]$$

$$\mathcal{E}[\![\langle\varepsilon_1,\ldots,\varepsilon_n\rangle]\!] = [\![\langle\{\mathcal{E}[\![\varepsilon_1]\!]\},\ldots,\{\mathcal{E}[\![\varepsilon_n]\!]\}\rangle]\!]$$

$$\mathcal{E}[\![\langle\rangle]\!] = [\![\langle\rangle]\!]$$

$$\mathcal{E}[\![\iota_i]\!] = [\![\iota_i]\!]$$

$$\mathcal{E}[\![\nu_i]\!] = [\![\nu_i]\!]$$

$$\mathcal{E}[\![\sigma_i]\!] = [\![\sigma_i]\!]$$

$$\mathcal{E}[\![\text{true}]\!] = [\![\text{true}]\!]$$

$$\mathcal{E}[\![\text{false}]\!] = [\![\text{false}]\!]$$

$$\mathcal{E}[\![\text{err}]\!] = [\![\text{err}]\!]$$

$$\mathcal{D} \in [\text{DefL} + \text{Def}] \to \text{Bvs}$$

$$\mathcal{D}[\![\delta_1 \text{ and} \ldots \text{and } \delta_n]\!] = [\![\langle\{\mathcal{D}[\![\delta_1]\!]\},\ldots,\{\mathcal{D}[\![\delta_n]\!]\}\rangle]\!]$$

$$\mathcal{D}[\![\iota_i\ \pi = \varepsilon]\!] = [\![\iota_i]\!]$$

$$\mathcal{D}[\![\beta = \varepsilon]\!] = [\![\beta]\!]$$

TABLE 5 (ctd.)

$$\mathcal{P} \in \text{ParL} \to \text{ExpB} \to \text{ExpB}$$

$$\mathcal{P}[\![\beta\ \pi]\!](\zeta) = [\![\lambda\beta.\ \{\mathcal{P}[\![\pi]\!](\zeta)\}]\!]$$

$$\mathcal{P}[\![\beta]\!](\zeta) = [\![\lambda\beta.\ \{\zeta\}]\!]$$

$$\mathcal{C} \in [\text{CaseL+Case}] \to \text{ExpB} \to \text{ExpB} \to \text{ExpB} \to \text{ExpB}$$

$$\mathcal{C}[\![\gamma\ \Gamma]\!]\zeta_1\zeta_2\zeta_3 = \mathcal{C}[\![\gamma]\!]\zeta_1\zeta_2(\mathcal{C}[\![\Gamma]\!]\zeta_1\zeta_2\zeta_3)$$

$$\mathcal{C}[\![\text{case } \Sigma\ \beta\text{: } \varepsilon]\!]\zeta_1\zeta_2\zeta_3 = [\![\{\mathcal{S}[\![\Sigma]\!](\zeta_1)\} \to [\![\lambda\beta\ .\ \{\mathcal{E}[\![\varepsilon]\!]\}]\!](\{\zeta_2\}), \{\zeta_3\}]\!]$$

$$\mathcal{C}[\![\text{case } \Sigma\text{: } \varepsilon]\!]\zeta_1\zeta_2\zeta_3 = [\![\{\mathcal{S}[\![\Sigma]\!](\zeta_1)\} \to \{\mathcal{E}[\![\varepsilon]\!]\}, \{\zeta_3\}]\!]\ .$$

$$\mathcal{C}[\![\text{default: } \varepsilon]\!]\zeta_1\zeta_2\zeta_3 = \mathcal{E}[\![\varepsilon]\!]$$

$$\mathcal{S} \in \text{StrL} \to \text{ExpB} \to \text{ExpB}$$

$$\mathcal{S}[\![\sigma_i\ \text{case } \Sigma]\!](\zeta) = [\![[\![\sigma_i = \{\zeta\}]\!] \vee \{\mathcal{S}[\![\Sigma]\!](\zeta)\}]\!]$$

$$\mathcal{S}[\![\sigma_i]\!](\zeta) = [\![\sigma_i = \{\zeta\}]\!]$$

$$\mathcal{O}_1 \in \text{PreOp}_1 \to \text{ExpB}$$

$$\mathcal{O}_1[\![\text{seg}]\!] = [\![\text{imp "SegFn"}]\!]$$

$$\mathcal{O}_1[\![\text{chars}]\!] = [\![\text{imp "CharsFn"}]\!]$$

$$\mathcal{O}_1[\![\text{string}]\!] = [\![\text{imp "StringFn"}]\!]$$

$$\mathcal{O}_1[\![\text{mapn}]\!] = \mathit{MapN}$$

TABLE 5 (ctd.)

$$\mathcal{O}_1[\![\text{mapt}]\!] = [\![\{MapT\}(\{MapN\})]\!]$$

$$\mathcal{O}_1[\![\text{label}]\!] = [\![\lambda \iota_5.\ [\![\iota_5 \downarrow [\![\text{dim}\ \iota_5]\!]]\!]\!]$$

$$\mathcal{O}_1[\![\text{spread}]\!] = [\![\lambda \iota_5.\ [\![[\![\text{dim}\ \iota_5]\!] - \nu_1]\!]\!]$$

$$\mathcal{O}_1[\![\omega]\!] = [\![\lambda \iota_5.\ [\![\omega\ \iota_5]\!]\!]$$

$$\mathcal{O}_2 \in [\text{InOp} + \text{PreOp}_2] \to \text{ExpB}$$

$$\mathcal{O}_2[\![\neq]\!] = [\![\lambda \iota_5.\lambda \iota_6.\ [\![\sim[\![\iota_5 = \iota_6]\!]]\!]\!]$$

$$\mathcal{O}_2[\![\text{aug}]\!] = [\![\lambda \iota_5.\lambda \iota_6.\ [\![\iota_5\ \text{cat}\ \langle \iota_6 \rangle]\!]\!]$$

$$\mathcal{O}_2[\![\text{pre}]\!] = [\![\lambda \iota_5.\lambda \iota_6.\ [\![\langle \iota_5 \rangle\ \text{cat}\ \iota_6]\!]\!]$$

$$\mathcal{O}_2[\![\text{node}]\!] = [\![\lambda \iota_5.\lambda \iota_6.\ [\![\iota_6\ \text{cat}\ \langle \iota_5 \rangle]\!]\!]$$

$$\mathcal{O}_2[\![\omega]\!] = [\![\lambda \iota_5.\lambda \iota_6.\ [\![\iota_5\ \omega\ \iota_6]\!]\!]$$

$$MapN = [\![\text{fix}\ \iota_5.\ \lambda\langle \iota_6, \iota_7, \iota_8, \iota_9 \rangle.\ [\![[\![\iota_8 \leq \iota_9]\!] \to \iota_5(\langle \iota_6,\ \iota_6(\langle \iota_7, \iota_8 \rangle),\ \iota_8 + 1,\ \iota_9 \rangle),\ \iota_7]\!]\!]$$

$$MapT = [\![\lambda \iota_5.\ \lambda\langle \iota_6, \iota_7 \rangle.\ [\![\iota_5(\langle\ \lambda\langle \iota_8, \iota_9 \rangle.\ [\![\iota_8\ \text{cat}\ \langle \iota_6(\iota_7 \downarrow \iota_9) \rangle]\!],\ \langle\rangle,\ \nu_1,\ \{\mathcal{O}_1[\![\text{spread}]\!]\}(\iota_7)\ \rangle)]\!]\!]$$

Comments

⟦λπ. E⟧ is "curried" λ-abstraction, i.e. λ *x y z*. *f* is equivalent to λ*x*. λ*y*. λ*z*. *f*. It is used in a *Seg* to bind the variables denoting functions whose definition is less a matter of semantics than of implementation, and which are therefore left unspecified in a semantic description.

⟦def Δ E⟧ connects the mutually (and internally) recursive semantic functions. In a simple case E would be just result $\mathcal{F}$, where $\mathcal{F}$ identifies the main semantic function, but one could also specify initial parameters.

⟦let Δ E⟧ connects simultaneous (non-recursive) definitions.

⟦result ε⟧ comes at the end of the series of definitions in a segment. The value of the whole segment will be the value of this expression taking the definitions into account, but note that any λ-abstractions ⟦λπ. E⟧ will govern it. A segment which is to define a series of functions for use in another segment (see ⟦seg⟧ below) would give a tuple of their names as its result.

⟦δ and ... and δ⟧ is governed by def or let, q.v. (This notation allows ⟦δ_1⟧, ⟦δ_1 and δ_2⟧, etc.)

⟦$\iota_i\pi$ = ε⟧ defines a function of the "curried" parameters π, and is equivalent to ⟦ι_i = λπ. ε⟧.

⟦β = ε⟧ defines a variable or a list (tuple) of variables. ⟦⟨β_1,...,β_n⟩ = ε⟧ is equivalent to ⟦β_1 = ε↓1 and ... and β_n = ε↓*n*⟧; it does *not* produce an error if dim ε > *n* (so that β_1,...,β_n can give the branches of a node denoted by ε).

⟦def δ in ε⟧ allows a single recursive definition to govern ε ∈ Exp.

⟦let δ in ε⟧ gives a single non-recursive definition.

⟦λπ. ε⟧ denotes a "curried" abstraction in an Exp. See ⟦λπ. E⟧ above.

⟦fix β. ε⟧ denotes the minimal fixed point of ⟦λβ. ε⟧, as in LAMB.

⟦$\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3$⟧ is a conditional, as in LAMB.

⟦clauses ε § Γ §⟧ is rather similar to the **switchon** construct in BCPL [24] and other languages(*), but ε must denote a parse-tree (embedded in Pw as in §2.4). The value denoted by the whole construct is that denoted in the first (left-most) *Case* in Γ with a string matching label ε. A **default** matches any label, and the standard ⟦**default** : **err**⟧ may be omitted from after the last *Case*.

⟦ε_1 ω ε_2⟧ denotes a diadic operation.

⟦ω ε⟧ denotes a monadic operation.

⟦ω ε_1 ε_2⟧ denotes a diadic operation.

⟦$\varepsilon_1(\varepsilon_2)$⟧ denotes functional application. The unambiguous grammar in Appendix B allows parentheses to be omitted, and specifies left-association.

⟦$\varepsilon_1[\varepsilon_2/\varepsilon_3]$⟧ is the usual notation for adding a layer to an environment ε_1, associating the value of ε_2 with the identifier denoted by ε_3.

⟦$\langle \varepsilon_1, \ldots, \varepsilon_n \rangle$⟧ denotes a tuple, whose components may be obtained

(*) In particular, Extended ISWIM [33].

using ⟦↓⟧ (q.v.), and whose dimension n may be found by ⟦dim⟧ (q.v.). Note that ⟨ε⟩ is clearly distinguished from ε.

⟦⟨⟩⟧ denotes the null tuple, whose dimension is 0.

⟦ι_i⟧ are identifiers of LAMB, but $i \geq 10$ is assumed, to avoid accidental binding of variables during the transformation from MSL to LAMB.

⟦ν_i⟧, ⟦σ_i⟧, ⟦true⟧, ⟦false⟧ and ⟦err⟧ are as in LAMB.

⟦γ Γ⟧: Γ is ignored if a string in γ matches ζ_1. ζ_3 will be simply ⟦err⟧.

⟦case Σ β: ε⟧ is the MSL substitute for the meta-variables of semantic equations. When a string in Σ matches ζ_1, the variables in β are used to denote the branches of the parse-tree ζ_2 in the evaluation of ε. If the governing switch is ⟦clauses ε_0 § Γ $⟧, then ⟦case Σ β: ε⟧ ≡ ⟦case Σ: $(\lambda\beta.\varepsilon)(\varepsilon_0)$⟧.

⟦case Σ: ε⟧: here Σ gives one or more strings σ_i, and ε is evaluated if ζ_1 matches any of them.

⟦default: ε⟧ occurs (sensibly) at the end of a list of cases, and is evaluated if no other match occurs.

⟦seg⟧ enables segmentation of large semantics (and systems) into hierarchical modules. The operand is usually a string, which identifies a segment. Because segments should not have free variables, an occurrence of ⟦seg σ_i⟧ may be replaced by the MSL-expression from which the body of the segment identified by σ_i was derived (if any). ⟦seg⟧ is clearly inessential for theoretical purposes, but will prove very useful in the compiler-generator. In a practical implementation, the value

of ⟦imp "SegFn"⟧ would be provided by an interface with an independent data-base.

⟦chars⟧ assumes that its operand is a string, and produces a tuple of single-character strings. ⟦imp "CharsFn"⟧ could be specified in LAMA if LAMB strings were non-primitive and defined in terms of their constituent characters.

⟦string⟧ is the inverse of ⟦chars⟧.

⟦mapn⟧: the operand must be a 4-tuple. See *MapN* below.

⟦mapt⟧: the operand must be a 2-tuple. It is defined using *MapN*. See *MapT* below.

⟦label⟧ assumes its operand was created by ⟦node $\varepsilon_1\varepsilon_2$⟧, and gives the value of ε_1.

⟦spread⟧ also assumes its operand was created by ⟦node $\varepsilon_1\varepsilon_2$⟧, and gives the number of elements of the tuple ε_2.

⟦ω⟧: the remaining monadic operators are as in LAMB.

⟦≠⟧: note that ⟦=⟧ is only defined on integer values and ⟦err⟧ - see Table 4.

⟦aug⟧ adds an element to the end of a tuple.

⟦pre⟧ prefixes an element to a tuple.

⟦node⟧: the first operand should be a string, and the second should be a tuple (possibly null) of nodes and/or integers.

⟦ω⟧: the remaining (infix) diadic operators are as in LAMB. We defer the promised discussion of $\underline{\circ}$, $\underline{*}$ and ⫴ to the end of this section.

MapN is defined recursively to apply the function ι_6 successively to the partial result ι_7 and each integer from ι_8 to ι_9.

MapT uses *MapN* (= ι_5) to map the branches of node ι_7 with function ι_6, and form a tuple from the results.

(End of comments on Table 5.)

We have described MSL, both formally and informally; we now give an example of its use. We shall take "A Small 'Continuation' Language" from [7], and describe its mathematical semantics in MSL. The original semantic equations are reproduced here to enable the reader to compare the two notations - this example has been chosen to illustrate both the similarities and the differences.

TABLE 6: Example in Semantic Equations

(APPENDIX 2 of [8])

A Small 'Continuation' Language

Syntactic Categories

$\xi \in$ Id	Usual Identifiers
$\gamma \in$ Cmd	Commands
$\varepsilon \in$ Exp	Expressions
$\phi \in$ Fn	Some Primitive Commands

Syntax

$\gamma ::= \phi \mid$ dummy $\mid$

$\gamma_0 ; \gamma_1 \mid \varepsilon \rightarrow \gamma_0, \gamma_1 \mid$ while ε do $\gamma \mid$

goto $\varepsilon \mid$ § $\gamma_0 ; \xi_1 : \gamma_1 ; \ldots \xi_{n-1} : \gamma_{n-1}$ §$\mid$

resultis ε

$\varepsilon ::= \xi \mid$ true $\mid$ false $\mid$

$\varepsilon_0 \rightarrow \varepsilon_1, \varepsilon_2 \mid$ valof γ

Value Domains

T	Truth Values
S	Machine States (Stores)
$\theta \in C = [S \rightarrow S]$	Command Continuations
$\delta \in D = [T + C]$	Denotations
$\delta \in E = [T + C]$	Expression Results
$\kappa \in K = [D \rightarrow C]$	Expression Continuations

Semantic Functions

$\rho : [[Id \rightarrow D] \times K] = Env$	Environments
$\mathcal{P} : [Cmd \rightarrow [Env \rightarrow [C \rightarrow C]]]$	
$\mathfrak{E} : [Exp \rightarrow [Env \rightarrow [K \rightarrow C]]]$	

TABLE 6 (ctd.)

Semantic Equations

C1. $\mathcal{P}[\![\phi]\!]\rho$ = (Some given function C → C associated with ϕ)

C2. $\mathcal{P}[\![\text{dummy}]\!]\rho\theta = \theta$

C3. $\mathcal{P}[\![\gamma_0;\gamma_1]\!]\rho\theta = \mathcal{P}[\![\gamma_0]\!]\rho\{\mathcal{P}[\![\gamma_1]\!]\rho\theta\}$

C4. $\mathcal{P}[\![\varepsilon \rightarrow \gamma_0,\gamma_1]\!]\rho\theta = \&[\![\varepsilon]\!]\rho\{Cond(\mathcal{P}[\![\gamma_0]\!]\rho\theta, \mathcal{P}[\![\gamma_1]\!]\rho\theta)\}$

C5. $\mathcal{P}[\![\text{while } \varepsilon \text{ do } \gamma]\!]\rho\theta = Y(\lambda\theta'.\&[\![\varepsilon]\!]\rho\{Cond(\mathcal{P}[\![\gamma]\!]\rho\theta', \theta)\})$

C6. $\mathcal{P}[\![\text{goto } \varepsilon]\!]\rho\theta = \&[\![\varepsilon]\!]\rho\{Jump\}$

where $Jump(\delta) = \delta|C$

C7. $\mathcal{P}[\![\S\gamma_0;\xi_1:\gamma_1;\ldots\xi_{n-1}:\gamma_{n-1}\S]\!]\rho\theta = \theta_0$

where $\theta_0 = \mathcal{P}[\![\gamma_0]\!]\rho'\theta_1$

$\theta_1 = \mathcal{P}[\![\gamma_1]\!]\rho'\theta_2$

. . . . . .

$\theta_{n-1} = \mathcal{P}[\![\gamma_{n-1}]\!]\rho'\theta$

and $\rho' = \rho[\theta_1,\theta_2,\ldots,\theta_{n-1}/\xi_1,\xi_2,\ldots,\xi_{n-1}]$

C8. $\mathcal{P}[\![\text{resultis } \varepsilon]\!]\rho\theta = \&[\![\varepsilon]\!]\rho\{\rho[\![\text{res}]\!]\}$

E1. $\&[\![\xi]\!]\rho\kappa = \kappa(\rho[\![\xi]\!])$

E2. $\&[\![\text{true}]\!]\rho\kappa = \kappa(tt)$

E3. $\&[\![\text{false}]\!]\rho\kappa = \kappa(ff)$

E4. $\&[\![\varepsilon_0 \rightarrow \varepsilon_1,\varepsilon_2]\!]\rho\kappa = \&[\![\varepsilon_0]\!]\rho\{Cond(\&[\![\varepsilon_1]\!]\rho\kappa, \&[\![\varepsilon_2]\!]\rho\kappa)\}$

E5. $\&[\![\text{valof } \gamma]\!]\rho\kappa = \mathcal{P}[\![\gamma]\!](\rho[\kappa/\text{res}])\{Fail\}$

Now the MSL version, which keeps close to the "style" of the original. Whilst the syntax conforms to the unambiguous grammar in Appendix B, the lexical conventions are those of semantic equations. (Like ALGOL 60 [25] we should perhaps distinguish the "publication" language from the "reference" language.) Note that an exclamation mark (!) introduces an end-of-line comment.

TABLE 7: Example in MSL

```
! Syntax  (not quite the same as the original one)

! γ ∈ Cmd        γ ::= ϕ  | dummy  | γ ; γ  |
!                       ε → γ, γ  | while ε do γ  |
!                       goto ε  | § γ ; Δ §  | resultis ε

! ε ∈ Exp        ε ::= ξ_i  | true  | false  |
!                       ε → ε, ε  | valof γ

! Δ ∈ LabL       Δ ::= δ ; Δ  | δ

! δ ∈ Lab        δ ::= ξ_i : γ

! ξ_i ∈ Ide      ξ_0, ξ_1, ... primitive identifiers

! ϕ_i ∈ Fun      ϕ_0, ϕ_1, ... primitive commands

! Domains

! β ∈ T   = truth values denoted by true, false
! σ ∈ S   = stores
! θ ∈ C   = [S → S]
!     D   = [T + C]
! κ ∈ K   = [D → C]
! ρ ∈ Env = [N → [D+K]]
!     N   = integers denoted by 0, 1, ...
```

TABLE 7 (ctd.)

! Semantic Functions

```
λ⟨ FunVal,                    ! FunVal ∈ [N → C → C]
  Jump,                       ! Jump   ∈ K
  Fail,                       ! Fail   ∈ C
  Finish⟩ .                   ! Finish ∈ C

let res = 0 and start = 1
    ! res and start will be used as "private" variables.

def 𝒫⟦t⟧ρθ = .                ! 𝒫 ∈ [Cmd → Env → C → C]
clauses t

§ case "φ " ⟨i⟩ :             FunVal⟦i⟧θ

  case "dummy" ⟨⟩ :           θ

  case "γ ; γ" ⟨c0,c1⟩ :      𝒫⟦c0⟧ρ{𝒫⟦c1⟧ρθ}

  case "ε → γ, γ" ⟨e,c0,c1⟩ : ℰ⟦e⟧ρ{Cond(𝒫⟦c0⟧ρθ,𝒫⟦c1⟧ρθ)}

  case "while ε do γ" ⟨e,c⟩ : fix θ'. ℰ⟦e⟧ρ{Cond(𝒫⟦c⟧ρθ',θ)}

  case "goto ε" ⟨e⟩ :         ℰ⟦e⟧ρ{Jump}

  case "§ γ ; Δ §" ⟨c,dl⟩ : (fix ζ. let ρ'=Lay⟨ρ,ζ, start pre ℒ⟦dl⟧⟩
                                 in 𝒫⟦c⟧ρ'(ζ↓2) pre 𝒟⟦dl⟧ρ'θ) ↓ 1

  case "resultis ε" ⟨e⟩ :     ℰ⟦e⟧ρ{ρ⟦res⟧}
§
```

TABLE 7 (ctd.)

```
and 𝓔⟦t⟧ρκ =                                  ! 𝓔 ∈ [Exp → Env → K → C]
clauses t
§ case "ξ " ⟨i⟩ :                             ρ⟦IdeVal⟦i⟧⟧
  case "true":                                κ(true)
  case "false":                               κ(false)
  case "ε → ε, ε" ⟨e0,e1,e2⟩ :                𝓔⟦e0⟧ρ{Cond(𝓔⟦e1⟧ρκ, 𝓔⟦e2⟧ρκ)}
  case "valof γ" ⟨c⟩ :                        𝓟⟦c⟧(ρ[κ/res]){Fail}
$

and 𝓓⟦t⟧ρθ =                                  ! 𝓓 ∈ [LabL → Env → C → C*]
clauses t
§ case "δ ; Δ" ⟨d,dl⟩ :                       let ζ = 𝓓⟦dl⟧ρθ in
                                              𝓓1⟦d⟧ρ(ζ↓1) pre ζ
  case "δ" ⟨d⟩ ;                              ⟨𝓓1⟦d⟧ρθ⟩
$

and 𝓓1⟦t⟧ρθ =                                 ! 𝓓1 ∈ [Lab → Env → C → C]
clauses t
§ case "ξi: γ" ⟨i,c⟩ :                        𝓟⟦c⟧ρθ
$
```

TABLE 7 (ctd.)

```
and ℒ⟦t⟧ =                                  ! ℒ ∈ [LabL → N*]
clauses t

§ case "δ ; Δ" ⟨d,dl⟩ :                     ℒ₁⟦d⟧ pre ℒ⟦dl⟧

  case "δ" ⟨d⟩ :                            ⟨ℒ₁⟦d⟧⟩
$
```

```
and ℒ₁⟦t⟧ =                                 ! ℒ₁ ∈ [Lab → N]
clauses t

§ case "ξᵢ: γ" ⟨i,c⟩ :                      IdeVal⟦i⟧
$
```

```
and Cond(ζ₀,ζ₁)(β) =                        ! Cond ∈ [[Any x Any] → T → Any]
    β → ζ₀, ζ₁

and IdeVal⟦i⟧ =                             ! IdeVal ∈ [N→ N]
    i + 2                ! to avoid confusion with res and start.

and Lay⟨ρ,ζ,η⟩ =                            ! Lay ∈ [[Env x C* x N*] → Env]
    mapn ⟨ λ⟨ρ',ν⟩ . ρ'[ζ↓ν/η↓ν], ρ, 1, dim η ⟩

result    λt. 𝔓⟦t⟧(λi.err){Finish}

! End of MSL version.
```

To conclude our presentation of MSL, we motivate the inclusion of the operators $\underline{\circ}$, $\underline{*}$ and $\|$. These operators belong also to LAMB, and were defined in §2.3 (Table 4). The $\underline{\circ}$ and $\underline{*}$ are related to the usual $\circ$ and $*$ of semantic equations, in that $f \mathbin{\underline{\circ}} g$ corresponds to $g \circ f$, and $f \mathbin{\underline{*}} g$ corresponds to $g * f$. We shall argue that $\underline{\circ}$ and $\underline{*}$ are much more convenient.

The operators $\circ$ and $*$ have been used to abbreviate λ-notation in semantic equations from the start ("*" was written as ":" until 1970). They significantly reduce the number of parentheses and λ-abstractions needed to denote composition of functions, especially when the functionalities are based on $S \to S$ and $S \to [E \times S]$. As an example we take an equation from an "Assignment Language" due to Strachey [26]:

$$(2) \qquad \mathcal{C}[\![\text{let } \xi = \varepsilon \text{ in } \gamma]\!]\rho = (\lambda\alpha.\ \mathit{Lose}(\alpha) \circ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]) * \mathit{StoreAway} * \mathcal{R}[\![\varepsilon]\!]\rho$$

Here we have

$$\mathcal{C} \in [\mathbf{Cmd} \to \mathbf{Env} \to \mathbf{S} \to \mathbf{S}]$$

$$\mathcal{R} \in [\mathbf{Exp} \to \mathbf{Env} \to \mathbf{S} \to [\mathbf{V} \times \mathbf{S}]]$$

$$\mathit{Lose} \in [\mathbf{L} \to \mathbf{S} \to \mathbf{S}]$$

$$\mathit{StoreAway} \in [\mathbf{V} \to \mathbf{S} \to [\mathbf{L} \times \mathbf{S}]]$$

In unabbreviated λ-notation the right-hand side of (2) is

$$(3) \qquad \lambda\sigma.\ \{\lambda\langle\beta,\sigma_1\rangle.\ (\lambda\langle\alpha,\sigma_2\rangle.\ \mathit{Lose}(\alpha)(\mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]/\sigma_2)))(\mathit{StoreAway}(\beta)(\sigma_1))\}(\mathcal{R}[\![\varepsilon]\!]\rho\sigma),$$

where $\rho \in \mathbf{Env}$, $\sigma \in \mathbf{S}$, $\alpha \in \mathbf{L}$ and $\beta \in \mathbf{V}$.

Both (2) and (3) "read" from right to left, i.e. in the order in which the component functions are applied to the store σ: $\mathcal{R}[\![\varepsilon]\!]\rho\sigma$ gives $\langle\beta,\sigma_1\rangle$, then $\mathit{StoreAway}(\beta)(\sigma_1)$ gives $\langle\alpha,\sigma_2\rangle$, then $\mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]\sigma_2$ gives σ_3, and $\mathit{Lose}(\alpha)(\sigma_3)$ gives the final σ_4.

In 1971, Wadsworth introduced a technique called "continuations" for describing jumps (**goto** statements) in programs [7] - in fact our example semantics in MSL used this technique. Basically, the semantic value of a command or expression is made to be dependent on a continuation parameter, which in effect is the value of the *rest* of the program.

Using continuations for both commands and expressions, (2) becomes

(4) $$\mathcal{C}[\![\texttt{let } \xi = \varepsilon \texttt{ in } \gamma]\!]\rho\theta =$$
$$\mathcal{R}[\![\varepsilon]\!]\rho\ \lambda\beta.\ \mathit{StoreAway}(\beta)\{\lambda\alpha.\ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]\{\mathit{Lose}(\alpha)\{\theta\}\}\}$$
$$= \mathcal{R}[\![\varepsilon]\!]\rho\ \|\ \lambda\beta.\ \mathit{StoreAway}(\beta)\|\ \lambda\alpha.\ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]\|\ \mathit{Lose}(\alpha)\|\ \theta$$

where we have used the MSL operator $\|$ in the last formula, to remove some of the parentheses (braces). The types are now

$$\mathcal{C} \in [\mathbf{Cmd} \to \mathbf{Env} \to \mathbf{C} \to \mathbf{C}]$$
$$\mathcal{R} \in [\mathbf{Exp} \to \mathbf{Env} \to [\mathbf{V} \to \mathbf{C}] \to \mathbf{C}]$$
$$\mathit{Lose} \in [\mathbf{L} \to \mathbf{C} \to \mathbf{S}]$$
$$\mathit{StoreAway} \in [\mathbf{V} \to [\mathbf{L} \to \mathbf{C}] \to \mathbf{C}]$$

and $\theta \in \mathbf{C} = [\mathbf{S} \to \mathbf{S}]$.

Clearly (4) "reads" from left to right.

The difficulty comes when the language described is such that it is possible to jump out of commands, but not out of expressions or the primitive functions *Lose* and *StoreAway*. Then (4) becomes (ignoring questions of non-termination and error):

(5) $$\mathcal{C}[\![\texttt{let } \xi = \varepsilon \texttt{ in } \gamma]\!]\rho\theta =$$
$$(\lambda\alpha.\ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi]\{\theta \circ \mathit{Lose}(\alpha)\}) * \mathit{StoreAway} * \mathcal{R}[\![\varepsilon]\!]\rho$$

where $\mathcal{C}$ is as for (4), but $\mathcal{R}$, *Lose* and *StoreAway* are as for (2). Equation (5) reads "from the outside in", and is not as clear as (2) or (4).

However if in (5) $\circ$ and * are replaced by $\underline{\circ}$ and $\underline{*}$, the right-hand side becomes

(6) $\mathcal{R}[\![\varepsilon]\!]\rho \mathbin{\underline{*}} \mathit{StoreAway} \mathbin{\underline{*}} \lambda\alpha.\ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi] \parallel \mathit{Lose}(\alpha) \mathbin{\underline{\circ}} \theta$

whereas (2) becomes

(7) $\mathcal{R}[\![\varepsilon]\!]\rho \mathbin{\underline{*}} \mathit{StoreAway} \mathbin{\underline{*}} \lambda\alpha.\ \mathcal{C}[\![\gamma]\!]\rho[\alpha/\xi] \mathbin{\underline{\circ}} \mathit{Lose}(\alpha).$

Both (6) and (7) read from left to right in the same way as (4). We hope that the advantages of this uniformity are self-evident, and that they compensate for such a turn-about in the basic notation of semantic clauses.

That concludes our presentation of MSL. We do not claim that MSL is the "ultimate" in semantic description languages: new techniques in mathematical semantics will probably demand new notation, and MSL doesn't even try to include all the currently used notation. MSL is intended only as a basic notation which can be used to describe (formally) extensions to more sophisticated notations.

CHAPTER 3

LAMB as a Machine Code

In this Chapter we continue our study of LAMB. §3.1 discusses our principal use for LAMB: as the code for use in a compiler-generator. §3.2 describes how we can obtain direct specifications of simple values which are denoted by complex LAMB-terms - by manipulating finite sets of integers or by applying denotation-preserving conversion rules. Reduction rules and a normal form for LAMB-terms are defined in §3.3. Then in §3.4 we give an algorithm for reducing LAMB-terms to normal form. It uses simulated substitution for β-reduction, and it is based on "call-by-need" (rather than on "call-by-name" or "call-by-value").

We do not give a proof of the correctness of the LAMB-interpreter algorithm presented here, although one could presumably be obtained by use of techniques such as those given by Milne [4]. - it would be lengthy. Instead we hope that the reader can be persuaded to give credence to the correctness of the algorithm by the fact that a straight-forward coding of the algorithm (in BCPL [24]) does produce the expected results when used on large (and also on pathological) LAMB-terms. Examples of the use of the algorithm in the compiler-generator system are given in Chapter 5.

§3.1 Combining LAMB-terms

The semantics of LAMB in §2.3 gives every LAMB-term a denotation in Pw. Suppose that a LAMB-term ϕ denotes (the graph of) a continuous function f on Pw; and that a second LAMB-term ζ denotes an arbitrary value z. Then clearly the LAMB-term $\phi(\zeta)$ denotes the value $f(z)$.

Now consider a LAMB-term Φ which denotes some semantic function $\mathcal{F}$ - a function from parse-trees (embedded in Pw as in §2.4) of a programming language, to their semantic values (also embedded in Pw). If τ is a LAMB-term denoting a particular parse-tree t, then $\Phi(\tau)$ denotes its semantic value $\mathcal{F}[\![t]\!]$. In other words we can regard $\Phi(\tau)$ as a coding of $\mathcal{F}[\![t]\!]$, i.e. as the code of the program which t represents. Obviously LAMB is not a run-of-the-mill machine code, but we shall show that it can be implemented with reasonable efficiency (§3.3). So the simple operation of combining Φ with τ to form the application $\Phi(\tau)$ can be considered as a part of the operation of compiling - that of code generation.

Part of the task of our compiler-generator is to produce the code of a code-generator from (the parse-tree of) a description of a mathematical semantics in MSL. Suppose $\tau_{\mathcal{F}}$ is a LAMB-term denoting the parse-tree of a description of $\mathcal{F}$ in MSL. The LAMB-term in Appendix C denotes the semantic function for MSL - let us call the LAMB-term Λ, and the semantic function $\mathcal{L}$. So $\mathcal{L}$ is a function from parse-trees of descriptions according to the grammar of MSL, to their values as semantic functions. Hence the LAMB-term $\Lambda(\tau_{\mathcal{F}})$ denotes $\mathcal{F}$, and can be used instead of Φ above; therefore combining $\Lambda(\tau_{\mathcal{F}})$ with τ in $\Lambda(\tau_{\mathcal{F}})(\tau)$ generates the code of the program whose parse-tree is denoted by τ. Because of this we consider $\Lambda(\tau_{\mathcal{F}})$ as the code of a code-generator, and Λ as the code of part of a compiler-generator.

In Chapter 4 we shall show how LAMB can be used as the code for other parts of compilers and of our compiler-generator.

We can also formulate a test for the "consistency" of the semantics of MSL, by combining LAMB-terms. Appendix D gives a circular description of the semantics of MSL in MSL. Suppose that $\mathcal{L}$ associates a semantic function $\mathcal{L}'$ with this description. For consistency in what we think the semantics of MSL is, $\mathcal{L}'$ must be equivalent to $\mathcal{L}$. Let $\tau_{\mathcal{L}'}$ be a LAMB-term denoting the parse-tree of the circular description; then $\Lambda(\tau_{\mathcal{L}'})$ denotes $\mathcal{L}'$. Hence $\Lambda(\tau_{\mathcal{L}'})$ and Λ must be equivalent in some sense. For the particular Λ given in Appendix C, we can in fact show that something stronger holds, namely that $\Lambda(\tau_{\mathcal{L}'})$ *reduces* to Λ by the application of the value-preserving conversion rules which we shall define in the next section.

§3.2 Implementing LAMB

In the last section we indicated how to "generate" the code of programs and compilers, simply by combining LAMB-terms denoting functions and parse-trees. The triviality of this operation guarantees its correctness, but correctness isn't everything. Whilst there is no mathematical difference between a complex expression and a simpler one if they both denote the same value, from a practical viewpoint the latter is much preferable because we have an increased chance of being able to intuit the denoted value directly. To take an extreme example, we recognize a difference between a numeral denoting the 1,000th prime number, and a complicated function, for calculating the nth prime, applied to the numeral 1000 - roughly speaking we are not interested in which of the many possible algorithms is used to calculate the result, but only in the result itself.

To transform a LAMB-term into "something simpler", we see two ways of proceeding: (i) LAMBDA can be implemented directly, by computing certain operations on elements of Pw, and then, using the semantic description of LAMB to translate LAMB into LAMBDA, an individual element of Pw might be obtained; (ii) a LAMB-term can be reduced, using conversion rules which preserve value, to other LAMB-terms, eventually yielding simple irreducible LAMB-terms when such exist.

We shall see that for full generality both these methods are needed; but we may consider them independently.

(i) Implementation of LAMBDA

An obvious pre-condition for obtaining a useful result by implementation (in finite time) is that the result wanted must be a finite element of Pw, such as a singleton representing an integer

or a finite set representing part of a function. Thus in general we cannot obtain the *code* of a program (considered as a function from input to output) unless we restrict the program to a finite domain of input - which for practical purposes means not only finite but also very small. However as the *output* of a program is usually finite, implementation of LAMBDA would provide a general method for evaluating programs mathematically.

The basis of a simple-minded implementation of LAMBDA is as follows. The semantics of LAMBDA is made into a function of an integer t(time), whose use is to limit the number of points in the graph of each λ-abstraction, viz.

(1) $\lambda x.\ \tau = .\{(n,m) \mid m \in \tau[e_n/x],\ n < t\}.$

For finite t it follows that, with this semantics, any LAMBDA-term ε denotes a finite element $e_{n(t)}$ of Pw; that $e_{n(t)} \subseteq e_{n(t+1)}$ (hence $n(t) \leq n(t+1)$); and that the limit of the sequence $e_{n(0)}, e_{n(1)}, \ldots$ is the element (say e_∞) denoted by ε through the original semantics of §1.4. As the predicates $k \in e_n$, $e_n \subseteq e_m$ and $e_n = e_m \cup e_k$ are all (primitive) recursive in n, m and k, the set operations of the semantic description can be replaced by simple arithmetical functions on the indices of the e_n, so that the semantics gives $n(t)$ directly, instead of $e_{n(t)}$, as the denotation of ε at time t.

For each expression ε, we assume that there is a (recursive) predicate *IsFinished*, such that *IsFinished*(n) is true iff e_n is a possible desired result, which satisfies

(2) *IsFinished*(n) and $e_n \subseteq e_m$ implies *IsFinished*(m).

(This reflects the idea of "positive information" in the Pw model.) Clearly, if *IsFinished*$(n(t))$ is true for some finite t, the required result has been found in "finite time"; conversely, if *IsFinished*$(n(t))$ is false for all t, then either $\{n(t) \mid t \in \omega\}$ is

finite, giving $e_\infty = e_{n(t')}$ for some t', or it is infinite - both cases imply *IsFinished*(m) must be false for all m such that $e_m \subseteq e_\infty$; hence our implementation does not "miss" any desired finite results.

(ii) Reduction of LAMB-terms

It will not surprise the reader that LAMB-terms can be reduced by application of a conversion rule analogous to β-conversion in the λ-calculus. This can achieve considerable simplification of LAMB-terms, whilst preserving the values they denote.

However there is more to LAMB than just λ: in particular there is the conditional expression $\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3$ which, if ε_1 reduces to **true** or **false**, can be reduced further to ε_2 or ε_3. However, consider $(\lambda\iota.\ \varepsilon_1) \rightarrow \varepsilon_2, \varepsilon_3$. Can this be reduced, preserving the denotation? The answer is that "it depends": it depends on what is assumed about the coding function (n,m) and the enumeration of finite sets e_n, which were used in giving the semantics of LAMBDA in §1.4. If nothing at all is assumed, then the conditional cannot be eliminated, because we cannot determine whether or not 0 is an element of the denotation of $\lambda\iota.\ \varepsilon_1$. At the other extreme, if (n,m) and e_n are as defined in §1.3, then it can be shown that the denotations of such expressions as $\lambda\iota.\ \nu_{i+1}$ do not contain *0* (but are non-empty) so $(\lambda\iota.\ \nu_{i+1}) \rightarrow \varepsilon_2, \varepsilon_3$ could be reduced to ε_3.

In fact we shall regard such LAMB-terms as the above conditionals as "puns" - their denotations are dependent on the "implementation" of LAMBDA. We allow puns as legal terms so that Scott's Definability Theorem (§1.4) can be used to show that LAMB is general enough; but in using LAMB in conjunction with MSL to describe the semantics of programming languages, it seems that we do not need to use puns at all. Hence for our purposes it will be sufficient to consider only the reduction of LAMB-terms not containing puns. In

any case it seems highly unlikely that one could give a sufficiently complete set of reduction rules for puns.

Therefore our reduction rules will not be completely general and will not always suffice to reduce, for example, a LAMB-term denoting an integer to the corresponding numeral; but this will not matter to us. We shall specify our rules in the next section.

§3.3 Reductions in LAMB

To specify a set of reduction rules for LAMB, we shall use the notation which we introduced in §2.3(to describe the syntactic mapping from LAMB to LAMA), extended to allow compound phrases to be nested on the left of "semantic" equations as well as on the right. Thus the left-hand side of an equation may specify the form of a phrase to an arbitrary degree of complexity, and, as usual, meta-variables specify the syntactic category to which a sub-phrase must belong, simultaneously naming the sub-phrase. We allow the omission of the enclosing brackets ⟦ ⟧ from inner compound phrases, when their positions can be inferred unambiguously by reference to the grammar for LAMB (Table 4).

We shall avoid indeterminacy by ensuring that if two equations can apply to the same phrase, then one is more "specific" than the other and has priority, i.e. the left-hand side of the first results from that of the second from replacement of a meta-variable by a compound phrase. For example

(1) $\mathcal{C}⟦⟦\lambda\iota.\ \varepsilon_1⟧(\varepsilon_2)⟧ = \ldots$

would be more specific than

(2) $\mathcal{C}⟦\varepsilon_1(\varepsilon_2)⟧ = \ldots$.

We postulate that the "identity reduction" applies to all phrases to which no other reduction applies, so that we may regard a set of reduction equations as specifying a syntactic transformation on the whole language.

Table 8 gives our reduction rules for LAMB. To show their correctness we would need to prove

(3) $(\mathcal{E} \circ \mathcal{F} \circ \mathcal{C})⟦\varepsilon⟧\rho = (\mathcal{E} \circ \mathcal{F})⟦\varepsilon⟧\rho$

for all LAMB-terms ε and environments ρ - recall that $\mathcal{E} \circ \mathcal{F}$ is the semantic function for LAMB. We shall not attempt a proof here, but

the method is clear: first adapt the proof by Wadsworth in [22] of the "Substitution Lemma" (for the λ-calculus) to LAMA - our domain of interpretation is very similar to his. Then show that various simple conversion rules are valid in LAMA. Finally use structural induction to show that

(4) $(\mathcal{F}\circ\mathcal{C})[\![\varepsilon]\!]$ converts in LAMA to $\mathcal{F}[\![\varepsilon]\!]$.

which implies (3). It is clear that the proof would be rather tedious, and out of proportion to the simplicity of the rules themselves. We hope that a careful comparison of Table 8 with Table 4 will serve to convince the reader that the reduction rules expressed in the former do indeed preserve the denotations of LAMB-terms as assigned by the latter.

It should be noted that the reduction rules are in fact a little "conservative": stronger ones, such as

(5) $\mathcal{C}[\![\nu_i = \sigma_j]\!] = i=j \rightarrow [\![\mathbf{true}]\!], [\![\mathbf{false}]\!]$

also preserve denotations. This reflects the fact that certain expressions, such as $[\![\nu_0]\!]$, $[\![\sigma_0]\!]$ and $[\![\mathbf{true}]\!]$, were distinguished in LAMB purely for pragmatic (mnemonic) purposes, and correspond to identical expressions in LAMA - this applies also to tuples and certain functions. By omitting the stronger rules from Table 8 we are saying that we shall keep, e.g., numerals $[\![\nu_i]\!]$, strings $[\![\sigma_i]\!]$ and truth-values $[\![\mathbf{true}]\!]$, $[\![\mathbf{false}]\!]$ separate in LAMB, and not expect the LAMB-operator "=" to identify them.

A normal form of a LAMB-term ε is an expression ε′ such that $IsNormal[\![\varepsilon']\!] = true$, and which is obtained from ε by a sequence of operations of the form:
replace a well-formed sub-expression ε″ by $\mathcal{C}[\![\varepsilon'']\!]$.

We conjecture that the equivalent in LAMB-reduction of the First Church-Rosser Theorem [27] holds, i.e. that if a LAMB-term ε has a normal form, then this normal form is unique (up to α-conversion). LAMB-reduction is, in effect, a combination of an extended β-reduction rule and a few rules for so-called δ-conversion.

In the next section we proceed to consider a reduction algorithm to find a normal form of a LAMB-term whenever one exists.

TABLE 8: Reduction in LAMB

Syntax As in Table 4.

Domains

$\alpha \in$ ExpB (as defined in Table 4.)

Mapping $\mathcal{C} \in$ [ExpB → ExpB]

$\mathcal{C}[\![\text{true} \rightarrow \varepsilon_2, \varepsilon_3]\!] = [\![\varepsilon_2]\!]$

$\mathcal{C}[\![\text{false} \rightarrow \varepsilon_2, \varepsilon_3]\!] = [\![\varepsilon_3]\!]$

$\mathcal{C}[\![\nu_i + \nu_j]\!] = [\![\nu_{i+j}]\!]$

$\mathcal{C}[\![\nu_i - \nu_j]\!] = i \leq i \rightarrow [\![\nu_{i-j}]\!], [\![\nu_i - \nu_j]\!]$

$\mathcal{C}[\![\nu_i \times \nu_j]\!] = [\![\nu_{i \times j}]\!]$

$\mathcal{C}[\![\nu_i = \nu_j]\!] = i=j \rightarrow [\![\text{true}]\!], [\![\text{false}]\!]$

$\mathcal{C}[\![\sigma_i = \sigma_j]\!] = i=j \rightarrow [\![\text{true}]\!], [\![\text{false}]\!]$

$\mathcal{C}[\![\text{err} = \text{err}]\!] = [\![\text{true}]\!]$

$\mathcal{C}[\![\text{err} = \nu_i]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\nu_i = \text{err}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\text{err} = \sigma_i]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\sigma_i = \text{err}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\text{err} = \text{true}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\text{err} = \text{false}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\text{true} = \text{err}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\text{false} = \text{err}]\!] = [\![\text{false}]\!]$

$\mathcal{C}[\![\nu_i \leq \nu_j]\!] = i \leq j \rightarrow [\![\text{true}]\!], [\![\text{false}]\!]$

TABLE 8 (ctd.)

$$\mathcal{C}[\![\text{true} \vee \text{true}]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{true} \vee \text{false}]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{false} \vee \text{true}]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{false} \vee \text{false}]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\text{true} \wedge \text{true}]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{true} \wedge \text{false}]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\text{false} \wedge \text{true}]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\text{false} \wedge \text{false}]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\langle \varepsilon_1,\ldots,\varepsilon_m\rangle \text{ cat } \langle \varepsilon_{m+1},\ldots,\varepsilon_n\rangle]\!] = [\![\langle \varepsilon_1,\ldots,\varepsilon_m,\varepsilon_{m+1},\ldots,\varepsilon_n\rangle]\!]$$

$$\mathcal{C}[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle \downarrow \nu_i]\!] = [\![\varepsilon_i]\!]$$

$$\mathcal{C}[\![\sim \text{true}]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\sim \text{false}]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{dim } \langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!] = [\![\nu_n]\!]$$

$$\mathcal{C}[\![\text{dim } \langle\rangle]\!] = [\![\nu_0]\!]$$

$$\mathcal{C}[\![\text{isatom } \nu_i]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{isatom } \sigma_i]\!] = [\![\text{true}]\!]$$

$$\mathcal{C}[\![\text{isatom } \langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![\text{isatom } \langle\rangle]\!] = [\![\text{false}]\!]$$

$$\mathcal{C}[\![[\![\lambda\beta.\ \varepsilon_1]\!](\varepsilon_2)]\!] = \mathcal{Y}[\![\beta]\!][\![\varepsilon_2]\!][\![\varepsilon_1]\!]$$

$$\mathcal{C}[\![[\![\text{val } \beta.\ \varepsilon_1]\!](\varepsilon_2)]\!] = \mathit{IsNormal}[\![\varepsilon_2]\!] \rightarrow \mathcal{Y}[\![\beta]\!][\![\varepsilon_2]\!][\![\varepsilon_1]\!],\ [\![[\![\text{val } \beta.\ \varepsilon_1]\!](\varepsilon_2)]\!]$$

$$\mathcal{C}[\![[\![\text{fix } \beta.\ \varepsilon_1]\!](\varepsilon_2)]\!] = [\![\{\mathcal{Y}[\![\beta]\!][\![\text{fix } \beta.\ \varepsilon_1]\!][\![\varepsilon_1]\!]\}(\varepsilon_2)]\!]$$

$$\mathcal{C}[\![[\![\varepsilon_0 \circ \varepsilon_1]\!](\varepsilon_2)]\!] = [\![\{\mathit{Strict}[\![\varepsilon_1]\!]\}(\{\mathit{Strict}[\![\varepsilon_0]\!]\}(\varepsilon_2))]\!]$$

$$\mathcal{C}[\![[\![\varepsilon_0 * \varepsilon_1]\!](\varepsilon_2)]\!] = [\![\{\mathit{Curry}\ [\![\varepsilon_1]\!]\}(\{\mathit{Strict}[\![\varepsilon_0]\!]\}(\varepsilon_2))]\!]$$

$$\mathcal{C}[\![[\![\varepsilon_0 \parallel \varepsilon_1]\!](\varepsilon_2)]\!] = [\![\{\mathit{Strict}[\![\varepsilon_0(\varepsilon_1)]\!]\}(\varepsilon_2)]\!]$$

$$\mathcal{C}[\![[\![\text{pair } \varepsilon_1]\!](\varepsilon_2)]\!] = [\![\{\mathit{Pair}[\![\varepsilon_1]\!]\}(\varepsilon_2)]\!]$$

$$\mathcal{C}[\![[\![\text{imp } \sigma_i]\!](\varepsilon_2)]\!] = \mathit{IsNormal}[\![\varepsilon_2]\!] \rightarrow \mathit{ImpFn}(i)[\![\varepsilon]\!],\ [\![[\![\text{imp } \sigma_i]\!](\varepsilon_2)]\!]$$

TABLE 8 (ctd.)

$$\mathcal{J} \in [\text{Bvs} \to \text{ExpB} \to \text{ExpB} \to \text{ExpB}]$$

$$\begin{aligned}
\mathcal{J}[\![\iota_i]\!]\alpha_2\alpha_1 &= \mathcal{J}_0[\![\{\alpha_1\}]\!]\alpha_2 i \\
\mathcal{J}[\![\langle \beta_1,\ldots,\beta_n\rangle]\!]\alpha_2\alpha_1 &= \mathcal{J}[\![\beta_1]\!][\![\{\alpha_2\}\downarrow\nu_1]\!](\\
&\qquad \mathcal{J}[\![\beta_2]\!][\![\{\alpha_2\}\downarrow\nu_2]\!](\\
&\qquad\quad \ldots\ \mathcal{J}[\![\beta_n]\!][\![\{\alpha_2\}\downarrow\nu_n]\!](\alpha_1)\ \ldots\ )) \\
\mathcal{J}[\![\langle\rangle]\!]\alpha_2\alpha_1 &= \alpha_1
\end{aligned}$$

$$\mathcal{J}_0 \in [\text{ExpB} \to \text{ExpB} \to \text{Int} \to \text{ExpB}]$$

$$\begin{aligned}
\mathcal{J}_0[\![\lambda\beta.\ \varepsilon]\!]\alpha_2 i &= \text{let } \eta = \mathit{UnusedBvs}[\![\lambda\beta.\ \varepsilon]\!][\![\{\alpha_2\}]\!](i) \text{ in} \\
&\qquad [\![\lambda\{\eta\}.\ \{\mathcal{J}_0[\![\{\mathcal{J}[\![\{\eta\}]\!][\![\beta]\!][\![\varepsilon]\!]\}]\!]\alpha_2 i\}]\!] \\
\mathcal{J}_0[\![\text{val } \beta.\ \varepsilon]\!]\alpha_2 i &= \text{similar to } \lambda\beta.\ \varepsilon \\
\mathcal{J}_0[\![\text{fix } \beta.\ \varepsilon]\!]\alpha_2 i &= \text{similar to } \lambda\beta.\ \varepsilon \\
\mathcal{J}_0[\![\varepsilon_1 \to \varepsilon_1,\ \varepsilon_2]\!]\alpha_2 i &= [\![\{\mathcal{J}_0[\![\varepsilon_1]\!]\alpha_2 i\} \to \{\mathcal{J}_0[\![\varepsilon_2]\!]\alpha_2 i\},\ \{\mathcal{J}_0[\![\varepsilon_3]\!]\alpha_2 i\}]\!] \\
\mathcal{J}_0[\![\varepsilon_1\ \omega\ \varepsilon_2]\!]\alpha_2 i &= [\![\{\mathcal{J}_0[\![\varepsilon_1]\!]\alpha_2 i\}\ \omega\ \{\mathcal{J}_0[\![\varepsilon_2]\!]\alpha_2 i\}]\!] \\
\mathcal{J}_0[\![\omega\ \varepsilon]\!]\alpha_2 i &= [\![\omega\ \{\mathcal{J}_0[\![\varepsilon]\!]\alpha_2 i\}]\!] \\
\mathcal{J}_0[\![\varepsilon_1(\varepsilon_2)]\!]\alpha_2 i &= [\![\{\mathcal{J}_0[\![\varepsilon_1]\!]\alpha_2 i\}(\{\mathcal{J}_0[\![\varepsilon_2]\!]\alpha_2 i\})]\!] \\
\mathcal{J}_0[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!]\alpha_2 i &= [\![\langle \{\mathcal{J}_0[\![\varepsilon_1]\!]\alpha_2 i\},\ldots,\{\mathcal{J}_0[\![\varepsilon_n]\!]\alpha_2 i\}\rangle]\!] \\
\mathcal{J}_0[\![\iota_j]\!]\alpha_2 i &= i=j \to \alpha_2,\ [\![\iota_j]\!]
\end{aligned}$$

$$\mathit{Strict}[\![\varepsilon]\!] = [\![\text{val } \iota_i.\ [\![\varepsilon(\iota_i)]\!]]\!] \quad \text{where } \iota_i \text{ does not occur free in } \varepsilon.$$

$$\mathit{Curry}[\![\varepsilon]\!] = [\![\lambda\iota_i.\ [\![[\![\text{val } \iota_j.\text{val } \iota_k.\ [\![\varepsilon(\iota_j)(\iota_k)]\!]]\!](\iota_i\downarrow\nu_1)(\iota_i\downarrow\nu_2)]\!]]\!] \quad \text{where } \iota_i,\iota_j,\iota_k \text{ do not occur free in } \varepsilon.$$

$$\mathit{Pair}[\![\varepsilon]\!] = [\![\text{val } \iota_i.\ \langle \varepsilon,\iota_i\rangle]\!] \quad \text{where } \iota_i \text{ does not occur free in } \varepsilon.$$

TABLE 8 (ctd.)

IsNormal ϵ [ExpB → Bool]

IsNormal⟦ε⟧ = *false* when ε is of such a form that one of the above equations is applicable to ε, or when this holds for some sub-phrase of ε.

IsNormal⟦ε⟧ = *true* otherwise.

(A definition of *IsNormal* could easily be given - it would have the same structure as that of $\mathcal{C}$ above.)

UnusedBvs ϵ [ExpB → ExpB → Int → Bvs]

UnusedBvs⟦λβ. ε_1⟧⟦ε_2⟧(*i*) gives a list β′ of *Bvs* of the same shape as β, such that no identifier ι_j in β′ has $j=i$ or occurs in $\beta, \varepsilon_1, \varepsilon_2$.

3.4 An Interpreter for LAMB

LAMB is similar to a λK-calculus [27], and allows the cancellation of sub-terms during reduction. Therefore we cannot expect the Second Church-Rosser Theorem [27] to hold for LAMB-reduction, and so a haphazard sequence of reductions of a LAMB-term ε (having a normal form) will not in general always yield a normal form. Consider any algorithm which systematically controls the order in which reductions are applied to well-formed sub-expressions of a term ε. For such an algorithm to succeed in finding a normal form *whenever* one exists, it is clear that it must have a property similar to that of the so-called normal-order reduction algorithm for the λ-calculus: it never tries to reduce to normal form any subexpression which could be cancelled by an another reduction. For example, in an applicative combination $(\lambda x.\ M)N$, the term N may be reduced immediately to normal form only if it is known that x would not be cancelled in the reduction of M. As it is difficult to decide this in advance, interpreters which guarantee to find normal forms (when possible) usually adopt a "wait and see" approach: either N is sustituted for all free occurences of x in M, or else we "remember" that x now denotes N until x is actually needed in the reduction of M - this technique is known as simulated substitution.

We shall give a suitable algorithm for LAMB-reduction later in this section. Basically it implements the essential principle of normal-order reduction by using simulated substitution for β-reductions, and by operating as follows:

(i) In a conditional expression $B \rightarrow M, N$ the term B is reduced to normal form first of all; if this yields **true** or **false** then either N or M is cancelled, respectively.

(ii) In the selection of a component of a tuple $T \downarrow N$, the term T is reduced only so far as is necessary to yield a tuple $\langle T_1, T_2, \ldots, T_n \rangle$; N is reduced to normal form, and if this gives a numeral ν_i, all the T_j but for T_i are cancelled.

Whereas normal order reduction corresponds closely to the "call-by-name" concept of ALGOL 60 [25], the algorithm given below uses a technique more like the "delay rule" of Vuillemin [28], which was in essence given earlier by Wadsworth [22] in another context. We shall follow Wadsworth and refer to this technique as "call-by-need", for it corresponds to evaluating parameters only when they are needed (and then remembering the result in case the parameter is needed again). More specifically, our algorithm effects simulated substitution of expressions for variables, but "overwrites" the denoted (substituted) expression with its normal form *if* this ever needs to be found. Thus the repeated reduction of the same expression to normal form, which is a cause of great inefficiency in pure normal-order reduction, is avoided. To consider the optimality of this "call-by-need", we restrict our attention for the moment to the ordinary λ-calculus (which is, of course, a sub-language of LAMB).

Vuillemin shows in [28] that "call-by-need" is in general more efficient than either "call-by-value" or "call-by-name" for computations (i.e. reductions) in the language he takes. However that language is in reality a version of the λ-calculus with a *restricted* form of β-reduction, and in fact it seems that the result does not quite extend to the full λ-calculus. To see why not, let us consider how a reduction algorithm works.

The general form of a "safe" λ-reduction algorithm is: find the left-most redex; replace it by its reduct; repeat.

This is most easily implemented as a pair of mutually-recursive functions (following Wadsworth [22] again): *ReducetoNormalForm* and *ReducetoLambdaForm*. Modified versions of these functions ($\mathcal{R}_1$ and $\mathcal{R}_2$ respectively), embodying the call-by-need mechanism, are described in Table 9 in the notation of semantic equations (using continuations to indicate temporal succession[7]). It can be seen that the algorithm is less than optimal when reducing

(1) $(\lambda f.\ (fa)f)(\lambda y.\ b(\underline{(\lambda z.\ zy)c}))$

to its normal form:

(2) $(b(ca))(\lambda y.\ b(cy))$.

The underlined redex can in fact be contracted independently of any substitution for y. It is not apparent how the call-by-need mechanism could be adapted to foresee this.

However, if we think of (1) above in terms of programming a computation, we might write it as:

(3) let $f(y) = b($let $z = c$ in $z(y))$
in $f(a)(f)$.

But of course, if efficiency is of concern, a programmer would probably avoid re-defining z in each call of f, as follows:

(4) let $z = c$ in
let $f(y) = b(z(y))$ in $f(a)(f)$.

We maintain that expressions for which the call-by-need mechanism is suboptimal correspond closely to instances of "bad programming" of computations. In our intended application of the algorithm, as described in Chapter 4, we shall not be wanting to reduce completely arbitrary λ-expressions to normal form, only ones which have been produced by our compiler-generator or by the writer of a semantic description. Hence we shall not worry about the non-optimality of our algorithm - we are satisfied that it is

(*) For example, an ALGOL-like version of the first equation for $\mathcal{R}_\mu$ might be:

```
procedure ReduceAbstraction(μ, ⟦λι_i. ε⟧, ρ, χ);
     integer μ;  tree ⟦λι_i. ε⟧;  env ρ;  label χ;
if μ=2
then begin res_1:=⟦λι_i. ε⟧;  comment res_1, res_2 are global;
           res_2:=ρ;
           goto χ
     end
else begin NewId(χ_1);          comment assigns to res_1;
     χ_1: begin tree ι_j; ι_j:=res_1;
              Lay(ρ, ι_j, nullenv, ι_i, χ_2);
          χ_2: begin env ρ_1; ρ_1:=res_1;
                   ReduceGeneral(1, ε, ρ_1, χ_3);
               χ_3: begin tree ε_1; env ρ'_1; ε_1:=res_1; ρ'_1:=res_2;
                        res_1:=⟦λι_j. ε_1⟧ ; res_2:=nullenv;
                        goto χ
                    end
               end
          end
     end;
```

substantially more efficient than pure normal-order reduction in terms of the number of contractions needed to find a normal-form - and no "expensive" copying or scanning operations are necessary to achieve this efficiency.

We now leave considerations of efficiency, and turn to the definition of an interpreter (reduction algorithm) for LAMB. This is given in Table 10, which extends the algorithm of Table 9 with reduction of tuples, conditionals and various operators. The operation of the algorithm should be clear if the "continuation application" infix operator ‖ (borrowed from MSL for convenience) is read operationally, i.e. $f \parallel g$ is read as "do f then do g", and $f \parallel \lambda x.g$ as "do f and assign the result to x then do g". But note that, to avoid a plethora of tuple-brackets, "Curried" continuations are used; hence often more than one result is "assigned".(*)

As mentioned in the introduction to this Chapter, we are not able to offer a proof of the correctness of the LAMB-interpreter here, and our belief in its correctness comes only from observing that a straight-forward coding of the interpreter in BCPL does give the expected results when reducing large and complex LAMB-terms. Some examples of this are given in Chapter 5 and in the Appendices. Before that, Chapter 4 will describe the compiler-generator system (which is the *raison d'être* of the LAMB-interpreter) in some detail.

TABLE 9: λ-Calculus Reduction Algorithm

Syntax

$\varepsilon \in$ Ex $\varepsilon ::= \lambda\iota_i.\ \varepsilon \mid \varepsilon(\varepsilon) \mid \iota_i$

$\iota_i \in$ Ide identifiers $\iota_0, \iota_1, \ldots$ of LAMBDA.

Domains

```
ρ ∈ Env = [Ide → Loc]          ! identifiers denote locations of Map
σ ∈ S   = [Map x Ide x Loc]    ! states of the algorithm (see p. 88)
φ ∈ Map = [Loc → Val]          ! gives contents of locations in Env
ν ∈ Loc = Int                  ! primitive locations
δ ∈ Val = [Ex x Env]           ! closures
χ ∈ X   = [Ex → Env → C]       ! "Curried" continuations
κ ∈ K   = [[Val + Env + Ide] → C]  ! continuations
θ ∈ C   = [S → S]              ! continuations
```

Algorithm

The functions $\mathcal{R}_1$, $\mathcal{R}_2$ defined below "co-operate" to find the normal form of a λ-expression. $\mathcal{R}_1[\![\varepsilon]\!]\rho\chi$ reduces ε to normal form, relative to the environment ρ, and supplies the resulting expression and (null) environment to the "closure continuation" χ. It uses $\mathcal{R}_2$ to try to reduce the left part of a combination to lambda form, and then uses $\mathcal{J}_1$ to contract the redex (if possible). $\mathcal{R}_2$, apart from returning a λ-abstraction if found, is defined similarly to $\mathcal{R}_1$.

TABLE 9 (ctd.)

$$\mathcal{R}_\mu \in [Ex \to Env \to X \to C] \qquad (\mu = 1, 2)$$

$$\begin{aligned}
\mathcal{R}_\mu[\![\lambda \iota_i.\ \varepsilon]\!]\rho\chi \quad = \quad & \mu{=}2 \to \chi[\![\lambda \iota_i.\ \varepsilon]\!]\rho, \\
& \mathit{NewId} \parallel \lambda \iota_j. \\
& \mathit{Lay}(\rho)\langle [\![\iota_j]\!], \mathit{nullenv}\rangle [\![\iota_i]\!] \parallel \lambda \rho_1. \\
& \mathcal{R}_1[\![\varepsilon]\!]\rho_1 \parallel \lambda \varepsilon_1.\lambda \rho_1'. \\
& \chi[\![\lambda \iota_j.\ \varepsilon_1]\!](\mathit{nullenv})
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}_\mu[\![\varepsilon_1(\varepsilon_2)]\!]\rho\chi \quad = \quad & \mathcal{R}_2[\![\varepsilon_1]\!]\rho \parallel \lambda \varepsilon_1'.\lambda \rho_1'. \\
& \mathcal{S}_\mu[\![\varepsilon_1'(\varepsilon_2)]\!]\rho_1'\rho\chi
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}_\mu[\![\iota_i]\!]\rho\chi \quad = \quad & \mathit{IsFree}(\rho)[\![\iota_i]\!] \to \chi[\![\iota_i]\!](\mathit{nullenv}), \\
& \mathit{Pick}(\rho)[\![\iota_i]\!] \parallel \lambda\langle \varepsilon_1, \rho_1\rangle. \\
& \mathcal{R}_\mu[\![\varepsilon_1]\!]\rho_1 \parallel \lambda \varepsilon_2.\lambda \rho_2. \\
& \mathit{ReLay}(\rho)\langle [\![\varepsilon_2]\!], \rho_2\rangle [\![\iota_i]\!] \parallel \\
& \chi[\![\varepsilon_2]\!]\rho_2
\end{aligned}$$

$$\mathcal{S}_\mu \in [Ex \to Env \to Env \to X \to C] \qquad (\mu = 1, 2)$$

$$\begin{aligned}
\mathcal{S}_\mu[\![[\![\lambda \iota_i.\ \varepsilon]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi \quad = \quad & \mathit{Lay}(\rho_1)\langle [\![\varepsilon_2]\!], \rho_2\rangle [\![\iota_i]\!] \parallel \lambda \rho. \\
& \mathcal{R}_\mu[\![\varepsilon]\!]\rho\chi
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_\mu[\![\varepsilon_1'(\varepsilon_2)]\!]\rho_1'\rho_2\chi \quad = \quad & \mathcal{R}_1[\![\varepsilon_2]\!]\rho_2 \parallel \lambda \varepsilon_2'.\lambda \rho_2'. \\
& \chi[\![\varepsilon_1'(\varepsilon_2')]\!](\mathit{nullenv})
\end{aligned}$$

(The convention of §3.3 implies that the first equation above for $\mathcal{S}_\mu$ has "priority" over the second equation.)

TABLE 9 (ctd.)

Implementation

The following functions could be defined in other ways, corresponding to different choices of the domains S and Env.

```
NewId(κ)⟨ϕ, ι_i, ν⟩ =                          ! NewId ∈ [K → C]
        κ⟦ι_{i+1}⟧⟨ϕ, ι_{i+1}, ν⟩
        ! The Ide component of S is the last used identifier.

Lay(ρ)(δ)⟦ι_i⟧(κ)⟨ϕ, ι_j, ν⟩ =                  ! Lay ∈ [Env → Val → Ide → K → C]
        κ(ρ[ν+1/ι_i])⟨ϕ[δ/ν+1], ι_j, ν+1⟩
        ! The Loc component of S is the last used location.

nullenv =                                      ! nullenv ∈ Env
        (λι_i. err)

IsFree(ρ)⟦ι_i⟧ =                               ! IsFree ∈ [Env → Ide → Bool]
        (ρ⟦ι_i⟧ = err)

Pick(ρ)⟦ι_i⟧(κ)⟨ϕ, ι_j, ν⟩ =                   ! Pick ∈ [Env → Ide → K → C]
        κ(ϕ(ρ⟦ι_i⟧))⟨ϕ, ι_i, ν⟩
        ! uses the Map component of S.

ReLay(ρ)⟦ι_i⟧(θ)⟨ϕ, ι_j, ν⟩ =                   ! ReLay ∈ [Env → Ide → C → C]
        θ⟨ϕ[δ/ρ⟦ι_i⟧], ι_j, ν⟩
        ! updates the Map component of S.
```

TABLE 10: LAMB Reduction Algorithm

Syntax: as defined in Table 4, with

$\varepsilon \in ExpB$

$\beta \in Bvs$

$\omega \in Op_1 + Op_2$

$\iota_i \in I$

$\nu_i \in Num$

$\sigma_i \in String$

Domains:

$\rho \in Env = [I \to Loc]$

$S = [Map \times I \times Loc]$

$Map = [I \to Val]$

$Loc = Int$

$Val = [ExpB \times Env]$

$\chi \in X = [ExpB \to Env \to C]$

$\kappa \in K = [[Val + Env + Bvs] \to C]$

$C = [S \to S]$

Algorithm: $\mathcal{R}_\mu \in [ExpB \to Env \to X \to C] \quad (\mu = 1, 2, 3)$

$! \mathcal{R}_1[\![\varepsilon]\!]\rho\chi$ reduces ε to normal form

$! \mathcal{R}_2[\![\varepsilon]\!]\rho\chi$ reduces ε to lambda-form

$! \mathcal{R}_3[\![\varepsilon]\!]\rho\chi$ reduces ε to tuple-form

$$\mathcal{R}_\mu[\![\lambda\beta.\ \varepsilon]\!]\rho\chi = \mu=2 \to \chi[\![\lambda\beta.\ \varepsilon]\!]\rho,$$
$$NewBvs[\![\beta]\!] \parallel \lambda\beta_1.$$
$$LayBvs(\rho)\langle[\![\beta_1]\!], nullenv\rangle[\![\beta]\!] \parallel \lambda\rho_1.$$
$$\mathcal{R}_1[\![\varepsilon]\!]\rho_1 \parallel \lambda\varepsilon_1.\lambda\rho_1'.$$
$$\chi[\![\lambda\beta_1.\ \varepsilon_1]\!](nullenv)$$

TABLE 10 (ctd.)

$\mathcal{R}_\mu[\![$ val β . $\varepsilon]\!]\rho\chi$ = similar to $[\![\lambda\beta$. $\varepsilon]\!]$

$\mathcal{R}_\mu[\![$ fix β . $\varepsilon]\!]\rho\chi$ = $\mu=2 \vee \mu=3 \rightarrow$ *LayBvs*$(\rho)\langle[\![$ fix β . $\varepsilon]\!], \rho\rangle[\![\beta]\!] \parallel \lambda\rho_1$.
$\mathcal{R}_\mu[\![\varepsilon]\!]\rho_1\chi$,
NewBvs$[\![\beta]\!] \parallel \lambda\beta_1$.
LayBvs$(\rho)\langle[\![\beta_1]\!], nullenv\rangle[\![\beta]\!] \parallel \lambda\rho_1$.
$\mathcal{R}_1[\![\varepsilon]\!]\rho_1 \parallel \lambda\varepsilon_1.\lambda\rho_1'$.
$\chi[\![$ fix β_1 . $\varepsilon_1]\!]$*(nullenv)*

$\mathcal{R}_\mu[\![\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3]\!]\rho\chi$ = $\mathcal{R}_1[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'$.
$\mathcal{B}_\mu[\![\varepsilon_1' \rightarrow \varepsilon_2, \varepsilon_3]\!]\rho\chi$

$\mathcal{R}_\mu[\![\varepsilon_1 \dot{-} \varepsilon_2]\!]\rho\chi$ = $\mu=2 \rightarrow \chi[\![\varepsilon_1 \dot{-} \varepsilon_2]\!]\rho$,
$\mathcal{R}_1[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'$.
$\mathcal{R}_1[\![\varepsilon_2]\!]\rho \parallel \lambda\varepsilon_2'.\lambda\rho_2'$.
$\chi[\![\varepsilon_1' \dot{-} \varepsilon_2']\!]$*(nullenv)*

$\mathcal{R}_\mu[\![\varepsilon_1 \dot{*} \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 \dot{-} \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 \parallel \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 \dot{-} \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 + \varepsilon_2]\!]\rho\chi$ = $\mathcal{R}_1[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'$.
$\mathcal{R}_1[\![\varepsilon_2]\!]\rho \parallel \lambda\varepsilon_2'.\lambda\rho_2'$.
$\mathcal{O}_2[\![\varepsilon_1' + \varepsilon_2']\!]\chi$

$\mathcal{R}_\mu[\![\varepsilon_1 - \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 \times \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 = \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 \leq \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 \vee \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

TABLE 10 (ctd.)

$\mathcal{R}_\mu[\![\varepsilon_1 \wedge \varepsilon_2]\!]\rho\chi$ = similar to $[\![\varepsilon_1 + \varepsilon_2]\!]$

$\mathcal{R}_\mu[\![\varepsilon_1 \text{ cat } \varepsilon_2]\!]\rho\chi$ = $\mu=3 \rightarrow \chi[\![\varepsilon_1 \text{ cat } \varepsilon_2]\!]\rho,$
$\mathcal{R}_1[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'.$
$\mathcal{R}_1[\![\varepsilon_2]\!]\rho \parallel \lambda\varepsilon_2'.\lambda\rho_2'.$
$\mathcal{O}_2[\![\varepsilon_1' \text{ cat } \varepsilon_2']\!]\chi$

$\mathcal{R}_\mu[\![\varepsilon_1 \downarrow \varepsilon_2]\!]\rho\chi$ = $\mathcal{R}_3[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'.$
$\mathcal{R}_1[\![\varepsilon_2]\!]\rho \parallel \lambda\varepsilon_2'.\lambda\rho_2'.$
$\mathcal{J}_\mu[\![\varepsilon_1' \downarrow \varepsilon_2']\!]\rho_1'\chi$

$\mathcal{R}_\mu[\![\text{pair } \varepsilon]\!]\rho\chi$ = $\mu=2 \rightarrow \chi[\![\text{pair } \varepsilon]\!]\rho,$
$\mathcal{R}_1[\![\varepsilon]\!]\rho \parallel \lambda\varepsilon'.\lambda\rho'.$
$\chi[\![\text{pair } \varepsilon']\!]$ *(nullenv)*

$\mathcal{R}_\mu[\![\sim \varepsilon]\!]\rho\chi$ = $\mathcal{R}_1[\![\varepsilon]\!]\rho \parallel \lambda\varepsilon'.\lambda\rho'.$
$\mathcal{O}_1[\![\sim \varepsilon']\!]\chi$

$\mathcal{R}_\mu[\![\text{dim } \varepsilon]\!]\rho\chi$ = $\mathcal{R}_3[\![\varepsilon]\!]\rho \parallel \lambda\varepsilon'.\lambda\rho'.$
$\mathcal{D}[\![\text{dim } \varepsilon']\!]\rho'\chi$

$\mathcal{R}_\mu[\![\text{isatom } \varepsilon]\!]\rho\chi$ = $\mathcal{R}_3[\![\varepsilon]\!]\rho \parallel \lambda\varepsilon'.\lambda\rho'.$
$\mathcal{A}[\![\text{isatom } \varepsilon']\!]\rho'\chi$

$\mathcal{R}_\mu[\![\text{imp } \varepsilon]\!]\rho\chi.$ = $\mathcal{R}_1[\![\varepsilon]\!]\rho \parallel \lambda\varepsilon'.\lambda\rho'.$
$\chi[\![\text{imp } \varepsilon']\!]$ *(nullenv)*

$\mathcal{R}_\mu[\![\varepsilon_1(\varepsilon_2)]\!]\rho\chi$ = $\mathcal{R}_2[\![\varepsilon_1]\!]\rho \parallel \lambda\varepsilon_1'.\lambda\rho_1'.$
$\mathcal{S}_\mu[\![\varepsilon_1'(\varepsilon_2)]\!]\rho_1'\rho\chi$

TABLE 10 (ctd.)

$$\mathcal{R}_\mu[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!]\rho\chi = \mu=3 \to \chi[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!]\rho,$$
$$\mathcal{R}_1[\![\varepsilon_1]\!]\rho \;\|\; \lambda\varepsilon_1.\lambda\rho_1.$$
$$\mathcal{R}_1[\![\varepsilon_2]\!]\rho \;\|\; \lambda\varepsilon_2.\lambda\rho_2.$$
$$\ldots$$
$$\mathcal{R}_1[\![\varepsilon_n]\!]\rho \;\|\; \lambda\varepsilon_n.\lambda\rho_n.$$
$$\chi[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\langle\rangle]\!]\rho\chi = \chi[\![\langle\rangle]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\iota_i]\!]\rho\chi = IsFree(\rho)[\![\iota_i]\!] \to \chi[\![\iota_i]\!](nullenv),$$
$$Pick(\rho)[\![\iota_i]\!] \;\|\; \lambda\langle \varepsilon_1,\rho_1\rangle.$$
$$\mathcal{R}_\mu[\![\varepsilon_1]\!]\rho_1 \;\|\; \lambda\varepsilon_2.\lambda\rho_2.$$
$$ReLay(\rho)\langle[\![\varepsilon_2]\!],\rho_2\rangle[\![\iota_i]\!] \;\|$$
$$\chi[\![\varepsilon_2]\!]\rho_2$$

$$\mathcal{R}_\mu[\![\nu_i]\!]\rho\chi = \chi[\![\nu_i]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\sigma_i]\!]\rho\chi = \chi[\![\sigma_i]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\mathtt{true}]\!]\rho\chi = \chi[\![\mathtt{true}]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\mathtt{false}]\!]\rho\chi = \chi[\![\mathtt{false}]\!](nullenv)$$

$$\mathcal{R}_\mu[\![\mathtt{err}]\!]\rho\chi = \chi[\![\mathtt{err}]\!](nullenv)$$

$$\mathcal{B}_\mu \in [ExpB \to Env \to X \to C] \qquad (\mu = 1, 2, 3)$$

$$\mathcal{B}_\mu[\![\mathtt{true} \to \varepsilon_2, \varepsilon_3]\!]\rho\chi = \mathcal{R}_\mu[\![\varepsilon_2]\!]\rho\chi$$

$$\mathcal{B}_\mu[\![\mathtt{false} \to \varepsilon_2, \varepsilon_3]\!]\rho\chi = \mathcal{R}_\mu[\![\varepsilon_3]\!]\rho\chi$$

$$\mathcal{B}_\mu[\![\varepsilon_1' \to \varepsilon_2, \varepsilon_3]\!]\rho\chi = \mathcal{R}_1[\![\varepsilon_2]\!]\rho \;\|\; \lambda\varepsilon_2'.\lambda\rho_2'.$$
$$\mathcal{R}_1[\![\varepsilon_3]\!]\rho \;\|\; \lambda\varepsilon_3'.\lambda\rho_3'.$$
$$\chi[\![\varepsilon_1' \to \varepsilon_2', \varepsilon_3']\!](nullenv)$$

TABLE 10 (ctd.)

$$\mathcal{O}_2 \in [\text{ExpB} \to X \to C]$$

$$\mathcal{O}_2[\![\varepsilon_1\ \omega\ \varepsilon_2]\!]\chi = (\chi \circ \mathcal{C})[\![\varepsilon_1\ \omega\ \varepsilon_2]\!](nullenv)$$

$$\mathcal{O}_1 \in [\text{ExpB} \to X \to C]$$

$$\mathcal{O}_1[\![\omega\ \varepsilon]\!]\chi = (\chi \circ \mathcal{C})[\![\omega\ \varepsilon]\!](nullenv)$$

$$\mathcal{C} \in [\text{ExpB} \to \text{ExpB}]$$

$\mathcal{C}[\![\varepsilon]\!]$ is as defined in Table 8.

$$\mathcal{J}_\mu \in [\text{ExpB} \to \text{Env} \to X \to C] \qquad (\mu = 1, 2, 3]$$

$$\mathcal{J}_\mu[\![\langle \varepsilon_1,\ldots,\varepsilon_n\rangle \downarrow \nu_i]\!]\rho\chi = \mathcal{R}_\mu[\![\varepsilon_i]\!]\rho\chi$$

$$\mathcal{J}_\mu[\![[\![\varepsilon_1\ \text{cat}\ \varepsilon_2]\!] \downarrow \nu_i]\!]\rho\chi = \mathcal{R}_\mu[\![\ [\![\nu_i \le [\![\dim\ \varepsilon_1]\!]]\!] \to \varepsilon_1 \downarrow \nu_i,\ [\![\varepsilon_2 \downarrow [\![\nu_i - [\![\dim\ \varepsilon_1]\!]]\!]]\!]\ ]\!]\rho\chi$$

$$\mathcal{J}_\mu[\![\varepsilon_1 \downarrow \varepsilon_2']\!]\rho\chi = \mathcal{R}_1[\![\varepsilon_1]\!]\rho\ !\ \lambda\varepsilon_1'.\lambda\rho_1'.\ \chi[\![\varepsilon_1' \downarrow \varepsilon_2']\!](nullenv)$$

$$\mathcal{D} \in [\text{ExpB} \to \text{Eny} \to X \to C]$$

$$\mathcal{D}[\![\dim\ \langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!]\rho\chi = \chi[\![\nu_n]\!](nullenv)$$

$$\mathcal{D}[\![\dim\ \langle\rangle]\!]\rho\chi = \chi[\![\nu_0]\!](nullenv)$$

$$\mathcal{D}[\![\dim\ [\![\varepsilon_1\ \text{cat}\ \varepsilon_2]\!]]\!]\rho\chi = \mathcal{R}_1[\![\ [\![\dim\ \varepsilon_1]\!] + [\![\dim\ \varepsilon_2]\!]\ ]\!]\rho\chi$$

$$\mathcal{D}[\![\dim\ \varepsilon]\!]\rho\chi = \mathcal{R}_1[\![\varepsilon]\!]\rho\ !\ \lambda\varepsilon'.\lambda\rho'.\chi[\![\dim\ \varepsilon']\!](nullenv)$$

TABLE 10 (ctd.)

$$\mathcal{A} \in [\mathrm{ExpB} \to \mathrm{Env} \to \mathrm{X} \to \mathrm{C}]$$

$$\mathcal{A}[\![\text{isatom } \langle \varepsilon_1,\ldots,\varepsilon_n\rangle]\!]\rho\chi = \chi[\![\text{false}]\!](\textit{nullenv})$$

$$\mathcal{A}[\![\text{isatom } \langle\rangle]\!]\rho\chi = \chi[\![\text{false}]\!](\textit{nullenv})$$

$$\mathcal{A}[\![\text{isatom } [\![\varepsilon_1 \text{ cat } \varepsilon_2]\!]]\!]\rho\chi = \mathcal{R}_1[\![\sim[\![[\![\text{isatom } \varepsilon_1]\!] \vee [\![\text{isatom } \varepsilon_2]\!]]\!]]\!]\rho\chi$$

$$\mathcal{A}[\![\text{isatom } \nu_i]\!]\rho\chi = \chi[\![\text{true}]\!](\textit{nullenv})$$

$$\mathcal{A}[\![\text{isatom } \sigma_i]\!]\rho\chi = \chi[\![\text{true}]\!](\textit{nullenv})$$

$$\mathcal{A}[\![\text{isatom } \varepsilon]\!]\rho\chi = \mathcal{R}_1[\![\varepsilon]\!]\rho \;\|\; \lambda\varepsilon'.\lambda\rho'.\ \chi[\![\text{isatom } \varepsilon']\!](\textit{nullenv})$$

$$\mathcal{J}_\mu \in [\mathrm{ExpB} \to \mathrm{Env} \to \mathrm{Env} \to \mathrm{X} \to \mathrm{C}] \qquad (\mu = 1, 2, 3)$$

$$\mathcal{J}_\mu[\![[\![\lambda\beta.\ \varepsilon]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi = \textit{LayBvs}(\rho_1)\langle[\![\varepsilon_2]\!],\rho_2\rangle[\![\beta]\!] \;\|\; \lambda\rho.\ \mathcal{R}_\mu[\![\varepsilon]\!]\rho\chi$$

$$\mathcal{J}_\mu[\![[\![\text{val } \beta.\ \varepsilon]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{R}_1[\![\varepsilon_2]\!]\rho_2 \;\|\; \lambda\varepsilon_2'.\lambda\rho_2'.\ \textit{LayBvs}(\rho_1)\langle[\![\varepsilon_2]\!],\rho_2'\rangle[\![\beta]\!] \;\|\; \lambda\rho.\ \mathcal{R}_\mu[\![\varepsilon]\!]\rho\chi$$

$$\mathcal{J}_\mu[\![[\![\varepsilon_0 \doteq \varepsilon_1]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{L}[\![\varepsilon_2]\!]\rho_2\rho_1 \;\|\; \lambda\iota_i.\lambda\rho_3.\ \textit{NewId} \;\|\; \lambda\iota_j.\ \mathcal{R}_\mu[\![[\![\text{val } \iota_j.[\![\varepsilon_1(\iota_j)]\!]]\!](\varepsilon_0(\iota_i))]\!]\rho_3\chi$$

$$\mathcal{J}_\mu[\![[\![\varepsilon_0 \doteq\!\!\!\!\!\!* \ \varepsilon_1]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{L}[\![\varepsilon_2]\!]\rho_2\rho_1 \;\|\; \lambda\iota_i.\lambda\rho_3.\ \textit{NewId} \;\|\; \lambda\iota_j.\ \mathcal{R}_\mu[\![[\![\text{val } \iota_j.[\![[\![\varepsilon_1(\iota_j\downarrow\nu_1)]\!](\iota_j\downarrow\nu_2)]\!]\!](\varepsilon_0(\iota_i))]\!]\rho_3\chi$$

$$\mathcal{J}_\mu[\![[\![\varepsilon_0 \parallel \varepsilon_1]\!](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{L}[\![\varepsilon_2]\!]\rho_2\rho_1 \;\|\; \lambda\iota_i.\lambda\rho_3.\ \mathcal{R}_\mu[\![[\![\varepsilon_0(\varepsilon_1)]\!](\iota_i)]\!]\rho_3\chi$$

TABLE 10 (ctd.)

$$\mathcal{J}_\mu[\![\text{pair } \varepsilon_1](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{L}[\![\varepsilon_2]\!]\rho_2\rho_1 \,\|\, \lambda\iota_i.\lambda\rho_3.\ \mathcal{R}_\mu[\![\langle \varepsilon_1, \iota_i\rangle]\!]\rho_3\chi$$

$$\mathcal{J}_\mu[\![\text{imp } \sigma_i](\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{R}_1[\![\varepsilon_2]\!]\rho_2 \,\|\, \lambda\varepsilon_2'.\lambda\rho_2'.\ (\chi\circ ImpFn(i))[\![\varepsilon_2]\!](nullenv)$$

$$\mathcal{J}_\mu[\![\varepsilon_1'(\varepsilon_2)]\!]\rho_1\rho_2\chi = \mathcal{R}_1[\![\varepsilon_2]\!]\rho_2 \,\|\, \lambda\varepsilon_2'.\lambda\rho_2'.\ \chi[\![\varepsilon_1'(\varepsilon_2')]\!](nullenv)$$

$$\mathcal{L} \in [\text{ExpB} \to \text{Env} \to \text{Env} \to \text{X} \to \text{C}]$$

$$\mathcal{L}[\![\varepsilon]\!]\rho\rho_1\chi = \mathcal{R}_1[\![\varepsilon]\!]\rho \,\|\, \lambda\varepsilon'.\lambda\rho'.\ NewId \,\|\, \lambda\iota_i.\ Lay(\rho_1)\langle[\![\varepsilon']\!], nullenv\rangle[\![\iota_i]\!] \,\|\, \lambda\rho_1'.\ \chi[\![\iota_i]\!]\rho_1'$$

$$NewBvs \in [\text{Bvs} \to \text{K} \to \text{C}]$$

$$NewBvs[\![\iota_i]\!](\kappa) = NewId(\kappa)$$

$$NewBvs[\![\langle \beta_1, \ldots, \beta_n\rangle]\!](\kappa) = NewBvs[\![\beta_1]\!] \,\|\, \lambda\beta_1'.\ NewBvs[\![\beta_2]\!] \,\|\, \lambda\beta_2'.\ \ldots\ \kappa[\![\langle \beta_1', \ldots, \beta_n'\rangle]\!]$$

$$NewBvs[\![\langle\rangle]\!](\kappa) = \kappa[\![\langle\rangle]\!]$$

TABLE 10 (ctd.)

$$LayBvs \in [Env \to Val] \to Bvs \to K \to C]$$

$$LayBvs(\rho_0)\langle \varepsilon, \rho\rangle [\![\iota_i]\!](\kappa) = Lay(\rho_0)\langle \varepsilon, \rho\rangle [\![\iota_i]\!](\kappa)$$

$$\begin{aligned} LayBvs(\rho_0)\langle \varepsilon, \rho\rangle [\![\langle \beta_1, \ldots, \beta_n \rangle]\!](\kappa) = {} & LayBvs(\rho_0)\langle [\![\varepsilon{\downarrow}\nu_1]\!], \rho\rangle [\![\beta_1]\!] \; ! \; \lambda\rho_1. \\ & LayBvs(\rho_1)\langle [\![\varepsilon{\downarrow}\nu_2]\!], \rho\rangle [\![\beta_2]\!] \; ! \; \lambda\rho_2. \\ & \quad \ldots \\ & LayBvs(\rho_{n-1})\langle [\![\varepsilon{\downarrow}\nu_n]\!], \rho\rangle [\![\beta_n]\!] \; ! \; \lambda\rho_n. \\ & \kappa(\rho_n) \end{aligned}$$

$$LayBvs(\rho_0)\langle \varepsilon, \rho\rangle [\![\langle\rangle]\!](\kappa) = \kappa(\rho_0)$$

Implementation:

The following are defined as in Table 9.

$NewId \in [K \to C]$

$Lay \in [Env \to Val \to I \to K \to C]$

$nullenv \in Env$

$IsFree \in [Env \to I \to Bool]$

$Pick \in [Env \to I \to K \to C]$

$ReLay \in [Env \to Val \to I \to C \to C]$

CHAPTER 4

A Compiler-Generator

In this chapter we present a compiler-generator based on mathematical semantics. The compiler-generator produces a compiler for a programming language from an unambiguous context-free grammar (written in a BNF-like notation called GRAMMA) and a semantic description (written in MSL). The compiler may be applied to a program text using the system to produce the code of that program, which in turn may be run after giving MSL descriptions of its input and of the implementation of any functions left undefined by the semantic description. The system - all of whose parts are coded in LAMB - operates by combining the code of a function with the code of its argument in a formal application, using a LAMB-interpreter to reduce this application to the code of the result.

We regard §4.1, which discusses the structure and operation of the compiler-generator system, as the main section of this chapter. §4.2 describes the syntax of GRAMMA, and §§4.3, 4.4 use MSL to describe a grammar-driven parser. The parser (which uses the so-called SLR(1) Algorithm) constitutes a semantics for GRAMMA, but the details of its description are not important, as many other algorithms could have been used instead - it is included mainly to demonstrate the feasibility of "programming" in MSL.

"Code generation" is not discussed separately from the general operation of the system, because we consider a compiler in the abstract to be a function from texts to values, rather than from texts to code. The values become encoded automatically (in LAMB) when we simulate the application of abstract functions by manipulating LAMB-expressions.

§4.1 Structure of the Compiler-Generator

We first consider the abstract structure of the compiler-generator, discussing abstract functions and values without reference to any concrete code. Later we shall show how LAMB can be used as a code for all components of the system, enabling an "operational simulation of the abstract entities".

For our purposes here, an abstract compiler is *not* a function from abstract text to abstract code - rather it is a function from abstract text to abstract *values*, which are often themselves functions. It will be convenient to consider an abstract text to be a *tuple* of abstract characters. An abstract parser is a function from abstract text to abstract parse-trees, and a (mathematical) semantics is a function from these parse-trees to abstract values; hence an abstract compiler may be formed as the composition of such a parser and semantics. These conventions enable us to base our compiler-generator directly on mathematical semantics.

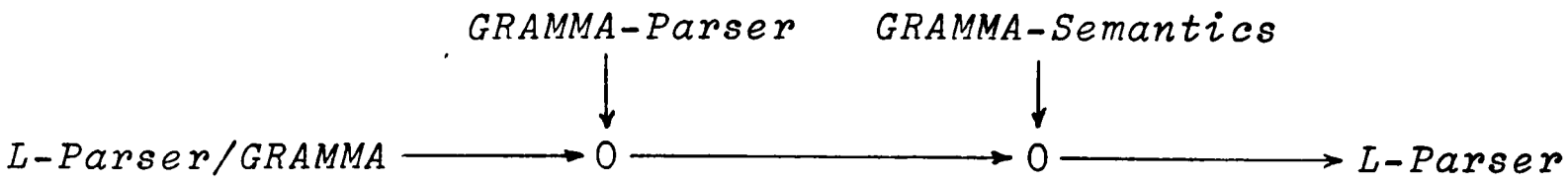
We shall illustrate the components of the abstract compiler-generator using $x \longrightarrow \overset{f}{\underset{\downarrow}{0}} \longrightarrow y$ to denote that $f(x) = y$. An abstract text denoting a function F and written in a language L will be denoted by F/L.

Hence an abstract compilation of a *Program* in language L would look like:

(1)
$$\textit{Program/L} \longrightarrow \overset{\textit{L-Parser}}{\underset{\downarrow}{0}} \longrightarrow \overset{\textit{L-Semantics}}{\underset{\downarrow}{0}} \longrightarrow \textit{Program}\ .$$

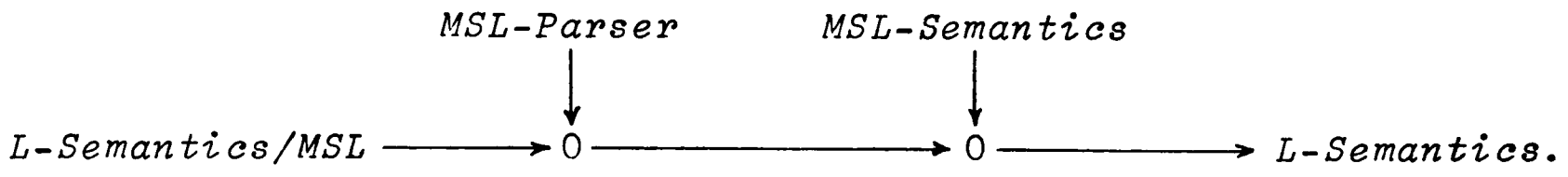
It is the purpose of the compiler-generator to produce *L-Parser* and *L-Semantics* from their written descriptions in GRAMMA and MSL respectively. This is accomplished by

(2)



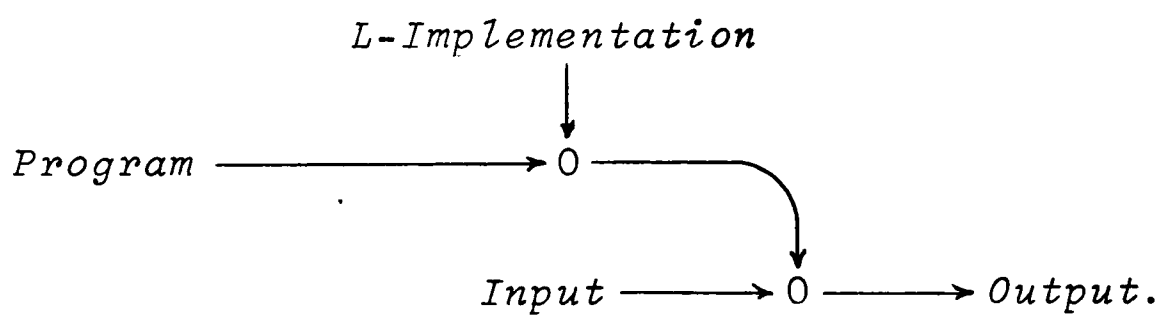
and

(3)



Usually a mathematical semantics does not completely specify the semantic value of a program, preferring instead to make it dependent on certain rather primitive functions by leaving them undefined in the description. Typically these functions might effect the accessing and updating of locations in an abstract store, or control the "order" in which the sub-expressions of an expression in a language are evaluated. Clearly we need to supply the values of these functions before we can evaluate a program to map input to output. It is quite convenient to describe them in MSL and then use the components of (3) to produce a function *L-Implementation* such that a program is run by:

(4)



The normal use of the abstract compiler-generator is just producing an abstract compile-and-run system; but we can also use it to form a new compiler-generator, based on new description languages, say XGRAMMA and XMSL. We do this by giving formal descriptions of the new notations (i.e. their syntax and semantics) in GRAMMA and MSL, and then using (2) and (3) to produce the corresponding functions *XGRAMMA-Parser, -Semantics* and *XMSL-Parser, -Semantics*. By taking XGRAMMA and XMSL to be simply GRAMMA and MSL respectively, we can make a test for the consistency of the system: the new compiler-generator produced from the circular descriptions must be equivalent

to the original one. In other words the effect should be the same whether

(i) the original components of the compiler-generator are used to interpret the circular descriptions in order to interpret an ordinary description; or

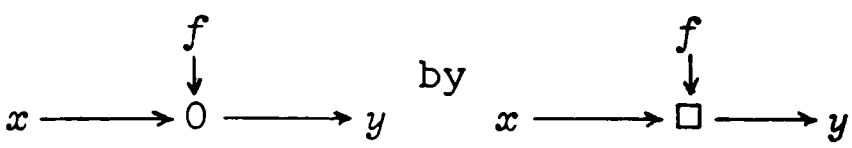
(ii) the ordinary description is interpreted directly by the original components.

We now show how LAMB is used to obtain a simulation of the abstract compiler-generator on a real computer. Basically we assume that all components of the system can be identified with *computable* elements of the domain Pw (in the sense of §1.3). By the results of Chapter 2 these elements are all LAMB-definable - in fact we shall specify below how to obtain suitable LAMB-definitions. Clearly if ϕ is a LAMB-expression denoting (the graph of) a function f, and χ denotes x, then the expression $\phi(\chi)$ denotes $f(x)$. Hence the simulation of generating, e.g., *L-Parser* in (2) above, consists of simply combining the LAMB-expressions denoting the text *L-Parser/GRAMMA*, the function *GRAMMA/Parser* and the function *GRAMMA/Semantics*, into a single expression.

While the simplicity of this simulation guarantees its correctness, it is obviously not very useful. The expression we obtain in this way denoting *Output* in (4) is always very large, even though *Output* might be only a single integer - and we are not able to intuit directly the value in Pw denoted by a complex LAMB-expression. What we need is for our simulation to reduce the complexity of such an expression without changing the value it denotes, ideally so that it will yield, for instance, a simple numeral of LAMB if the value denoted is an integer.

In Chapter 3 we presumed that a certain reduction algorithm for

LAMB is correct, i.e. value-preserving. That algorithm can be implemented quite easily on a computer: using it we can obtain a suitable simulation of the abstract compiler-generator. Instead of simulating the application of a function by just combining two LAMB-expressions, we combine the two expressions and then reduce the combination to "normal form". Let us denote this simulation of



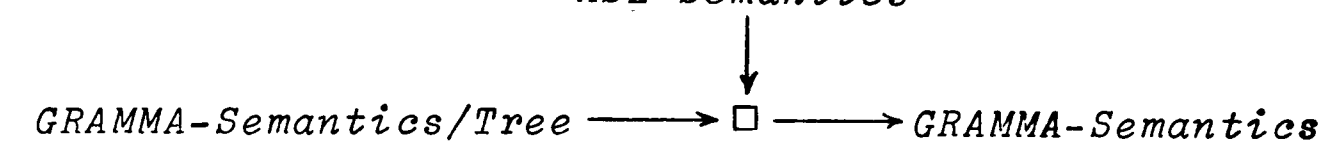
by

, so that diagrams (1) - (4) above illustrate our simulated compiler-generator if "0" is replaced by "□". Usually this simulation will be correct, as the intermediate reductions arising from "sensible" descriptions in MSL will terminate (thanks to our definition of normal form in Chapter 3); but for full generality, reduction must in fact be delayed until the final combination (denoting *Output* in (4)) has been formed.

Of course in an implementation of this simulation, we need to input and output LAMB-expressions; also it is useful to be able to inspect intermediate expressions, and control the points at which reduction is done. We shall not describe how this might be achieved, as it is mostly a matter of interaction with a real operating system.

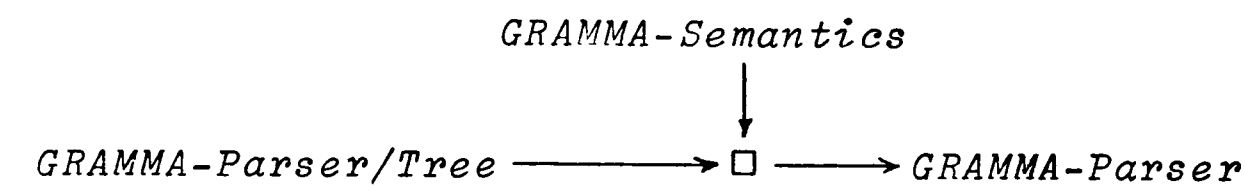
However, we do offer some brief comments on "boot-strapping" the system. The functions in (2) and (3) are rather complex, and clearly it would be a major task to produce their LAMB-definitions by hand. We note that *MSL-Semantics* is denoted by the LAMB-expression in Appendix C, which we used to define MSL in terms of LAMB. If any (independent) grammar-driven syntax analyzer is available, it can be used to create the parse-trees of *GRAMMA-Parser/GRAMMA* and *GRAMMA-Semantics/MSL*, which can easily be converted to LAMB-expressions denoting the abstract parse-trees *GRAMMA-Parser/Tree* and *GRAMMA-Semantics/Tree*. Then

(5)



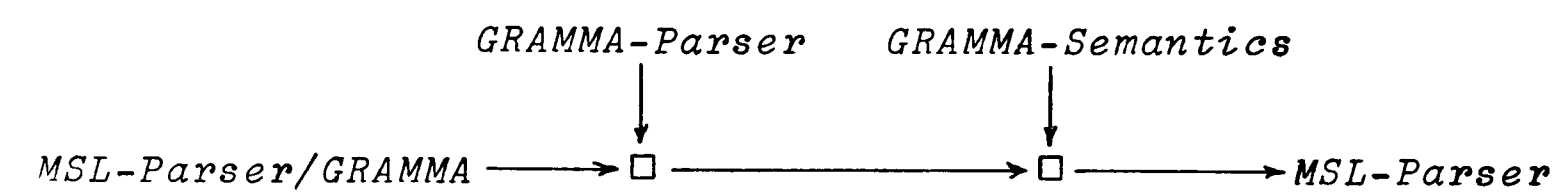
and

(6)



so

(7)



which completes the system.

In the following sections we shall give *GRAMMA-Parser/GRAMMA* and *GRAMMA-Semantics/MSL*. Appendix B contains *MSL-Parser/GRAMMA*, and Appendix C contains *MSL-Semantics*. The MSL segments in Table 11 below combine other segments, as in the boot-strap above, to produce the compiler-generator. (Each segment is delimited by the string which identifies it and a terminating stop (.).) Note that several segments are written directly in LAMB, but that after the initial boot-strapping stage they can be re-written in MSL and compiled into LAMB using the system.

TABLE 11: Compiler-Generator System

```
'GRAMMA-Semantics'

(imp 'SegFn')('MSL-Semantics')(
    (imp 'SegFn')('GRAMMA-Semantics/Tree'))
.



'GRAMMA-Parser'

(imp 'SegFn')('GRAMMA-Semantics')(
    (imp 'SegFn')('GRAMMA-Parser/Tree'))
.
```

```
'MSL-Parser'

(imp 'SegFn')('GRAMMA-Semantics')(
    (imp 'SegFn')('GRAMMA-Parser')(
        (imp 'SegFn')('MSL-Parser/GRAMMA')))
.



'Compiler-Gererator/MSL'

 <LParserGRAMMA, LSemanticsMSL>.

let LParser = (seg 'GRAMMA-Semantics')(
                    (seg 'GRAMMA-Parser')(LParserGRAMMA))

let LSemantics = (seg 'MSL-Semantics')(
                    (seg 'MSL-Parser')(LSemanticsMSL))

result LParser o LSemantics
.
```

§4.2 GRAMMA

In this section we describe the syntax and semantics of GRAMMA, our syntax description language which complements MSL. We consider the semantic value of a description in GRAMMA to be a function from texts to parse-trees, i.e. an abstract parser, and we shall give the semantics of GRAMMA by using MSL to describe a parser driven by the parse-trees of descriptions in GRAMMA. The parser, which is based on the SLR(1) recognizer algorithm [29], has been chosen for its relative simplicity combined with adequate generality; other algorithms could be used instead. We hope that GRAMMA is sufficiently like BNF and other syntax description languages to be understood from the informal comments we offer below - we only include the formal definition (§§4.3 - 4.4) for completeness and to demonstrate the feasibility of "programming" in MSL.

In fact a parser corresponding to a description in GRAMMA is the composition of a lexical analyzer and a syntax analyzer. The lexical analyzer transforms a sequence of characters into a sequence of basic symbols, which makes the syntax analysis both easier (less generality is needed) and more efficient. The lexical analyzer is based on an algorithm which is (much) less general than the SLR(1) algorithm.

Table 12 gives a "circular" description of the syntax of GRAMMA. As in the unambiguous grammars for LAMB and MSL, the language described is a "hardware representation" (c.f. [25]) which may be related to a more elegant "publication language" using Appendix Z.

A grammar in GRAMMA consists of three section, headed by **syn**, **lex** and **aux**. The first section uses an abbreviated BNF-like notation to define syntactic categories recursively. Non-terminal symbols

are formed as sequences of letters, beginning with a capital. Terminal symbols may be the corresponding sequence of characters enclosed in quotes (' '), but complex or "variable" terminal symbols should be named by a sequence of underlined capitals (the suffix _ is used to distinguish variable terminal symbols corresponding to numerals, identifiers, etc.) and specified in the **lex** section (non-recursively). As in BNF, the symbols ::= and | are used as separators. We refer to the specification of a single non-terminal as a *clause*, and to its alternatives as *phrases*. Each clause starts on a new line. After each phrase, following a colon, a string (of LAMB) is specified, to be used when forming a corresponding node in a parse-tree; but this may be omitted when either (i) the phrase consists of a simple terminal symbol (then it is taken as the specification of the label), or (ii) there is exactly one non-terminal symbol in the phrase (then there will be no corresponding node in the parse-tree). The purpose of specifying these strings is so that nodes corresponding to semantically-similar phrases may have the same label; in fact the labels would usually correspond to the phrases of a reduced ambiguous grammar as used in semantic equations. The Brooker *-? convention may be used: (...)*allows one or more occurrences of ... , and (...)? allows zero or more, the brackets being omitted when ... consists of a single named symbol or a single group governed by * or ?.

The **lex** section is simpler: a clause may be of the same form as in the **syn** section, but phrases do not specify labels, and their component symbols may be only single characters (enclosed by quotes ' ') or names of classes of single characters specified in the **aux** section. A clause may also be of the same form as in the **aux** section. There is usually one clause specifying λ, the null basic symbol,

which is ignored in syntax analysis,and so may be used to eliminate comments and layout.

-In the **aux** section, "=" or "≠" is used as a separator instead of "::=". Each phrase of a clause specifies either a single character (enclosed in quotes and possibly using "*n", "*s", "*4" to denote the standard characters NEWLINE, SPACE, FOURSPACES respectively) or a range of characters with three dots (...) separating the lower and upper limits. A clause specifies the union of the characters given by the phrases when "=" is used as a separator; and the complement of that union in the set of all characters when "≠" is used.

GRAMMA was designed more for the convenience of the user than for that of a grammar-driven analyzer: phrases may have an arbitrarily complex structure and lexical symbols are specified implicitly in the **syn** section. In Appendix E we use MSL to specify syntactic mappings from GRAMMA to a simpler domain called GRAM. The Appendix starts with descriptions of GRAMMA and GRAM by simple grammars in the usual format.

Basically the mapping transforms a clause into a **prod** by attaching the defined name to each phrase, making a list of the symbols occurring in the phrase, and transforming the structure formed from ( ) *? into a list of successor sets which give the adjacency relation between symbols using their indices in the list. The mapping also forms **prods** in **lex** by extracting the implicit definitions from **syn**.

We keep our lexical and syntax analyzers separate; hence we obtain a complete parser from a parse-tree of a description in GRAMMA as shown in Table 13.

In the next two sections we examine "Lex" and "Syn".

TABLE 12: 'GRAMMA-Parser/GRAMMA'

```
syn
    Grammar  ::=  NLS   Syn Lex Aux          :'Syn Lex Aux'
    Syn      ::=  'syn' NLS SClause*         :'syn SC*'
    SClause  ::=  Name '::=' SPhraseL NLS    :'Name::=SPL'
    SPhraseL ::=  SPhrase (BAR SPhrase)*?    :'SP*'
    SPhrase  ::=  SWordL ':' Symb            :'SWL:Symb'
               |  SWordL                     :'SWL'
    SWordL   ::=  SWord*?                    :'SW*?'
    SWord    ::=  Name                       :'Name'
               |  Symb                       :'Symb'
               |  LB SWord*? RB              :'SW*?'
               |  SWord OPT                  :'SW OPT'
               |  SWord REP                  :'SW REP'
    Lex      ::=  'lex' NLS LClause*?        :'lex LC*?'
    LClause  ::=  SClause
               |  AClause
    Aux      ::=  'aux' NLS AClause*?        :'aux AC*?'
    AClause  ::=  Name '=' APhraseL NLS      :'Name=APL'
               |  Name '‡' APhraseL NLS      :'Name‡APL'
    APhraseL ::=  APhrase (BAR APhrase)*?    :'AP*'
    APhrase  ::=  Symb                       :'Symb'
               |  Symb '...' Symb            :'Symb...Symb'
    Name     ::=  SNAME                      :'SN '
               |  LNAME                      :'LN '
               |  LNAME '_'                  :'LV '
               |  'λ'
    Symb     ::=  SYMBOL                     :'SY '


lex
    NLS      ::=  NL* SPACE*?
    BAR      ::=  (NL* SPACE*?)? '|' NL*?
    LB       =    '('
    RB       =    ')'
    OPT      =    '?'
    REP      =    '**'
    SNAME    ::=  U A*?
    LNAME    ::=  W*
    SYMBOL   ::=  '*' C B*? '*'
               |  '*' '**' X '*'
               |  '*' '*'.

    λ        ::=  SPACE*
```

TABLE 12 (ctd.)

```
aux
    U          =  'A' ... 'Z'
    A          =  'A' ... 'Z'   |  'a' ... 'z'   |  '0' ... '9'
    W          =  'A' ... 'Z'   |  '0' ... '9'
    NL         =  '*n'
    B          ‡  '*n'   |  '*''
    C          ‡  '*n'   |  '**'   |  '*''
    X          ‡  '*n'   |  '*s'   |  '*4'
    SPACE      =  '*s'   |  '*4'
```

•

TABLE 13: 'GRAMMA-Semantics/MSL'

```
'GRAMMA-Semantics/MSL'

 LParserTree.

let ParserGRAM = (seg 'GRAMMA→GRAM')(LParserTree)

let <SynGRAM,FollGRAM, LexGRAM, AuxGRAM> = ParserGRAM

result (seg 'Lex')<LexGRAM, AuxGRAM> o
        (seg 'Syn')<SynGRAM,FollGRAM>
.
```

§4.3 Lexical Analysis

We take the commonly-held view that lexical languages should be recognizable by finite-state automata. To specify such a language, we consider iteration to be more natural than the recursion used in *regular languages* [30]: e.g. an integer *INT* would be specified as "a sequence of digits" by

INT ::= *D**

(where *D* = '0' ... '9'), rather than as "a digit, or a digit followed by an integer" by

INT ::= *D* | *D INT* .

It is clear that an iterative description can be transformed into a regular grammar; the possibility or otherwise of the inverse transformation need not concern us here, as the iterative system is certainly general enough to be useful.

Lexical grammars differ from syntactic grammars in that all clauses specify "goals" (a syntactic grammar has a special "root" clause). Often it is possible that several goals may be achieved starting from the beginning of the same sequence of characters, for example consider ":=" and ":" . This "ambiguity" is resolved by postulating that the largest possible symbol be recognized.

Appendix F gives the MSL segment "*Lex*", which describes a grammar-directed lexical analyzer. Basically the analyzer operates by keeping a set of states which indicate which goals conform to the characters scanned so far. When the state-set becomes empty, the previous state-set should contain only one "terminal" state, which indicates the goal achieved - otherwise the input is invalid or the lexical grammar is fundamentally ambiguous. The input to the lexical analyzer is in the form of a tuple of character-values - we assume the value of a character is the same as the value of the LAMB-string

formed from it - and the output is a tuple of symbols, which are in fact simple terminal nodes.

Doubtless the algorithm in Appendix F could be made more efficient, by generating decision tables from the grammar. As this would tend to obscure the simple nature of the algorithm, we do not do this here.

§4.4 Syntax Analysis

Our grammar-directed syntax analyzer is based on the SLR(1) recognizer algorithm, described in [29]. The main reason for choosing this algorithm is its relative simplicity, but also it seems that it provides a good compromise between generality and efficiency. In fact to get efficiency, state tables must be generated - we do not describe this here, but note that the interpreter of the tables used in parsing must be more complicated than the one given in [29], as nodes corresponding to phrases specified by the *-convention may have an arbitrary number of branches.

The algorithm we give in Appendix G was taken from [29] and extended to allow a non-sequential succession relation between the symbols in a phrase. This seems to be a natural way of incorporating the Brooker-*-? convention into SLR(1) grammars, and the mapping mentioned at the end of §4.2 effectively forms the succession relation from the original phrase.

We describe the operation of our syntax analyzer below, using the notation of [29] so that our extensions to the original algorithm can be clearly identified.

Let $\emptyset$ denote the empty set and Λ denote the null string.

If X and Y are sets of strings and xy denotes the concatenation of x and y, then $XY = \{xy \mid x \in X \text{ and } y \in Y\}$

If $X^0 = \{\Lambda\}$ then sets X^i are defined by

$$X^{i+1} = X^i X \text{ for } i \geq 0;$$

also X^* is defined by

$$X^* = \bigcup_{i=0}^{\infty} X^i$$

A context-free grammar (with an end marker and with successors)

(*) For example, the phrases

$$A ::= B\ C^*\ D$$
$$E ::= F?\ G\ H?$$
$$I ::= J\ K^*?$$

correspond to the productions:

$$A \rightarrow \{1\}\ B\ \{2\}\ C\ \{2,3\}\ D\ \{4\}$$
$$E \rightarrow \{1,2\}\ F\ \{2\}\ G\ \{3,4\}\ H\ \{4\}$$
$$I \rightarrow \{1\}\ J\ \{2,3\}\ K\ \{2,3\}.$$

is a quadruple $G = (V_N, V_T, P, W)$, where

V_N is a finite set of non-terminal symbols,

V_T is a finite set of terminal symbols,

$V_N \cap V_T = \emptyset$ and V denotes $V_N \cup V_T$,

P is a tuple of productions, which are elements of $V_N \times V^*$,

W is a tuple of successions, which are tuples of non-empty sets of integers called successors.

Let S be the number of productions. We denote the pth production and the pth succession simultaneously by

$$A_p \rightarrow U_{p0} X_{p1} U_{p1} \ldots X_{pn_p} U_{pn_p} \qquad (1 \leq p \leq s)$$

where A_p denotes an element of V_N, the X_{pi} denote elements of V and the U_{pi} denote sets of integers.(*) The U_{pi} must satisfy $j \in U_{pi}$ implies $1 \leq j \leq n_p + 1$. By convention the first production and succession are

$$A_1 \rightarrow \{1\} A_2 \{2\} \perp \{3\}$$

where $\perp \in V_T$ denotes an end marker such that $\perp \neq X_{p_i}$ for any $p \geq 2$.

T_p denotes the set $follow(A_p) \cap V_T$ $(1 \leq p \leq s)$, where *follow* is a function which we shall define later - basically *follow* yields those symbols which may follow an instance of a phrase which can be derived from the right-hand side of the pth production.

A stateset $\mathcal{S}_i$ is a set of elements $[p, j, k]$, where the occurrence of such an element in a stateset means informally that potentially k symbols of production p have been recognised, and the next symbol expected is $X_{p(j+1)}$ if $j < n_p$, or an element of T_p if $j = n_p$.

The initial stateset $\mathcal{S}_0$ is $\{[1,0,0]\}$. The algorithm involves maintaining a stack throughout the parse which is represented (to the left of the vertical bar) as

(1) $$\mathcal{S}_0 \mathcal{S}_1 \ldots \mathcal{S}_m \mid y \ \omega.$$

$y \in V_T$ denotes the next input symbol and ω denotes the (unexamined) remainder of the input string. Assume that the parse has reached the stage represented in (1) so that there are m elements in the stack. The algorithm proceeds as follows.

Step 1. Compute $\mathcal{S}_m'$ from $\mathcal{S}_m$ where $\mathcal{S}_m'$ is defined recursively as the smallest set satisfying

(2) $$\mathcal{S}_m' = \mathcal{S}_m \cup \{[q,0,0] \mid \exists [p,j,k] \in \mathcal{S}_m' . \exists l \in U_{p_j} . X_{p_l} = A_q\}$$

i.e. if $[p,j,k]$ is in the set, some successor X_{p_l} of X_{p_j} is potentially about to be recognized; if X_{p_l} is a non-terminal symbol defined by, for example, the qth production, then the first symbol of that production could be about to be recognized, so $[q,0,0]$ is added to the set to reflect this.

Step 2. Compute the following sets of terminal symbols.

(3) $$Z = \{a \in V_T \mid \exists [p,j,k] \in \mathcal{S}_m' . j < n_p, a = X_{p(j+1)}\}$$

(4) $$Z_{pk} = T_p \text{ if } [p,n_p,k] \in \mathcal{S}_m'$$
$$= \emptyset \text{ otherwise.}$$

Membership of Z_{pk} by the next input symbol y signals the recognition of a string generated by the pth production. Z is the set of terminal symbols which might legitimately be encountered next upon the input string and which indicate that the end of a string generated by some production has not been reached. (From (7) it will follow that when $y \in Z_{pk}$, the stateset $\mathcal{S}_{m-k}$ includes the element $[p,0,0]$ and, by (3), an element $[p',j',k']$ such that $X_{p'(j'+1)} = A_p$.)

A grammar is defined to be SLR(1) if in all parses of valid sentences of the language we have for all k, k' and $p \neq q$

$$Z \cap Z_{pk} = Z_{pk} \cap Z_{qk'} = \emptyset$$

Under this condition, an unambiguous choice of action can be made, depending upon the next character on the input string:

(1) If $y \in Z$, stack y so that the stack-input representation becomes

$$\mathcal{S}_0 \mathcal{S}_1 \cdots \mathcal{S}_m \, y \mid \omega$$

and rename the stack-input symbols producing

(5) $$\mathcal{S}_0 \mathcal{S}_1 \cdots \mathcal{S}_m X_{m+1} \mid y \, \omega'$$

i.e. y is named X_{m+1} and the first symbol of ω becomes the new y.

(2) If $y \in Z_{pk}$, on removing the top k elements from the stack and inserting A_p, the stack-input representation becomes

$$\mathcal{S}_0 \mathcal{S}_1 \cdots \mathcal{S}_{m-k} \, A_p \mid y \, \omega$$

which after renaming symbols is

(6) $$\mathcal{S}_0 \mathcal{S}_1 \cdots \mathcal{S}_m \, X_{m+1} \mid y \, \omega.$$

(3) If $y \notin Z$ and $y \notin Z_{pk}$ for any p,k then y is illegal, and the parse is halted in an error condition.

Step 3. Using the $\mathcal{S}_m$ defined by (5) or (6) compute $\mathcal{S}_m'$ as in Step 1 (or recall it) and replace X_{m+1} on the stack by $\mathcal{S}_{m+1}$ where

(7) $$\mathcal{S}_{m+1} = \{[p,j+1,k+1] \mid \exists [p,j,k] \in \mathcal{S}_m', \; j < n_p, \; X_{m+1} = X_{p(j+1)}\}$$

i.e. the symbol X_{m+1} left on the stack in Step 2 has just been recognized.

If $\mathcal{S}_{m+1} = \{[1,2,2]\}$, the parse is complete, A_2 having been recognized. Otherwise, we have reduced either m or ω, and the algorithm proceeds from Step 1.

It can be seen that the above description is a simple extension of that in [29] to allow non-sequential successors. The computation of $follow(A_p)$ is also similar as follows.

First non-terminal symbols generating the empty string are determined recursively using

$$A \overset{*}{\Rightarrow} \Lambda \text{ iff } A \to U_0 B_1 U_1 \cdots B_n U_n \overset{*}{\Rightarrow} \Lambda;$$

$U_0 B_1 U_1 \ldots B_n U_n \overset{*}{\Rightarrow} \Lambda$ iff $j_1, \ldots, j_k$ such that $j_1 \in U_0$, $j_k = n+1$ and for $1 \leq i < k$, $j_{i+1} \in U_{j_i}$ and $B_{j_i} \overset{*}{\Rightarrow} \Lambda$.

For a terminal symbol x, define $first(x) = \{x\}$.

For a non-terminal symbol A, take

$X \in first(A)$ iff $X = A$ or

$\exists Y.\ X \in first(Y),\ \exists (A \rightarrow U_0 B_1 U_1 \ldots B_k S_k Y \ldots),\ U_0 B_1 U_1 \ldots B_k S_k \overset{*}{\Rightarrow} \Lambda$.

Similarly, for a terminal symbol x, define $last(x) = \{x\}$.

For a non-terminal symbol A, take

$X \in last(A)$ iff $X = A$ or

$\exists Y.\ X \in last(Y),\ \exists (A \rightarrow \ldots Y U_{k-} B_k U_k \ldots B_n U_n),\ U_{k-1} B_k U_k \ldots B_n U_n \overset{*}{\Rightarrow} \Lambda$.

Finally,

$X \in follow(Y)$ iff $\exists (A \rightarrow \ldots Y_1 U_{j-1} B_j U_j \ldots B_k U_k X_1),\ Y \in last(Y_1),\ X \in first(X_1),\ U_{j-1} B_j U_j \ldots B_k U_k \overset{*}{\Rightarrow} \Lambda$.

Appendix G gives the MSL version of the above algorithm, representing sets as tuples and obtaining recursively-specified sets in a deterministic manner. As well as recognizing strings, the MSL version creates the corresponding nodes of a parse-tree, using labels specified in the grammar; it also makes a distinction between "constant" (V_C) and "variable" (V_V) terminal symbols: both are treated as members of V_T, but the latter are inserted as leaves into the parse-tree.

CHAPTER 5

An Example of Compiler-Generation

In this chapter we give an example of using the compiler-generator system to generate and use a compiler for a small programming language called TestL. §5.1 describes the syntax of TestL, using GRAMMA, §5.2 uses MSL to describe TestL semantics and §5.3 gives a simple implementation of the functions left undefined by the semantics, to enable programs to be evaluated. §5.4 gives some simple test-programs, their code and the results of evaluating them.

We do not propose TestL as a practical programming language - its features are just a suitable compromise between simplicity and convenience. TestL is small enough to make its formal description easy to understand yet powerful enough to express all the usual algorithms usually taken as examples in papers on semantics.

It must be admitted that because of time and space restrictions, the compiler-generator at present uses a grammar-driven parser written in BCPL - in fact a straightforward translation of the MSL version given in the Appendices. Everything else is done by a LAMB interpreter (itself coded in BCPL).

§5.1 Syntax of TestL

Table 14 gives a formal description of the syntax of TestL, using GRAMMA. TestL consists of a variety of constructs, taken from languages such as ALGOL, BCPL[24] and PASCAL [10] - our aim is a small but powerful language with a simple semantics, easily understood by the reader. Certainly TestL could be made more convenient by adding "syntactic sugaring", but we do not intend it to be used as a practical language.

The right-hand section of Table 14 can be interpreted as a descriptive grammar of the sort used in semantic equations, whereas the left-hand section is unambiguously prescriptive (and satisfies the SLR(1) condition). Note the use of the default conventions in the omission of certain label specifications: *Com*, *BoolA*, (*Bool*), *ExpA* and (*Exp*) are semantically irrelevant, and corresponding nodes will not occur in the parse-trees; the labels of nodes corresponding to the operators ∨, ∧, ... are strings formed from the operators themselves. The definition of *LAYOUT* uses the conventions of BCPL to specify the characters NEWLINE, SPACE and FOURSPACES. Note also that ')' '=' is used in the definition of **fun** $I(Il) = E$, to allow layout-characters between ')' and '=' (see §4.2).

TABLE 14: 'TestL-Parser/GRAMMA'

```
syn
    Com   ::= 'begin' ComL 'end'                    :'begin C end'
            | 'while' Bool 'do' Com                 :'while B do C'
            | Ide ':=' Exp                          :'I:=E'
            | 'write' Exp                           :'write E'
            | 'res' Exp                             :'res E'
    ComL  ::= Def ';' ComL                          :'D;C'
            | Com ';' ComL                          :'C;C'
            | Com
    Def   ::= 'var' Ide ':=' Exp                    :'var I:=E'
            | 'fun' Ide '(' IdeL ')' '=' Exp        :'fun I(I1)=E'
    Bool  ::= BoolA LogOp BoolA                     :'B Op B'
            | Not BoolA                             :'Op B'
            | BoolA
    BoolA ::= ExpA RelOp ExpA                       :'E Op E'
            | '(' Bool ')'
    Exp   ::= Bool '→' Exp ',' Exp                  :'B→E,E'
            | 'valof' Com                           :'valof C'
            | ExpA NumOp ExpA                       :'E Op E'
            | ExpA
    ExpA  ::= Ide '(' ExpL ')'                      :'I(E1)'
            | Ide                                   :'I'
            | NUM                                   :'N'
            | 'read'
            | '(' Exp ')'
    ExpL  ::= Exp ',' ExpL                          :'E,E1'
            | Exp                                   :'E'
    IdeL  ::= Ide (',' Ide)*?                       :'I*'
    Ide   ::= IDE                                   :'I'
    LogOp ::= '∨' | '∧'
    Not   ::= '~'
    RelOp ::= '=' | '<'
    NumOp ::= '+' | '-' | 'X'

lex
    IDE   ::= L*
    NUM   ::= D*
          ::= LAYOUT*

aux
    L       = 'a'...'z'
    D       = '0'...'9'
    LAYOUT  = '*n' | '*s' | '*4'
.
```

§5.2 Semantics of TestL

The semantics of TestL has been chosen not only for its simplicity but also for its diversity. Thus we use continuations for command values, but not for expressions or declarations; we do some basic type-checking; and we deal with lists formed both by recursion (*ExpL*) and iteration (*IdeL*) according to Table 14.

Table 15 gives the semantics of TestL using MSL. In fact this and the previous Table are given in "hardware representations" of our description languages - the more elegant "reference language" version may be imagined with the aid of Appendix Z, but we note in particular that doubled lower-case letters are used instead of Italic letters (with suffixed digits instead of subscripts), and doubled upper-case letters instead of Script letters (denoting semantic functions).

We hope that the semantics of TestL are self-explanatory; however we note the following fundamental points about TestL.

There are three (manifest) types: **locn**, **fn** and **num**. An identifier denotes a **num** only if it is a formal parameter of a function, whereas identifiers denoting **locn**s and **fn**s are declared using **var** *I* := *E* and **fun***I*(*Il*) = *E* respectively. The value of an expression *E* is of type **num**; if the denotation of an identifier in *E* is of type **locn**, then the current contents of the location is used in evaluating *E*. Only identifiers denoting **locn**s may be assigned to.

A definition of a function in TestL may be internally recursive, but a list of definitions is sequential. No test is made that the formal parameters of a function are distinct; the actual parameters are evaluated from left to right before entering the body of the function, and expressions are also evaluated from left to right.

Finally, the omission of a **res** *E* from a **valof** *C* will cause **err** to be returned as the result; whereas **res** *E* occurring "outside" all occurrences of **valof** *C* will produce a value in the wrong domain, which will be equivalent to **err** when used. Note that $\underline{\circ}$ and $\underline{\star}$ are "strict" on non-termination ($\perp$) and errors (**err**).

When the value of the segment below is applied to a parse-tree of a TestL program, the result is the semantic value of the program, which is a function of the values of the primitive functions *Locate*, *Update*, etc.

TABLE 15: 'TestL-Semantics/MSL'

! Reduced Syntax:

```
! (Com)   C  ::= begin C end  |  while E do C  |  I := E
!              |  write E  |  res E  |  D ; C  |  C ; C
! (Def)   D  ::= var I := E  |  fun I(I1) = E
! (Bool)  B  ::= B Op B  |  Op B  |  E Op E
! (Exp)   E  ::= B → E, E  |  valof C  |  E Op E
!              |  I(E1)  |  I  |  N
! (ExpL)  E1 ::= E, E1  |  E
! (IdeL)  I1 ::= I*
! (Ide)   I  ::= I
! (Op)    Op ::= ∨ | ∧ | ~ | = | ≤ | + | - | X
```

! Semantic Domains:

```
! b: Bool  = truth-values true, false
! c: C     = [S → S]
! d: D     = [[Locn + Fn + Num] X Type]
! e: E     = [Bool + Val]
! r: Env   = [I → D]
! f: Fn    = [Num* → S → [Num X S]]
!    I     = Num (identifiers used in Env)
! i: Input = unspecified
! a: Locn  = unspecified locations in S
! n: Num   = integers (≥ 0)
!    O     = null domain (contains err)
! o: Output= unspecified
! s: S     = unspecified stores
! t: Type  = strings 'locn', 'fn', 'num'
!    Val   = Num (storable values)
! z: Z     = universal domain
```

```
λtto.          ! : Com

λ<Locate,      ! : [Val → S → [Locn X S]]
  Update,      ! : [[Locn X Val] → S → S]
  Contents,    ! : [Locn → S → [Val X S]]
  Apply,       ! : [Fn → Num* → S → S]
  Read,        ! : [S → [Num XS]]
  Write,       ! : [Num → S → S]
  Start,       ! : [Input → S]
  Finish>.     ! : [S → Output]
```

TABLE 15 (ctd.)

```
def CC[tt](r)(c) =      ! : [Com → Env → C → C]

clauses tt

§ case 'begin C end' <cc>:

      CC[cc](r)(c)

  case 'while B do C' <bb,cc>:

      fix c1.
        EE[bb](r) * λb. b → CC[cc](r)(c1), c

  case 'I:=E' <ii,ee>:

      let <a,t> = r(II[ii]) in
      t‡'locn' → err,
      EE[ee](r) * λv. Update<a,v> o c

  case 'write E' <ee>:

      EE[ee](r) * λv. Write(v) o c

  case 'res E' <ee>:

      EE[ee](r)

  case 'D;C' <dd,cc>:

      DD[dd](r) * λd. CC[cc](Lay<r,<d>,<IId[dd]>>) || c

  case 'C;C' <cc1,cc2>:

      CC[cc1](r) || CC[cc2](r) || c

§

and DD[tt](r) =         ! : [Def → Env → S → [D X S]]

clauses tt

§ case 'var I:=E' <ii,ee>:

      EE[ee](r) * λv. Locate(v)
```

TABLE 15 (ctd.)

```
  case 'fun I(I1)=E' <ii,iil,ee>:

       pair (
         fix f. λp.
           let r1 = Lay<r,p,III[iil]> in
           let r2 = Lay<r1,<f>,<IId[tt]>> in
           EE[ee](r2) )

§

and IId[tt] =                    ! : [Def → [I X Type]]

clauses tt

§ case 'var I:=E' <ii,ee>:

       <II[ii], 'locn'>

  case 'fun I(I1)=E' <ii,iil,ee>:

       <II[ii], 'fn'>

§

and III[tt] =                    ! : [IdeL → I*]

clauses tt

§ case 'I*':

       mapt <(λtt1.<II[tt1],'num'>), tt>

§

and II[tt] =                     ! : [Ide → I]

clauses tt

§ case 'I ' <n>:

       n

§
```

TABLE 15 (ctd.)

```
and EE[tt](r) =          ! : [[Bool + Exp] → Env → S → [E X S]]

clauses tt

§ case 'B Op B' case 'E Op E' <ee1,op,ee2>:

      EE[ee1](r) * λe1. EE[ee2](r) * λe2.
        pair ( DiOpVal[op]<e1,e2> )


  case 'Op B' <op,ee>:

      EE[ee](r) * λe.
        pair ( MonOpVal[op](e) )


  case 'B→E,E' <ee1,ee2,ee3>:

      EE[ee1](r) * λb.
        b → EE[ee2](r), EE[ee3](r)


  case 'valof C' <cc>:

      CC[cc](r) || pair(err)


  case 'I(El)' <ii,eel>:

      let <f,t> = r(II[ii]) in
      t‡'fn' → err,
      PP[eel](r)<> * λp. Apply(f)(p)


  case 'I' <ii>:

      let <z,t> = r(II[ii]) in
      t='fn' → err,
      t='locn' → Contents(z),
      t='num' → pair(z),
      err


  case 'N ' <n>:

      pair ( NumVal[n] )


  case 'read' <>:

      Read

§
```

TABLE 15 (ctd.)

```
and PP[tt](r)(p) =      ! : [ExpL → Env → Num* → S → [Num* X S]]
clauses tt
§ case 'E,El' <ee,eel>:
      EE[ee](r) * λe. PP[eel](r)(p aug e)
  case 'E' <ee>:
      EE[ee](r) * λe. pair ( p aug e )
§

and DiOpVal[tt]<e1,e2> =  ! : [Op → [E X E] → E]
clauses tt
§ case '+':  e1 + e2
  case '-':  e1 - e2
  case 'X':  e1 X e2
  case '=':  e1 = e2
  case '≤':  e1 ≤ e2
  case 'v':  e1 v e2
  case '^':  e1 ^ e2
§

and MonOpVal[tt](e) =  ! : [Op → E → E ]
clauses tt
§ case '~':  ~ e
§

and NumVal[n] =         ! : [Num → Num]
      let <z> = chars '0' in
      let s = chars n in
      mapn < (λ<m,i>. ((mX10)+(s↓i))-z), 0, 1, dim s >
```

TABLE 15 (ctd.)

```
and Lay<r,p,iil> =

      mapn < (λ<r1,n>. r1[<p↓n,iil↓n↓2> / iil↓n↓1]), r, 1, dim iil >


result Start o CC[ttO](err) || Finish     ! : [Input → Output]
•
```

§5.3 The Implementation of TestL

Table 16 defines the primitive functions which were left undefined in the semantics of TestL. In fact we believe that a semantics should ideally include a set of predicates giving the essential properties of the primitive functions (such as the relationship between *Update* and *Contents*), which could then be proved to hold for a specific implementation - but we must wait for a suitable notation for predicates involving continuations. Many different abstract implementations of TestL could be given - we consider ours to be particularly simple. However, the "memory" part of the state is a tuple, rather than a function, for purely practical reasons concerning efficiency (spacewise).

When the value of the segment below is applied to the semantic value of a program in TestL, the result is a function from input to output.

TABLE 16: 'TestL-Implementation/MSL'

```
! Implementation Domains:

! a: Locn   = Num
! s: S      = [Memory X Area X Input X Output]
! m: Memory= [[Memory X Locn X Val] + O]
!    Area   = Num
! i: Input  = [[Num X Input] + O]
! o: Output= Num*

! (See 'TestL-Semantics/MSL' for functionalities.)

λ Prog.

let Locate(v)<m,n,i,o> = <n+1, <<m,n+1,v>,n+1,i,o>>

and Update<a,v><m,n,i,o> = <<m,a,v>, n, i, o>

and Contents(a)(s) =
      def Find(m) =
          isatom m → err,
          (m↓2)=a → m↓3,
          Find(m↓1)
      in
          <Find(s↓1), s>

and Apply(f)(p) = f(p)

and Read<m,n,i,o> = <i↓1, <m,n,i↓2,o>>

and Write(z)<m,n,i,o> = <m,n,i,(o aug z)>

and Start(i) = <err, 0, i, <>>

and Finish<m,n,i,o> = o

result Prog<Locate,Update,Contents,Apply,Read,Write,Start,Finish>
•
```

§5.4 TestL Programs

The Tables below give test programs written in TestL, with both their LAMB-code as produced by the generated compiler and the output from evaluating the code on the specified inputs. The examples are small, but we hope that they are sufficiently complex to suggest that larger examples would also give correct answers.

From the LAMB-code of the test programs it is easy to see why we put the operators $\underline{\circ}$, $\underline{*}$ and **pair** in LAMB, rather than defining them in terms of other LAMB-expressions. The code can be read very easily from left to right, by interpreting $f \mathbin{\underline{\circ}} g$ "operationally" as "do f, then do g", and $f \mathbin{\underline{*}} \lambda x.g$ as "do f, then do g with x denoting the value produced by f". (Jones [3] also argues the case for having both an operational and a mathematical interpretation of the same symbols.) Without the use of $\underline{*}$, the code would in fact grow exponentially with the size of the program, as $f \mathbin{\underline{*}} g$ is (roughly) equivalent to $\lambda\sigma.\ g(f(\sigma)\downarrow 1)(f(\sigma)\downarrow 2)$.

To give the reader some idea of the performance of the LAMB-interpreter: the compilations of the test programs take about three minutes, and their executions about two minutes each. These times would be reduced by a factor of about 20 if the BCPL form of the reduction algorithm were to be compiled into Modular One machine code, rather than into the virtual machine code used at present - in comparison, our BCPL compiler takes up to 10 minutes to compile large programs.

TABLE 17: Factorial by Looping

```
'Prog/TestL'

begin var n := read;
      var s := 1;
      while 1 ≤ n do
      begin s := s X n;
            n := n - 1
      end;
      write s
end
•
```

```
'Prog'

λ<Locate,Update,Contents,Apply,Read,Write,Start,
 Finish>. Start o
 (Read *
  λv16.
   Locate(v16)) *
  λd4. Locate(1) *
   λd5. fix c15. (Contents(d4) *
    λe210. pair (1 ≤ e210)) *
    λb5.
    b5 →
     (Contents(d5) *
      λe111. Contents(d4) *
       λe211. pair (e111 X e211)) *
      λv17. Update(<d5, v17>) o
       (Contents(d4) *
        λe112. pair (e112 - 1)) *
        λv18. Update(<d4, v18>) o c15,
     Contents(d5) *
      λv19. Write(v19) o Finish
•
```

```
'Input'

<3,
 <>>
•
```

```
'Output'

<6>
•
```

TABLE 18: Factorial by Recursion

```
'Prog/TestL'

begin var n := read;
      fun f(n) =
            n ≤ 1 → 1,
            n X f(n-1);
      write f(n)
end
•


'Prog'

λ<Locate,Update,Contents,Apply,Read,Write,Start,
 Finish>. Start o
 (Read *
  λv5.
   Locate(v5)) *
  λd4. (pair (fix f3. λp6.
   (p6 ↓ 1) ≤ 1 →
    pair 1,
    Apply(f3)(
      <(p6 ↓ 1) - 1>) *
    λe23. pair ((p6 ↓ 1) X e23))) *
  λd5. ((Contents(d4) *
   λe4. pair (<e4>)) *
   λpZ.
    Apply(d5)(pZ)) *
   λv6. Write(v6) o Finish
•


'Input'

<6,
 <>>
•


'Output'

<720>
•
```

TABLE 19: The Towers of Hanoi

```
'Prog/TestL'

begin var n := read;
      fun h(n,a,b,c) =
            n≤0 → 0,
            valof
            begin var t := 0;
                  t := h(n-1,a,c,b);
                  write (10Xa) + b;
                  t := t + h(n-1,c,b,a);
                  res t + 1
            end;
      write h(n,1,2,3)
end
•
```

TABLE 19 (ctd.)

```
'Prog'

λ<Locate,Update,Contents,Apply,Read,Write,Start,
 Finish>. Start o
 (Read *
  λv17.
   Locate(v17)) *
  λd7. (pair (fix f3. λp6.
   (p6 ↓ 1) ≤ 0 →
    pair 0,
    Locate(0) *
     λd8. Apply(f3)(
       <(p6 ↓ 1) - 1,
        p6 ↓ 2,
        p6 ↓ 4,
        p6 ↓ 3>) *
      λv18. Update(<d8, v18>) o
       Write(
         (10 X (p6 ↓ 2)) + (p6 ↓ 3)) o
        (Contents(d8) *
         λe115. Apply(f3)(
           <(p6 ↓ 1) - 1,
            p6 ↓ 4,
            p6 ↓ 3,
            p6 ↓ 2>) *
          λe28. pair (e115 + e28)) *
         λv19. Update(<d8, v19>) o
          Contents(d8) *
           λe116. pair (e116 + 1))) *
   λd9. ((Contents(d7) *
    λe4. pair (<e4, 1, 2, 3>)) *
    λp7.
     Apply(d9)(p7)) *
    λv20. Write(v20) o Finish
•

'Input'

<3,
 <>>
•

'Output'

<12, 13, 23, 12, 31, 32, 12, 7>
•
```

CONCLUSION

To conclude this dissertation, we consider the possible future development of the MSL notation and of the compiler-generator system.

We feel that MSL, as presented here, is quite successful in keeping close to the usual notation of semantic equations. We hope that it will be regarded as an alternative format of the latter notation, and will be used for describing mathematical semantics whenever precision and machine-checkability are deemed to be more important than extreme compactness and mathematical informality - for instance, when *defining* a programming language, as distinct from describing or simply discussing it. Certainly the novice reader of mathematical semantics should find his task easier with MSL, especially if his background is in computing rather than in mathematics: for not only does MSL look like a programming language (e.g. PAL [19]) - and, incidentally, can be given a simple "operational" interpretation - but also it has a concrete syntax and a standard semantics for reference purposes (in contrast to the notation of semantic equations).

Although MSL is, we hope, useful in its present form, there are a few major alterations which we should like to make to it as soon as possible. (The syntax and semantics of these alterations would be defined in GRAMMA and in the original MSL, respectively.) They are as follows:

(i) MSL would be extended to include the definitions of semantic

domains, and the specifications of the types of the semantic functions. This would enable the compiler-generator system to make a number of checks on the code of the generated compiler. Also it would make it possible to add suitable operators for injection and projection between sum domains and their summands - at the moment the user has to define functions to implement injections and projections in terms of tuples.

(ii) We would extend MSL to include the description of the syntactic domains by an "ideal" grammar. Then it would be feasible to revert to the original notation of "meta-variables" , and even to incorporate the extended version which we found useful in Chapter 3 to describe reduction rules and algorithms.

(iii) The mathematical "..." notation would be added to MSL in a restricted form, as in e.g.:

```
def E[[ Exp]] ρ = cases
§
  case [[ ε1,...,εm]] :    mapk ⟨ 1, m, (λi. E[[εi]]ρ) ⟩
§
```

It would be difficult to introduce a precise "..." convention for use in the right-hand sides of cases.

(iv) More standard iterating and other functions would be provided for the user.

However, care must be taken not to make MSL too large and complex. We believe this to be especially important if we are to make mathematical semantics accessible to programmers and compiler-writers - who, after all, are the *raison d'être* of programming languages - instead of keeping it the preserve of workers in the theory of computation. Recent developments [31] show that VDL is a strong competitor to semantic equations for giving formal definitions of programming languages, and its success seems

to be due at least partly to the ease with which one can read the descriptions in an "operational" (or "imperative") way. (Perhaps it is also partly due to a laudable restraint in the use of uncommon symbols!) We venture to hope that the best features of semantic equations and VDL can be combined in an extended version of MSL.

Turning to the compiler-generator system, we see that much practical work needs to be done before the system can be used easily by anyone other than the author. At present an error in the user's input causes the system to stop and report (sometimes) the nature of the error; it is then up to the user to locate the error by tracing the operation of the system and/or by examining a core-dump. Clearly a sophisticated diagnostics facility, incorporating error recovery, must be provided for the user.

Development work of less impact on the user would be concerned with improving the efficiency of the system. This could proceed along two lines independently: (i) improving the algorithms used in the system, and (ii) optimising "the machine". As regards (i), the most obvious starting-point is the syntax analysis algorithm. Recall that the present one is a straight-forward adaptation of the grammar-driven recognition algorithm in [29]. As is remarked in [29], a substantial increase in the speed of parsing can be obtained by changing the recognition algorithm to produce a table, and then interpreting the table with the aid of a stack. This is true also for our lexical analysis algorithm. Concerning (ii), we do not see at present how the fundamental LAMB-interpreter (using "call-by-need") could be improved upon with respect to the number of reductions performed, without introducing expensive testing and copying operations. However the BCPL program which implements this algorithm spends a large proportion of its time doing "garbage

collection": this could be eliminated by providing a better strategy, based either on reference counts or on periodic marking garbage collection. Ultimately it is hoped to micro-program the LAMB-interpreter. If a fast method for finding the denotations of bound variables in environments can be developed, we can see no reason why LAMB could not be acceptably efficient in comparison to the lower-level codes in every-day use. An alternative approach, already being investigated by Clifford Hones at Oxford, would be to compile LAMB into a linear stack-based code before interpreting it.

Finally we consider what is perhaps the most exciting possibility: the formulation of a proof of the correctness of the compiler-generator system. Because of the simplicity of the way in which the compiler-generator operates (by forming applicative combinations of LAMB-expressions and reducing them) and the use of only one code (LAMB) in the system, there are just four points in need of formal proof: (i) a parser generated by the system from a grammar for a language L should transfrom all legal strings of L into the corresponding parse-trees; (ii) the LAMB reduction algorithm (Table 10) should preserve the denotations of LAMB-terms; (iii) the actual LAMB-interpreter used in the system should be a correct implementation of the algorithm in (ii); and (iv) the semantic description of MSL in MSL (Appendix D) should be consistent with the semantics of MSL in LAMB (Appendix C). The difficulty or otherwise of these proofs remains to be seen, although (iv) could presumably be done automatically by the LAMB-interpreter proven in (ii) and (iii).

We hope that from our work there will eventually be developed a convenient, efficient and correct compiler-generator system of

complete generality, using as its semantic description language a variant of MSL suited to both programmers and theorists. Even if this hope is not realized fully, we believe that the system will still be of sufficient use and interest to justify its existence.

APPENDIX A: LAMB Grammar

```
syn
    Exp  ::=  'λ' Bvs '.' Exp                    :'λB.E'
           |  'val' Bvs '.' Exp                  :'val B.E'
           |  'fix' Bvs '.' Exp                  :'fix B.E'
           |  ExpA '→' Exp ',' Exp               :'E→E,E'
           |  ExpA
    ExpA ::=  ExpB OpA ExpB                      :'E Op E'
           |  OpB ExpB                           :'Op E'
           |  ExpB
    ExpB ::=  ExpB ExpC                          :'E(E)'
           |  ExpC
    ExpC ::=  '<' Exp (',' Exp)*? '>'            :'<E*?>'
           |  '<' '>'                            :'<E*?>'
           |  IDE                                :'I '
           |  NUM                                :'N '
           |  STR                                :'S '
           |  'true'  |  'false'  |  'err'
           |  '(' Exp ')'
    Bvs  ::=  IDE                                :'I '
           |  '<' Bvs (',' Bvs)*? '>'            :'<B*?>'
           |  '<' '>'                            :'<B*?>'
    OpA  ::=  'o'  |  '*'  |  '||'
           |  '+'  |  '-'  |  'X'
           |  '='  |  '<'  |  'v'
           |  '^'  |  'cat'  |  '↓'
    OpB  ::=  'pair'  |  '~'  |  'dim'
           |  'isatom'  |  'imp'

lex
    IDE  ::=  L LD*? S?
           |  U ULD*? S?
    NUM  ::=  D*
    STR  ::=  '*' C B*? '*'
           |  '*' '**' X '*'
           |  '*' '*'
    λ    ::=  '!' CH*? '*n'
           |  LAYOUT*
```

APPENDIX A (ctd.)

aux

L	=	'a' ... 'z'
LD	=	'a' ... 'z' | '0' ... '9'
S	=	' '
U	=	'A' ... 'Z'
ULD	=	'A' ... 'Z' | 'a' ... 'z' | '0' ... '9'
D	=	'0' ... '9'
C	‡	'*n' | '*' | '**'
B	‡	'*n' | '*'
X	=	'*' | '**' | 'n' | 's' | '4'
CH	‡	'*n'
LAYOUT	=	'*n' | '*s' | '*4'

APPENDIX B: MSL Grammar

```
syn
    Seg   ::= 'λ' Pars '.' Seg                    :'λP.E'
            | 'def' DefL Seg                      :'def D in E'
            | 'let' DefL Seg                      :'let D in E'
            | 'result' Exp
    DefL  ::= Def ('and' Def)*?                   :'D*'
    Def   ::= IDE Pars '=' Exp                    :'I P=E'
            | Bvs '=' Exp                         :'B=E'
    Exp   ::= 'def' Def 'in' Exp                  :'def D in E'
            | 'let' Def 'in' Exp                  :'let D in E'
            | 'λ' Pars '.' Exp                    :'λP.E'
            | 'fix' Bvs '.' Exp                   :'fix B.E'
            | ExpA '→' Exp ',' Exp                :'E→E,E'
            | 'clauses' ExpA '§' CaseL '§'        :'clauses E§C§'
            | ExpA OpA Exp                        :'E Op E'
            | ExpA
    ExpA  ::= ExpB OpB ExpB                       :'E Op E'
            | OpC ExpB                            :'Op E'
            | OpD ExpD ExpD                       :'Op E E'
            | ExpB
    ExpB  ::= ExpB OpE ExpC                       :'E Op E'
            | ExpC
    ExpC  ::= ExpC ExpD                           :'E(E)'
            | ExpD
    ExpD  ::= ExpD '[' Exp '/' Exp ']'            :'E[E/E]'
            | '<' Exp (',' Exp)*? '>'             :'<E*?>'
            | '<' '>'                             :'<E*?>'
            | IDE                                 :'I '
            | NUM                                 :'N '
            | STR                                 :'S '
            | 'true'
            | 'false'
            | 'err'
            | '(' Exp ')'
            | '[' ExpA ']'
    Pars  ::= Bvs Pars                            :'B P'
            | Bvs                                 :'B'
    Bvs   ::= IDE                                 :'I '
            | '<' Bvs (',' Bvs)*? '>'             :'<B*?>'
            | '<' '>'                             :'<B*?>'
            | '(' IDE ')'                         :'I '
            | '[' IDE ']'                         :'I '
    CaseL ::= Case CaseL                          :'C C'
            | Case
    Case  ::= 'case' StrL Bvs ':' Exp             :'case S B:E'
            | 'case' StrL ':' Exp                 :'case S:E'
            | 'default' ':' Exp                   :'default:E'
    StrL  ::= Str 'case' StrL                     :'S case S'
            | Str
```

APPENDIX B (ctd.)

```
Str  ::=  STR                                          :'S'
OpA  ::=  'o'  |  '*'  |  '||'
OpB  ::=  '+'  |  '-'  |  'X'
      |   '='  |  '<'  |  '‡'
      |   'v'  |  '^'
      |   'aug'  |  'cat'  |  'pre'
OpC  ::=  'pair'  |  '~'  |  'isatom'  |  'dim'
      |   'seg'  |  'chars'  |  'string'
      |   'mapn'  |  'mapt'  |  'label'  |  'spread'
OpD  ::=  'node'
OpE  ::=  '↓'
```

lex

```
IDE  ::=  L LD*? S?
      |   U ULD*? S?
NUM  ::=  D*
STR  ::=  '*' C B*? '*'
      |   '*' '**' X '*'
      |   '*' '*'
λ    ::=  ':' CH*? '*n'
      |   LAYOUT*
```

aux

```
L       =  'a' ... 'z'
LD      =  'a' ... 'z'  |  '0' ... '9'
S       =  ' '
U       =  'A' ... 'Z'
ULD     =  'A' ... 'Z'  |  'a' ... 'z'  |  '0' ... '9'
D       =  '0' ... '9'
C       ‡  '*n'  |  '*'  |  '**'
B       ‡  '*n'  |  '*'
X       =  '*'  |  '**'  |  'n'  |  's'  |  '4'
CH      ‡  '*n'
LAYOUT  =  '*n'  |  '*s'  |  '*4'
```

APPENDIX C: MSL Semantics in LAMB

```
'MSL-Semantics'

λActivate.
 (λ<E,P,B,C,S,O2,O1,
   MapN,MapT,App,Abs,Fix,Con,Err,Select,
   Dim,BvTup,Eq,Tup,Or,Not,Cat,Imp,
   Minus,LessEq,Plus,InOp,PreOp,Id,Bv,Num>. E o Activate)(
   fix <E,P,B,C,S,O2,O1,
    MapN,MapT,App,Abs,Fix,Con,Err,Select,
    Dim,BvTup,Eq,Tup,Or,Not,Cat,Imp,
    Minus,LessEq,Plus,InOp,PreOp,Id,Bv,Num>.
    <λt.
      (λv1.
       v1 = 'λP.E' →
        (λ<p,e>.
          P(p)(
            E(e)))(t),

       v1 = 'def D in E' →
        (λ<d,e>.
          (λb.
           App(
            <Abs(
               <b,
                E(e)>),
             Fix(
               <b,
                E(d)>)>))(
           B(d)))(t),

      v1 = 'let D in E' →
       (λ<d,e>.
         (λb.
           App(
             <Abs(
                <b,
                 E(e)>),
              E(d)>))(
           B(d)))(t),

       v1 = 'fix B.E' →
        (λ<b,e>.
          Fix(
            <b,
             E(e)>))(t),
```

APPENDIX C (ctd.)

```
v1 = 'E→E,E' →
 (λ<e1,e2,e3>.
   Con(
     <E(e1),
      E(e2),
      E(e3)>))(t),

v1 = 'clauses E§C§' →
 (λ<e,c>.
   (λee.
     App(
       <Abs(
          <Bv(1),
           C(c)(
               Bv(1))(ee)(Err)>),
        Select(
          <ee,
           Dim(<ee>)>)>))(
    E(e)))(t),

v1 = 'E Op E' →
 (λ<e1,op,e2>.
   O2(op)(
     <E(e1),
      E(e2)>))(t),

v1 = 'Op E' →
 (λ<op,e>.
   O1(op)(
     <E(e)>))(t),

v1 = 'Op E E' →
 (λ<op,e1,e2>.
   O2(op)(
     <E(e1),
      E(e2)>))(t),

v1 = 'E(E)' →
 (λ<e1,e2>.
   App(
     <E(e1),
      E(e2)>))(t),
```

APPENDIX C (ctd.)

```
v1 = 'E[E/E]' →
 (λ<e1,e2,e3>.
   App(
     <Abs(
        <BvTup(
           <Bv(1),
            Bv(2),
            Bv(3)>),
         Abs(
           <Bv(4),
            Con(
              <Eq(
                 <Bv(4),
                  Bv(3)>),
               Bv(2),
               App(
                 <Bv(1),
                  Bv(4)>)>)>)>),
      Tup(
        <E(e1),
         E(e2),
         E(e3)>)>))(t),

v1 = '<E*?>' →
Tup(
  (λ<v6,v7>.
    (fix v5. λ<v1,v2,v3,v4>.
     v3 ≤ v4 →
      v5(
        <v1,
         v1(<v2, v3>),
         v3 + 1, v4>),
      v2)(
      <λ<v8,v9>. v8 cat (<v6(
           v7 ↓ v9)>),
       <>, 1,
       (dim v7) - 1>))(<E, t>)),

(v1 = 'I ') v
 (v1 = 'N ') v
  (v1 = 'S ') v
   (v1 = 'true') v
    (v1 = 'false') v
     v1 = 'err' → t,
```

APPENDIX C (ctd.)

```
v1 = 'D*' →
 Tup(
   (λ<v6,v7>.
     (fix v5. λ<v1,v2,v3,v4>.
      v3 < v4 →
       v5(
         <v1,
          v1(<v2, v3>),
          v3 + 1, v4>),
       v2)(
       <λ<v8,v9>. v8 cat (<v6(
            v7 ↓ v9)>),
        <>, 1,
        (dim v7) - 1>))(<E, t>)),

 v1 = 'I  P=E' →
  (λ<i,p,e>.
    P(p)(
      E(e)))(t),

 v1 = 'B=E' →
  (λ<b,e>.
    E(e))(t),
   err)(
   t ↓ (dim t)),

λt. λff.
 (λv1.
  v1 = 'B P' →
   (λ<b,p>.
     Abs(
       <b,
        P(p)(ff)>))(t),

   v1 = 'B' →
    (λ<b>.
      Abs(<b, ff>))(t),
    err)(
   t ↓ (dim t)),

λt.
 (λv1.
  v1 = 'D*' →
   BvTup(
     (λ<v6,v7>.
       ( fix v5. λ<v1,v2,v3,v4>.
        v3 < v4 →
         v5(
           <v1,
            v1(<v2, v3>),
            v3 + 1, v4>),
         v2)(
         <λ<v8,v9>. v8 cat (<v6(
              v7 ↓ v9)>),
          <>, 1,
          (dim v7) - 1>))(<B, t>)),
```

APPENDIX C (ctd.)

```
        v1 = 'I  P=E' →
         (λ<i,p,e>.
           Id(i))(t),

         v1 = 'B=E' →
          (λ<b,e>. b)(t),
          err)(
        t ↓ (dim t)),

    λt. λff. λgg. λhh.
     (λv1.
      v1 = 'C C' →
       (λ<c1,c2>.
         C(c1)(ff)(gg)(
           C(c2)(ff)(gg)(hh)))(t),

       v1 = 'case S B:E' →
        (λ<s,b,e>.
          Con(
            <S(s)(ff),
             App(
               <Abs(
                  <b,
                   E(e)>), gg>), hh>))(t),

        v1 = 'case S:E' →
         (λ<s,e>.
           Con(
             <S(s)(ff),
              E(e), hh>))(t),

         v1 = 'default:E' →
          (λ<e>.
            E(e))(t),
           err)(
       t ↓ (dim t)),

    λt. λff.
     (λv1.
      v1 = 'S case S' →
       (λ<s1,s2>.
         Or(
           <S(s1)(ff),
            S(s2)(ff)>))(t),

       v1 = 'S ' →
        (λ<i>.
          Eq(<ff, t>))(t),
         err)(
       t ↓ (dim t)),
```

APPENDIX C (ctd.)

```
λt. λ<ff,gg>.
 (λv1.
  (v1 = 'o') v
   (v1 = '*') v
    (v1 = '||') v
     (v1 = '+') v
      (v1 = '-') v
       (v1 = 'X') v
        (v1 = '=') v
         (v1 = '≤') v
          (v1 = 'v') v
           (v1 = '^') v
            (v1 = 'cat') v
             v1 = '↓' →
    InOp(
      <ff,
       t ↓ (dim t), gg>),

    v1 = '‡' →
     Not(
       <Eq(<ff, gg>)>),

     v1 = 'aug' →
      Cat(
        <ff,
         Tup(<gg>)>),

      v1 = 'pre' →
       Cat(
         <Tup(<ff>), gg>),

       v1 = 'node' →
        Cat(
          <gg,
           Tup(<ff>)>),
        err)(
   t ↓ (dim t)),

λt. λ<ff>.
 (λv1.
  (v1 = 'pair') v
   (v1 = '~') v
    (v1 = 'isatom') v
     v1 = 'dim' →
    PreOp(
      <t ↓ (dim t), ff>),

    v1 = 'seg' →
     App(
       <Imp(<'SegFn'>), ff>),

     v1 = 'chars' →
      App(
        <Imp(<'CharsFn'>), ff>),

      v1 = 'string' →
```

APPENDIX C (ctd.)

```
            App(
              <Imp(<'StringFn'>), ff>),

            v1 = 'mapn' →
             App(<MapN, ff>),

             v1 = 'mapt' →
              App(
                <MapT(MapN), ff>),

              v1 = 'label' →
               Select(
                 <ff,
                  Dim(<ff>)>),

               v1 = 'spread' →
                Minus(
                  <Dim(<ff>),
                   Num(1)>),
                err)(
       t ↓ (dim t)),

    Fix(
      <Bv(5),
    Abs(
      <BvTup(
         <Bv(1),
          Bv(2),
          Bv(3),
          Bv(4)>),
       Con(
         <LessEq(
            <Bv(3),
             Bv(4)>),
          App(
            <Bv(5),
             Tup(
               <Bv(1),
                App(
                  <Bv(1),
                   Tup(
                     <Bv(2),
                      Bv(3)>)>),
                Plus(
                  <Bv(3),
                   Num(1)>),
                Bv(4)>)>),
          Bv(2)>)>)>),
```

APPENDIX C (ctd.)

```
λmm.
 Abs(
   <BvTup(
      <Bv(6),
       Bv(7)>),
    App(
      <mm,
       Tup(
         <Abs(
            <BvTup(
               <Bv(8),
                Bv(9)>),
             Cat(
               <Bv(8),
                Tup(
                  <App(
                     <Bv(6),
                      Select(
                        <Bv(7),
                         Bv(9)>)>)>)>)>),
            Tup(<>),
            Num(1),
            Minus(
              <Dim(
                 <Bv(7)>),
               Num(1)>)>)>)>),

λ<ff,gg>. (<ff, gg>) cat (<'E(E)'>),

λ<bb,ff>. (<bb, ff>) cat (<'λB.E'>),

λ<bb,ff>. (<bb, ff>) cat (<'fix B.E'>),

λ<ff,gg,hh>. (<ff, gg, hh>) cat (<'E→E,E'>),
(<>) cat (<'err'>),

λ<ff,gg>.
 InOp(<ff, '↓', gg>),

λ<ff>.
 PreOp(<'dim', ff>),

λtt. tt cat (<'<B*?>'>),

λ<ff,gg>.
 InOp(<ff, '=', gg>),

λtt. tt cat (<'<E*?>'>),

λ<ff,gg>.
 InOp(<ff, 'v', gg>),

λ<ff>.
 PreOp(<'~', ff>),

λ<ff,gg>.
```

APPENDIX C (ctd.)

```
  InOp(<ff, 'cat', gg>),

λ<ff>.
  PreOp(<'imp', ff>),

λ<ff,gg>.
  InOp(<ff, '-', gg>),

λ<ff,gg>.
  InOp(<ff, '<', gg>),

λ<ff,gg>.
  InOp(<ff, '+', gg>),

λ<ff,ll,gg>. (<ff,
  (<>) cat (<ll>), gg>) cat (<'E Op E'>),

λ<ll,ff>. (<(<>) cat (<ll>), ff>) cat (<'Op E'>),

λi. (<i>) cat (<'I '>),

λi. (<'v', i>) cat (<'I '>),

λi. (<i>) cat (<'N '>)>)
```

APPENDIX D: MSL Semantics in MSL

! Reduced Syntax for MSL: (c.f. Appendix B)

```
! (Exp)      E  ::= λP.E  |  def D in E  |  let D in E
!                |  fix B.E  |  E→E,E  |  clauses E§C§
!                |  E Op E  |  Op E  |  Op E E
!                |  E(E)  |  E[E/E]  |  <E*?>
!                |  I  |  N  |  S
!                |  true  |  false  |  err

! (Pars)     P  ::= B P  |  B

! (Bvs)      B  ::= I  |  <B*?>

! (Defs)     D  ::= D*  |  I P=E  |  B=E

! (Cases)    C  ::= C C  |  case S B:E  |  case S:E  |  default:E

! (Strs)     S  ::= S case S  |  S

! (InOp)     Op ::= o  |  *  |  ||  |  +  |  -  |  X
!                |  =  |  ≤  |  ‡  |  v  |  ^
!                |  aug  |  cat  |  pre  |  ↓

! (PreOp1)   Op ::= pair  |  ~  |  isatom  |  dim
!                |  seg  |  chars  |  string
!                |  mapn  |  mapt  |  label  |  spread

! (PreOp2)   Op ::= node
```

! Reduced Syntax for LAMB:

```
! (ExpB)     E  ::= λB.E  |  val B.E  |  fix B.E
!                |  E→E,E  |  E Op E  |  Op E
!                |  E(E)  |  <E*?>
!                |  I  |  N  |  S
!                |  true  |  false  |  err

! (Bvs)      B  ::= I  |  <B*?>

! (InOpB)    Op ::= o  |  *  |  ||  |  +  |  -  |  X
!                |  =  |  ≤  |  v  |  ^  |  cat  |  ↓

! (PreOpB)   Op ::= pair  |  ~  |  isatom  |  dim  |  imp
```

APPENDIX D (ctd.)

```
! Some of the conventions of the usual hardware representation of
! identifiers in MSL (see Appendix Z) are ignored below, for the sake
! of brevity.  In particular, script letters are represented by single
! upper-case letters in this Appendix, Greek letters by double lower-
! case letters and Italic lower-case by single lower-case letters.

! Semantics:

λActivate .     ! Activate: [ExpB→D], where D, the domain of inter-
                !           pretation of LAMB, is defined in Table 4.

def E[t] =      ! E: [[Exp+Defs]→ExpB]
clauses t
§ case 'λP.E' <p,e>:    P[p](E[e])
    ! P expands [λB1 B2 ... Bn.E] to [λB1.λB2. ... λBn.E].

  case 'def D in E' <d,e>:    let b = B[d] in
                              App<Abs<b,E[e]>,Fix<b,E[d]>>
    ! App, Abs, etc. construct nodes, and are defined later.

  case 'let D in E' <d,e>:    let b = B[d] in
                              App<Abs<b,E[e]>,E[d]>

  case 'fix B.E' <b,e>:    Fix<b,E[e]>

  case 'E→E,E' <e1,e2,e3>:    Con<E[e1],E[e2],E[e3]>

  case 'clauses E§C§' <e,c>:    let ee = E[e] in
                                App<Abs<Bv(1),C[c](Bv(1))(ee)(Err)>,
                                    Select<ee,Dim<ee>>>
    ! An abstraction is used to avoid excessive expansion.

  case 'E Op E' <e1,op,e2>:    O2[op]<E[e1],E[e2]>

  case 'Op E' <op,e>:    O1[op]<E[e]>

  case 'Op E E' <op,e1,e2>:    O2[op]<E[e1],E[e2]>

  case 'E(E)' <e1,e2>:    App<E[e1],E[e2]>

  case 'E[E/E]' <e1,e2,e3>:    App<Abs<BvTup<Bv(1),Bv(2),Bv(3)>,
                                       Abs<Bv(4),
                                           Con<Eq<Bv(4),Bv(3)>,
                                               Bv(2),
                                               App<Bv(1),Bv(4)>>>>,
                               Tup<E[e1],E[e2],E[e3]>>
    ! The extra abstraction is to gain efficiency with call-by-need.

  case '<E*?>':    Tup(mapt<E,t>)

  case 'I ' case 'N ' case 'S '
  case 'true' case 'false' case 'err':    t
```

APPENDIX D (ctd.)

```
  case 'D*':     Tup(mapt<E,t>)     ! c.f. definition of B below.

  case 'I  P=E' <i,p,e>:     P[p](E[e])

  case  'B=E' <b,e>:     E[e]
§
```

```
and P[t](ff) =     ! P: [Pars→ExpB→ExpB]
clauses t
§ case 'B P' <b,p>:     Abs<b,P[p](ff)>

  case 'B' <b>:     Abs<b,ff>
§
```

```
and B[t] =     ! B: [Defs→Bvs]
clauses t
§ case 'D*':     BvTup(mapt<B,t>)

  case 'I  P=E' <i,p,e>:     Id(i)     ! ‡ Bv(i)

  case 'B=E' <b,e>:     b
§
```

```
and C[t](ff)(gg)(hh) =     ! C: [Cases→ExpB→ExpB→ExpB→ExpB]
clauses t
§ case 'C C' <c1,c2>:     C[c1](ff)(gg)( C[c2](ff)(gg)(hh) )

  case 'case S B:E' <s,b,e>:     Con<S[s](ff),
                                     App<Abs<b,E[e]>,gg>,
                                     hh>

  case 'case S:E' <s,e>:     Con<S[s](ff),E[e],hh>

  case 'default:E' <e>:     E[e]
§
```

```
and S[t](ff) =     ! S: [Strs→ExpB→ExpB]
clauses t
§ case 'S case S' <s1,s2>:     Or<S[s1](ff),S[s2](ff)>

  case 'S ' <i>:     Eq<ff,t>     ! ff will denote a string.
§
```

APPENDIX D (ctd.)

```
and O2[t]<ff,gg> =    ! O2: [[InOp→PreOp2]→[ExpB×ExpB]→ExpB]
clauses t
§ case 'o' case '*' case '||'
  case '+' case '-' case '×'
  case '=' case '<' case '∨'
  case '∧' case 'cat' case '↓':    InOp<ff,label t,gg>

  case '‡':    Not<Eq<ff,gg>>

  case 'aug':    Cat<ff,Tup<gg>>

  case 'pre':    Cat<Tup<ff>,gg>

  case 'node':    Cat<gg,Tup<ff>>
§

and O1[t]<ff> =    ! O1: [PreOp1→ExpB→ExpB]
clauses t
§ case 'pair' case '~'
  case 'isatom' case 'dim':    PreOp<label t,ff>

  case 'seg':    App<Imp<'SegFn'>,ff>

  case 'chars':    App<Imp<'CharsFn'>,ff>

  case 'string':    App<Imp<'StringFn'>,ff>

  case 'mapn':    App<MapN,ff>

  case 'mapt':    App<MapT(MapN),ff>

  case 'label':    Select<ff,Dim<ff>>

  case 'spread':    Minus<Dim<ff>,Num(1)>
§
```

APPENDIX D (ctd.)

```
and MapN =      ! MapN: ExpB
        Fix<Bv(5),
            Abs<BvTup<Bv(1),Bv(2),Bv(3),Bv(4)>,
                Con<LessEq<Bv(3),Bv(4)>,
                    App<Bv(5),
                        Tup<Bv(1),
                            App<Bv(1),Tup<Bv(2),Bv(3)>>,
                            Plus<Bv(3),Num(1)>,
                            Bv(4)>>,
                    Bv(2)>>>
    ! i.e. MapN denotes:
    ! fix mm. λ<f,r,k,n>. k<n → mm<f,f<r,k>,k+1,n>, r .

and MapT(mm) =      ! MapT: [ExpB→ExpB]
        Abs<BvTup<Bv(6),Bv(7)>,
            App<mm,
                Tup<Abs<BvTup<Bv(8),Bv(9)>,
                        Cat<Bv(8),
                            Tup<App<Bv(6),Select<Bv(7),Bv(9)>>>>>,
                    Tup<>,
                    Num(1),
                    Minus<Dim<Bv(7)>,Num(1)>>>>
    ! i.e. MapT(mm) denotes:
    ! λ<f,t>. mm<(λ<r,k>. r cat <f[t↓k]>), <>, 1, ((dim t)-1)> .

and App<ff,gg>      = node '@(@)' <ff,gg>

and Abs<bb,ff>      = node 'λB.E' <bb,ff>

and Fix<bb,ff>      = node 'fix B.E' <bb,ff>

and Con<ff,gg,hh>   = node 'E→E,E' <ff,gg,hh>

and Err             = node 'err' <>

and Select<ff,gg>   = InOp<ff,'↓',gg>

and Dim<ff>         = PreOp<'dim',ff>

and BvTup(tt)       = node '<B*?>' (tt)

and Eq<ff,gg>       = InOp<ff,'=',gg>

and Tup(tt)         = node '<E*?>' (tt)

and Or<ff,gg>       = InOp<ff,'∨',gg>

and Not<ff>         = PreOp<'~',ff>

and Cat<ff,gg>      = InOp<ff,'cat',gg>

and Imp<ff>         = PreOp<'imp',ff>

and Minus<ff,gg>    = InOp<ff,'-',gg>
```

APPENDIX D (ctd.)

```
and LessEq<ff,gg>  = InOp<ff,'≤',gg>

and Plus<ff,gg>    = InOp<ff,'+',gg>

and InOp<ff,ll,gg> = node 'E Op E' <ff,node(ll)<>,gg>

and PreOp<ll,ff>   = node 'Op E'<node(ll)<>,ff>

and Id(i) = node 'I ' <i>

and Bv(i) = node 'I ' <'v',i>

and Num(i) = node 'N ' <i>


result  E o Activate
```

APPENDIX E: 'GRAMMA→GRAM/MSL'

```
'GRAMMA→GRAM/MSL'

! Reduced Syntax for GRAMMA:

! (Gramma)     Gramma ::= Syn Lex Aux

! (Syn)        Syn ::= syn SC*
! (SClause)    SC  ::= Na::=SPL
! (SPhraseL)   SPL ::= SP*
! (SPhrase)    SP  ::= SWL:Sy  |  SWL
! (SWordL)     SWL ::= SW*?
! (SWord)      SW  ::= Na  |  Sy  |  SW*?  |  SW OPT  |  SW REP

! (Lex)        Lex ::= lex LC*?
! (LClause)    LC  ::= Na::=SPL  |  Na=APL  |  Na‡APL

! (Aux)        Aux ::= aux AC*?
! (AClause)    AC  ::= Na=APL  |  Na‡APL
! (APhraseL)   APL ::= AP*
! (APhrase)    AP  ::= Sy  |  Sy...Sy

! (Name)       Na  ::= SN  |  LN  |  LV  |  λ
! (Symb)       Sy  ::= SY


! Reduced Syntax for GRAM:

! (Gram)       Gram ::= S L

! (Syn1)       S   ::= SPL FL
! (SProdL)     SPL ::= SP*
! (SProd)      SP  ::= Na WdL SuL SY
! (Name1)      Na  ::= SN  |  LN  |  LV  |  λ  |  SY
! (WordL)      WdL ::= Wd*?
! (Word)       Wd  ::= SN  |  LN  |  LV  |  SY  |  CH
! (SuccL)      SuL ::= Su*
! (Succ)       Su  ::= (NU )*

! (FollowL)    FL  ::= F*
! (Follow)     F   ::= Na WdL

! (Lex1)       L   ::= LPL APL
! (LProdL)     LPL ::= LP*?
! (LProd)      LP  ::= Na WdL SuL  |  Na=RL  |  Na‡RL

! (AProdL)     APL ::= AP*?
! (AProd)      AP  ::= Na=RL  |  Na‡RL
! (RangeL)     RL  ::= R*
! (Range)      R   ::= CH...CH
```

APPENDIX E (ctd.)

```
let <Aug,Cat,MapL,MapN,MapT> = seg 'Useful Fns 1'

let <<>,<>,Uni> = seg 'Useful Fns 2'

let <endsymb> = seg 'Useful Fns 3'


def Gram[gramma] =    ! :→ Gram    (types of parameters are omitted)
clauses gramma
§ case 'Syn Lex Aux' <syn,lex,aux>:
                    let spl = Gs1[syn] in
                    let foll = Follows[syn] in
                    let lpl = Gl1[lex][syn] in
                    let apl = Ga1[aux] in
                    node 'S L' < node 'SPL FL' <spl,foll>,
                                 node 'LPL APL' <lpl,apl> >
§


and Gs1[syn] =    ! :→ SProdL
clauses syn
§ case 'syn SC*':   node 'SP*' ( RootProd[syn]
                                 pre MapT(Cat)<Gs2[syn]> )
§


and Gs2[sc] =     ! :→ SProd*
clauses sc
§ case 'Na::=SPL' <na,spl>:    Gs3[spl][na]
§


and Gs3[spl][na] =    ! :→ SProd*
clauses spl
§ case 'SP*':    MapT(Aug)< λsp.Gs4[sp][na], spl >
§


and Gs4[sp][na] =    ! :→ SProd
clauses sp
§ case 'SW:Sy' <swl,sy>:
                    let wdl = node 'Wd*?' (Gs5[spl]) in
                    let sul = node 'Su*' (Succs[swl](spread wdl)) in
                    node 'Na WdL SuL SY ' <na,wdl,sul,sy↓1>

  case 'SWL' <swl>:
                    let wdl = node 'Wd*?' (Gs5[swl]) in
                    let sul = node 'Su*' (Succs[swl](spread wdl)) in
                    let s = (spread wdl) ‡ 1 → '',
                            (label wdl↓1) ‡ 'SY ' → '',
                            wdl↓1↓1    in
                    node 'Na WdL SuL SY ' <na,wdl,sul,s>
§
```

APPENDIX E (ctd.)

```
and Gs5[sw1] =      ! :→ Word*
clauses sw1
§ case 'SW*?':      MapT(Cat)<Gs5,sw1>

  case 'Na' case 'Sy' <t>:      <t>

  case 'SW OPT' case 'SW REP' <sw>:      Gs5[sw]
§

and Succs[sw1](n) =      ! :→ Succ*
        ( node '(NU )*' (Eqs[sw1](1)) ) pre
        MapN(Aug)< λj.node'(NU )*'(Seqs[sw1](j)), n >

and Eqs[sw1](j) =      ! :→ Int*
    def Eqs1[sw](i)(k) =      ! :→ Int*
        ~(i<(spread sw)) → <j>,
        let l = k + Hold[sw↓i] in
        ~(j<l) → Eqs1[sw](i+1)(l),
        clauses sw↓i
        § case 'Na' case 'Sy':      <j>
          case 'SW OPT':      j‡(k+1) → Eqs1[sw↓i](1)(k),
                              Eqs1[sw↓i](1)(k) cat Eqs[sw1](l+1)
            ! hence     Eqs[...(Xj...Xl)?...](j)
            ! includes Eqs[...(Xj...Xl)?...](l+1).
          case 'SW REP' case 'SW*?':      Eqs[sw↓i](1)(k)
        §
    in
    Eqs1[sw1](1)(0)

and Seqs[sw1](j) =      ! :→ Int*
    def Seqs1[sw](i)(k) =      ! :→ Int*
        ~(i<(spread sw)) → err,
        let l = k + Hold[sw↓i] in
        ~(j<l) → Seqs1[sw](i+1)(l),
        clauses sw↓i
        § case 'Na' case 'Sy':      Eqs[sw1](j+1)
          case 'SW REP':      j‡l → Seqs[sw↓i](1)(k),
                              Seqs1[sw↓i](1)(k) cat Eqs[sw1](k+1)
            ! hence     Seqs[...Xk(...Xj)*...](j)
            ! includes Eqs[...Xk(...Xj)*...](k+1)
            ! and      Eqs[...Xk(...Xj)*...](j+1)
          case 'SW OPT' case 'SW*?':      Seqs1[sw↓i](1)(k)
        §
    in
    Seqs1[sw1](1)(0)
```

APPENDIX E (ctd.)

```
and Hold[sw] =     ! :→ Int
clauses sw
§ case 'Na'
  case 'Sy':       1
  default:         def Hold1(i) =    ! :→ Int
                       i≤(spread sw) → Hold[sw↓i] + Hold1(i+1),
                       0
                   in Hold1(1)
§
```

```
and RootProd[syn] =    ! :→ SProd
clauses syn
§ case 'syn SC*':   clauses syn↓1
                    § case 'Na::=SPL' <na,spl>:
                           let na1 = node 'SN ' <''> in
                           let wd1 = node 'Wd*?' <na,endsymb> in
                           let su1 = node 'Su*' <
                                     node '(NU )*' <1>,
                                     node '(NU )*' <2>,
                                     node '(NU )*' <3> > in
                           node 'Na WdL SuL SY ' <na1,wd1,su1,''>
                    §
§
```

```
and Gl1[lex][syn] =    ! :→ LProdL
clauses lex
§ case 'lex LC*?':     node 'LP*?' ( MapT(Cat)<Gl2,lex> cat
                                     Implicit[syn] )
§
```

```
and Gl2[lc] =     ! :→ LProd*
clauses lc
§ case 'Na::=SPL' <na,spl>:    Gl3[spl][na]
  case 'Na=APL' <na,apl>:     < node 'Na=RL' <na,Ga3[apl]> >
  case 'Na‡APL' <na,apl>:     < node 'Na‡RL' <na,Ga3[apl]> >
§
```

```
and Gl3[spl][na] =    ! :→ LProd*
clauses spl
§ case 'SP*':     MapT(Aug)< λsp.Gl4[sp][na], spl >
§
```

```
and Gl4[sp][na] =     ! :→ LProd
clauses sp
§ case 'SWL' <swl>:      let wd1 = node 'Wd*?' (Gl5[swl]) in
                         let su1 = node 'Su*' (Succs[swl](spreadwd1)) in
                         node 'Na WdL SuL' <na,wd1,su1>
§
```

APPENDIX E (ctd.)

```
and G15[sw] =     ! :→ Word*
clauses sw
§ case 'SW*?':     MapT(Cat)<G15,sw>
  case 'Na' <na>:     <na>
  case 'Sy' <sy>:     <G16[sy]>
  case 'SW OPT' case 'SW REP' <sw1>:     G15[sw1]
§

and G16[sy] =     ! :→ Word
clauses sy
§ case 'SY ' <i>:     node 'CH ' <Char(i)>
§

and Implicit[syn] =     ! :→ LProd*
    MapL(Aug)<FormLProd,Mentioned[syn]>

and FormLProd[sy] =     ! :→ LProd
clauses t
§ case 'SY ' <i>:   let l = chars i in
                    let na = sy in
                    let wdl = node 'Wd*?' (
                          mapn <  <r,k>. r aug (node'CH '<l↓k>),
                                  <>, 2, (dim s)-1 > ) in
                  ! i.e. omitting the string quotes.
                  let sul = node 'Su*' (
                          mapn < λ<r,k>. r aug (node'(NU )*'<j>),
                                  <>, 1, (dim s)-1 > ) in
                  node 'Na WdL SuL' <na,wdl,sul>
§

and Mentioned[t] =     ! :→ Symb*
clauses t
§ case  'syn SC*'
  case 'SP*'
  case 'SW*?':                MapT(Uni)<Mentioned,t>
  case 'Na::=SPL' <t1,t2>:    Mentioned[t2]
  case 'SWL:Sy'<t1,t2>:       Mentioned[t1]
  case 'SWL'
  case 'SW OPT'
  case 'SW REP' <t1>:         Mentioned[t1]
  case 'Na':                  <>
  case 'Sy' <sy>:             <sy>
§

and Ga1[aux] =     ! :→ AProdL
clauses aux
§ case 'aux AC*?':     node 'AP* ' ( MapT(Aug)<Ga2,aux> )
§
```

APPENDIX E (ctd.)

```
and Ga2[ac] =     ! :→ AProd
clauses ac
§ case 'Name=APL' <na,apl>:  node 'Na=RL' <na,Ga3[apl]>
  case 'Na≠APL' <na,apl>:    node 'Na≠RL' <na,Ga3[apl]>
§

and Ga3[apl] =    ! :→ RangeL
clauses apl
§ case 'AP*':    node 'R*' ( MapT(Aug)<Ga4,apl> )
§

and Ga4[ap] =     ! :→ Range
clauses ap
§ case 'Sy' <sy>:                node 'CH ...CH ' <Ga5[sy],Ga5[sy]>
  case 'Sy...Sy' <sy1,sy2>:      node 'CH ...CH ' <Ga5[sy1],Ga5[sy2]>
§

and Ga5[sy] =     ! :→ Int
clauses sy
§ case 'SY ' <i>:    Char(i)
§

and Char(i) =     ! :→ Int
    let s = chars i in
    (dim s)=3 → s↓2,
    err

and Follows[syn1] =     ! :→ FollowL
    (seg 'Follows')[syn1]

result Gram
•
```

APPENDIX E (ctd.)

```
'Follows/MSL'

λ spl.

let <Aug,Cat,MapL,MapN> = seg 'Useful Fns 1'

let <<>,IsIn,Uni,Closure,ListClosure> = seg 'Useful Fns 2'

let <<>,Find,IsVn> = seg 'Useful Fns 3'

let <A,N,X,Succ> = (seg 'Syn Fns')<spl,<>>

let s = spread spl

let vn =
    MapN(Uni)< λp.<A(p)>, s>

def ve =
    ListClosure<Vanish,<>>

and Vanish(l) =    ⍰ :→ Name*
    MapN(Uni)<Vanish1(l),s>

and Vanish1(l)(p) =    ⍰ :→ Name*
    IsIn<Reach<l,p,0>,N(p)+1> → <A(p)>,
    <>

and Reach<l,p,j> =    ⍰ :→ Int*
    Closure< λk.Reach1<l,p,k>, Succ<p,j> >

and Reach1<l,p,k> =    ⍰ :→ Int*
    k=(N(p)+1) → <>,
    IsIn<l,X<p,k>> → Succ<p,k>,
    <>
```

APPENDIX E (ctd.)

```
def first =
    node 'F*' ( MapL(Aug)<FormFirst,vn> )

and FormFirst(a) =      ! :→ Follow
    node 'Na WdL' <a, node 'Wd*?'(Closure<First1,<a>>) >

and First1(x) =      ! :→ Word*
    IsVn(x) → MapN(Uni)<First2(x),s>,
    <>

and First2(x)(p) =      ! :→ Word*
    IsEq<A(p),x> → MapL(Uni)< Symb(p), Reach<ve,p,0> >,
    <>

and Symb(p)(j) =      ! :→ Word*
    (1≤j) ^ (j≤N(p)) → <X<p,j>>,
    <>
```

```
def last =
    node 'F*' ( MapL(Aug)<FormLast,vn> )

and FormLast(a) =      ! :→ Follow
    node 'Na WdL' <a, node 'Wd*?'(Closure<Last1,<a>>) >

and Last1(x) =      ! :→ Word*
    IsVn(x) → MapN(Uni)<Last2(x),s>,
    <>

and Last2(x)(p) =      ! :→ Word*
    MapL(Uni)< Symb(p), ReachBack<ve,p,N(p)+1> >

and ReachBack<l,p,j> =      ! :→ Int*
    MapN(Cat)<
        λk. IsIn<Reach<l,p,k-1>,j>→<k-1>,<> ,
        N(p) >
```

APPENDIX E (ctd.)

```
def follows =
    node 'F*' ( MapL(Aug)<FormFollow,vn> )

and FormFollow(a) =    ! :→ Follow
    node 'Na WdL' <a, node 'Wd*?'(MapN(Uni)<Follow1(a),s>) >

and Follow1(a)(p) =    ! :→ Word*
    MapL(Cat)< OnlyVn, MapN(Uni)<Follow2<a,p>,N(p)> >

and OnlyVn(x) =    ! :→ Word*
    IsVn(x) → <x>,
    <>

and Follow2<a,p>(j) =    ! :→ Word*
    IsIn< Find(last)(X<p,j>), a> →
        MapL(Uni)< Find(first), MapL(Uni)<Symb(p),Reach<ve,p,j>> >,
    <>

result follows
```

APPENDIX F: 'Lex/MSL'

```
'Lex/MSL'

λ<lpl,apl>.

let <Aug,<>,MapL,MapN,MapT> = seg 'Useful Fns 1'

let <<>,<>,Uni> = seg 'Useful Fns 2'

let <endsymb,<>,IsVn,IsVc,IsVv> = seg 'Useful Fns 3'

let <Start,Peep,Scan,IsEnd,Out,Finish> = seg 'Useful Fns 4'

let <A,IsOK,IsFinal,Succ> = (seg 'Lex Fns')<lpl,apl>


def LexLoop(z) =
    IsEnd || λb.
    b → z,
    Peep || λch.
    let firststates = MapN(Aug)< λp.<p,0>, spread lpl > in
    LexCycle<firststates, Advance<firststates,ch>, <>, ch > ||
    LexLoop(z)


and LexCycle<states,states1,chl,ch>(z) =
    (dim states1)=0 → OutSymb<states,chl> || z,
    Scan ||
    Peep || λch1.
    LexCycle<states1, Advance<states1,ch1>, chl aug ch, ch1>(z)


and Advance<states,ch> =
    MapL(Uni)<Advance1(ch),states>


and Advance1(ch)<p,j> =
    IsOK(ch)<p,j+1> → Advance2<p,j+1>,
    <>


and Advance2<p,i> =
    MapL(Aug)< λj.<p,j-1>, Succ<p,i> >
```

APPENDIX F (ctd.)

```
and OutSymb<states,chl>(z) =
    let ps = MapL(Uni)<Final,states> in
    (dim ps)≠1 → err,
    let x = A(ps↓1) in
    IsVc(x) → Out(x) || z,
    IsVv(x) → Out(x) ||
              Out(string chl) || z,
    z


result Start o LexLoop(Out(endsymb)||Finish)
•
.
```

APPENDIX F (ctd.)

```
'Lex Fns/MSL'

λ<lpl,apl>.

let <Aug,<>,<>,<>,MapT> = seg 'Useful Fns 1'

let A(p) =
    lpl↓p↓1

def IsOK(ch)<p,j> =
clauses lpl↓p
§ case 'Na WdL SuL' <na,wdl,sul>:
                    let x = wdl↓j in
                    clauses x
                    § case 'LN '<i>:    IsInRange<ch,FindRange(i)>
                      case 'CH '<i>:    ch = i
                    §
  case 'Na=RL' case 'Na≠RL':
                    j≠1 → false,
                    IsInRange<ch,lpl↓p>
§

and IsInRange<ch,ap> =
clauses ap
§ case 'Na=RL' <na,rl>:    IsInPart<ch,rl,1,spread rl>
  case 'Na≠RL' <na,rl>:    ~IsInPart<ch,rl,1,spread rl>
§

and IsInPart<ch,rl,k,n> =
    ~(k≤n) → false,
    (rl↓k↓1≤ch) ^ (ch≤rl↓k↓2) → true,
    IsInPart<ch,rl,k+1,n>

and FindRange(i) =
    FindRange1<i,apl,1,spread apl>

and FindRange1<i,apl,k,n> =
    ~(k≤n) → err,
    apl↓k↓1↓1 = i → apl↓k,
    FindRange1<i,apl,k+1,n>

let IsFinal<p,j> =
clauses lpl↓j
§ case 'Na WdL SuL' <na,wdl,sul>:    j = (spread wdl)
  case 'Na=RL' case 'Na≠RL':         j = 1
§
```

APPENDIX F (ctd.)

```
let Succ<p,j> =
clauses lpl↓p
§ case 'Na WdL SuL' <na,wdl,sul>:    MapT(Aug)< λk.k, sul↓j >
  case 'Na=RL' case 'Na≠RL':         j=0 → <1>,
                                     j=1 → <2>,
                                     err
§

result <A,IsOK,IsFinal,Succ>
•
```

APPENDIX G: 'Syn/MSL'

```
'Syn/MSL'

λ<spl,foll>.

let <Aug,Cat,MapL,MapN> = seg 'Useful Fns 1'

let <<>,Uni,IsIn,Closure,ListClosure,IsEq> = seg 'Useful Fns 2'

let <Find,IsVn,IsVc,IsVv> = seg 'Useful Fns 3'

let <Start,Peep,Scan> = seg 'Useful Fns 4'

let <nullstk,Push,Top,Pop> = seg 'Useful Fns 5'

let <A,N,X,Succ,L,CanFollow> = (seg 'Syn Fns')<spl,foll>

let s = spread spl
```

```
def Syn(z) =
    let nodestk = nullstk in
    let firststate = <1,0,0> in
    let statestk = Push<firststate,nullstk> in
    SynCycle<nodestk,statestk>(z)


and SynCycle<nodestk,statestk>(z) =
    Step1(Top(statestk)) || λsm1.
    Step2<sm1,nodestk,statestk> || λ<sm2,nodestk1,statestk1,xm1>.
    Step3<sm2,xm1> || λsn.
    IsFinal(sn) → z(Root(nodestk1)),
    SynCycle<nodestk1,Push<sn,statestk1>>(z)


and Step1(sm)(z) =
    z( Closure<Predict,sm> )


and Predict<p,j,k> =
    MapL(Uni)<Predict1(p),Succ<p,j>>


and Predict1(p)(i) =
    i = (N(p)+1) → <>,
    IsVn(X<p,i>) → MapN(Cat)<Predict2(X<p,i>),s>,
    <>
```

APPENDIX G (ctd.)

```
and Predict2(a)(q) =
    IsEq<a,A(q)> → MapL(Aug)< λj.<q,j-1,0>, Succ<q,0> >,
    <>

and Step2<sm1,nodestk,statestk>(z) =
    Peep || λy.
    IsInZ<y,sm1> → Accept(y)<sm1,nodestk,statestk> || z,
    Decision<y,sm1> || λd.
    z( Reduce(d)<nodestk,statestk> )

and IsInZ<y,sm1> =
    IsInZ1<y,sm1,1,dim sm1>

and IsInZ1<y,sm1,i,n> =
    ~(i≤n) → false,
    let <p,j,k> = sm1↓i in
    j=N(p) → IsInZ1<y,sm1,i+1,n>,
    IsEq<y,X<p,j+1>> → true,
    IsInZ1<y,sm1,i+1,n>

and Accept(y)<sm1,nodestk1,statestk>(z) =
    Scan ||
    IsVv(y) →
        Peep || λi.
        Scan ||
        let nodestk1 = Push<y,Push<i,nodestk>> in
        z<sm1,nodestk1,statestk,y>,
    let nodestk1 = Push<y,nodestk> in
    z<sm1,nodestk,statestk,y>

and Decision<y,sm1>(z) =
    let r = MapL(Uni)<Reducible(y),sm1> in
    (dim r)≠1 → err,
    z(r↓1)

and Reducible(y)<p,j,k> =
    j≠N(p) → <>,
    ~CanFollow<p,y> → <>,
    <<p,j,k>>
```

APPENDIX G (ctd.)

```
and Reduce(d)<nodestk,statestk> =
    Reduce1(d)<nodestk,statestk,<>>

and Reduce1<p,j,k><nodestk,statestk,r> =
    k=0 →
        let sm2 = Closure<Predict,Top<statestk>> in
        let node = (L(p)='?') → r↓1, node(L(p))(r) in
        let nodestk1 = Push<A(p),Push<node,nodestk>> in
        <sm2,nodestk1,statestk,A(p)>,
    Reduce1<p,j,k-1>(Remove<nodestk,statestk,r>)

and Remove<nodestk,statestk,r> =
    let x = Top(nodestk) in
    let nodestk1 = Pop(nodestk) in
    let statestk1 = Pop(statestk) in
    IsVc(x) → <nodestk1,statestk1,r>,
    let i = Top(nodestk1) in
    let nodestk2 = Pop(nodestk1) in
    <nodestk2,statestk1, i pre r>

and Step3<sm2,xm1>(z) =
    z( MapL(Uni)<Advance(xm1),sm2> )

and Advance(xm1)<p,j,k> =
    j=N(p) → <>,
    ~IsEq<xm1,X<p,j>> → <>,
    MapL(Uni)< λi.<p,i-1,k+1>, Succ<p,j+1> >

and IsFinal(sn) =
    (dim sn)≠1 → false,
    IsEq<sn↓1,<1,2,2>>

and Root(nodestk) =
    Top(Pop(Pop(nodestk)))

result Start o Syn(λt.t)
•
```

APPENDIX G (ctd.)

```
'Syn Fns/MSL'

 <spl,foll>.

let <Aug,<>,<>,<>,MapT> = seg 'Useful Fns 1'

let <<>,IsIn> = seg 'Useful Fns 2'

let <<>,Find> = seg 'Useful Fns 3'

let A(p) =
    spl↓p↓1

let N(p) =
    spread spl↓p↓2

let X<p,j> =
    spl↓p↓2↓j

let Succ<p,j> =
clauses spl↓p
§ case 'Na WdL SuL SY ' <na,wdl,sul,i>:
                    MapT(Aug)< λk.k, sul↓j >
§

let L(p) =
    spl↓p↓4

let CanFollow<p,y> =
    let nas = Find(foll)(A(p)) in
    IsIn<nas,y>

result <A,N,X,Succ,L,CanFollow>
•
```

APPENDIX H: 'Useful Fns/MSL'

```
'Useful Fns 1/MSL'

let MapL(Inc)<f,l> =
    mapn < λ<r,k>.Inc<r,f(l↓k)>,
           <>, 1, dim l >


let MapN(Inc)<f,n> =
    mapn < λ<r,k>.Inc<r,f(k)>,
           <>, 1, n >


let MapT(Inc)<f,t> =
    mapn < λ<r,k>.Inc<r,f(t↓k)>,
           <>, 1, spread t >


let Aug<r,x> =
    r aug x


let Cat<r,x> =
    r cat x

result <Aug,Cat,MapL,MapN,MapT>
•
```

APPENDIX H (ctd.)

```
'Useful Fns 2/MSL'

let <<>,Cat,MapL> = seg 'Useful Fns 1'


def IsEq<x,y> =
    (isatom x) ^ (isatom y) → x=y,
    (isatom x) v (isatom y) → false,
    (dim x) ≠ (dim y) → false,
    IsEqL<x,y,1,dim x>

and IsEqL<x,y,i,n> =
    ~(i≤n) → true,
    ~IsEq<x↓i,y↓i> → false,
    IsEqL<x,y,i+1,n>

def IsIn<s,x> =
    IsIn1<s,x,1,dim s>

and IsIn1<s,x,i,n> =
    ~(i≤n) → false,
    IsEq<x,s↓i> → true,
    IsIn1<s,x,i+1,n>

let Uni<r,s> =
    MapL(Cat)< (λx.IsIn<r,x>→x><>,<x>), s>


def Closure<f,s> =
    Closure<f,s,1>


and Closure1<f,s,i> =
    ~(i≤(dim s)) → s,
    Closure1<f, Uni<s,f(s↓i)>, i+1>

def ListClosure<f,s> =
    ListClosure1<f,s,dim s>


and ListClosure1<f,s,i> =
    ~(i≤(dim s)) → s,
    ListClosure1<f, Uni<s,f(s)>, dim s>


result <IsEq,IsIn,Uni,Closure,ListClosure>
•
```

APPENDIX H (ctd.)

```
'Useful Fns 3/MSL'

let <Aug,<>,<>,<>,MapT> = seg 'Useful Fns 1'

let <IsEq> = seg 'Useful Fns 2'


let endsymb = node 'LN ' <''>


def Find(foll)(a) =
    ~IsVn(a) → <a>,     ! used only in Follow2,'Follows/MSL'.
    clauses foll
    § case 'F*':    Find1<foll,a,1,spread foll>
    §

and Find1<foll,a,i,n> =
    ~(i≤n) → err,
    clauses foll↓i
    § case 'Na WdL' <na,wdl>:
                    IsEq<a,na> → MapT(Aug)<  x.x, wdl>,
                    Find1<foll,a,i+1,n>
    §

and IsVn(a) =
    IsNa<a,1>

and IsVc(a) =
    IsNa<a,2>

and IsVv(c) =
    IsNa<a,3>

and IsNa<a,n> =
clauses a
§ case 'SN ':    n=1
  case 'LN '
  case 'SY ':    n=2
  case 'LV ':    n=3
  case 'λ':      false
§


result <endsymb,Find,IsVn,IsVc,IsVv>
```

APPENDIX H (ctd.)

```
'Useful Fns 4/MSL'

 let Start(in) =
     <in,1,<>>

let Peep(z)<in,ptr,out> =
     z(in↓ptr)<in,ptr,out>

let Scan(z)<in,ptr,out> =
     z<in,ptr+1,out>

let IsEnd(z)<in,ptr,out> =
     z(~(ptr≤(dim in)))<in ptr,out>

let Out(z)<in,ptr,out> =
     z<in,ptr, out aug x >

let Finish<in,ptr,out> =
     out

result <Start,Peep,Scan,IsEnd,Out,Finish>
```

APPENDIX H (ctd.)

```
'Useful Fns 5/MSL'

let nullstk = err

let Push<x,stk> = <x,stk>

let Top(stk) = stk↓1

let Pop(stk) = stk↓2

result <nullstk,Push,Top,Pop>
•
```

APPENDIX I: 'TestL-Semantics'

```
'TestL-Semantics'

λtto. λ<Locate,Update,Contents,Apply,Read,Write,Start,
 Finish>.
 ( <CC,DD,IId,IIl,II,EE,PP,
   DiOpVal,MonOpVal,NumVal,Lay>. Start o
   CC(tto)(err) || Finish)(
   fix <CC,DD,IId,IIl,II,EE,PP,
    DiOpVal,MonOpVal,NumVal,Lay>.
    <λtt. λr. λc.
      (λv1.
       v1 = 'begin C end' →
        (λ<cc>.
          CC(cc)(r)(c))(tt),

       v1 = 'while B do C' →
        (λ<bb,cc>. fix c1. EE(bb)(r) *
          λb.
          b →
           CC(cc)(r)(c1),
           c)(tt),

v1 = 'I:=E' →
 (λ<ii,ee>.
   (λ<a,t>.
    ~ (t = 'locn') → err,
     EE(ee)(r) *
      λv. Update(<a, v>) o c)(
     r(
       II(ii))))(tt),

v1 = 'write E' →
 (λ<ee>. EE(ee)(r) *
   λv. Write(v) o c)(tt),

 v1 = 'res E' →
  (λ<ee>.
    EE(ee)(r))(tt),

  v1 = 'D;C' →
   (λ<dd,cc>. DD(dd)(r) *
     λd. CC(cc)(
       Lay(
         <r,
          <d>,
          <IId(dd)>>)) || c)(tt),

   v1 = 'C;C' →
    (λ<cc1,cc2>. CC(cc1)(r) ||
```

APPENDIX I (ctd.)

```
                CC(cc2)(r) || c)(tt),
              err)(
      tt ↓ (dim tt)),

  λtt. λr.
   (λv1.
    v1 = 'var I:=E' →
     (λ<ii,ee>. EE(ee)(r) *
       λv.
        Locate(v))(tt),

    v1 = 'fun I(Il)=E' →
     (λ<ii,iil,ee>. pair (fix f. λp.
        (λr1.
          (λr2.
            EE(ee)(r2))(
            Lay(
              <r1,
               <f>,
               <IId(tt)>>)))(
          Lay(
            <r, p,
             IIl(iil)>))))(tt),
     err)(
    tt ↓ (dim tt)),

λtt.
 (λv1.
  v1 = 'var I:=E' →
   (λ<ii,ee>.
     <II(ii), 'locn'>)(tt),

  v1 = 'fun I(Il)=E' →
   (λ<ii,iil,ee>.
     <II(ii), 'fn'>)(tt),
   err)(
  tt ↓ (dim tt)),

λtt.
 (λv1.
  v1 = 'I*' →
   (λ<v6,v7>.
     (fix v5. λ<v1,v2,v3,v4>.
      v3 ≤ v4 →
       v5(
         <v1,
          v1(<v2, v3>),
          v3 + 1, v4>),
       v2)(
       <λ<v8,v9>. v8 cat (<v6(
            v7 ↓ v9)>),
        <>, 1,
        (dim v7) - 1>))(
     <λtt1.
       <II(tt1), 'num'>, tt>),
   err)(
```

APPENDIX I (ctd.)

```
      tt ↓ (dim tt)),

    λtt.
     (λv1.
      v1 = 'I' →
       (λ<n>. n)(tt),
       eerr)(
       tt ↓ (dim tt)),

    λtt. λr.
     (λv1.
      (v1 = 'B Op B') ∨
       v1 = 'E Op E' →
       (λ<ee1,op,ee2>. EE(ee1)(r) *
         λe1. EE(ee2)(r) *
          λe2. pair (DiOpVal(op)(<e1, e2>)))(tt),

       v1 = 'Op B' →
        (λ<op,ee>. EE(ee)(r) *
          λe. pair (MonOpVal(op)(e)))(tt),

        v1 = 'B→E,E' →
         (λ<ee1,ee2,ee3>. EE(ee1)(r) *
           λb.
           b →
            EE(ee2)(r),
            EE(ee3)(r))(tt),

    v1 = 'valof C' →
     (λ<cc>. CC(cc)(r) ||
       pair err)(tt),

     v1 = 'I(El)' →
      (λ<ii,eel>.
        (λ<f,t>.
         ~ (t = 'fn') → err,
          PP(eel)(r)(<>) *
           λp.
            Apply(f)(p))(
          r(
            II(ii))))(tt),

      v1 = 'I' →
       (λ<ii>.
         (λ<z,t>.
          t = 'fn' → err,

           t = 'locn' →
            Contents(z),

            t = 'num' →
             pair z,
             err)(
           r(
             II(ii))))(tt),
```

APPENDIX I (ctd.)

```
            v1 = 'N' →
             (λ<n>. pair (NumVal(n)))(tt),

             v1 = 'read' →
              (λ<>. Read)(tt),
              err)(
        tt ↓ (dim tt)),

    λtt. λr. λp.
     (λv1.
      v1 = 'E,El' →
       (λ<ee,eel>. EE(ee)(r) *
         λe.
          PP(eel)(r)(
            p cat (<e>)))(tt),

       v1 = 'E' →
        (λ<ee>. EE(ee)(r) *
          λe. pair (p cat (<e>)))(tt),
        err)(
       tt ↓ (dim tt)),

    λtt. λ<e1,e2>.
     (λv1.
      v1 = '+' →
       e1 + e2,

       v1 = '-' →
        e1 - e2,

       v1 = 'X' →
        e1 X e2,

        v1 = '=' →
         e1 = e2,

         v1 = '≤' →
          e1 ≤ e2,

          v1 = 'v' →
           e1 v e2,

           v1 = '^' →
            e1 ^ e2,
            err)(
       tt ↓ (dim tt)),

    λtt. λe.
     (λv1.
      v1 = '~' →
       ~ e,
       err)(
       tt ↓ (dim tt)),

    λn.
     (λ<z>.
```

APPENDIX I (ctd.)

```
      (λs.
        (fix v5. λ<v1,v2,v3,v4>.
         v3 ≤ v4 →
          v5(
            <v1,
             v1(<v2, v3>),
             v3 + 1, v4>),
          v2)(
          <λ<m,i>. ((m X 10) ÷ (s ↓ i)) - z, 0, 1,
           dim s>))(
        (imp 'CharsFn')(n)))(
      (imp 'CharsFn')('0')),

λ<r,p,iil>.
  (fix v5. λ<v1,v2,v3,v4>.
   v3 ≤ v4 →
    v5(
      <v1,
       v1(<v2, v3>),
       v3 + 1, v4>),
    v2)(
    <λ<r1,n>.
      (λ<v1,v2,v3>. v4.
       v4 = v3 → v2,
        v1(v4))(
      <r1,
       <p ↓ n,
        (iil ↓ n) ↓ 2>,
       (iil ↓ n) ↓ 1>), r, 1,
dim iil>)>)
```

APPENDIX J: 'TestL-Implementation'

```
'TestL-Implementation'

Prog.
(λ<Locate,Update,Contents,Apply,Read,Write,Start,
  Finish>.
  Prog(<Locate, Update, Contents, Apply, Read, Write, Start, Finish>))(
  <λv. λ<m,n,i,o>.
    <n + 1,
     <<m,
       n + 1, v>,
      n + 1, i, o>>,

   λ<a,v>. λ<m,n,i,o>.
    <<m, a, v>, n, i, o>,

   λa. λs.
    (λFind.
      <Find(
         s ↓ 1), s>)(
      fix Find. λm.
      isatom m → err,

       (m ↓ 2) = a →
       m ↓ 3,
       Find(
         m ↓ 1)),

   λf. λp.
    f(p),

   λ<m,n,i,o>.
    <i ↓ 1,
     <m, n,
      i ↓ 2, o>>,

   λz. λ<m,n,i,o>.
    <m, n, i,
     o cat (<z>)>,

   λi.
    <err, 0, i,
     <>>,

   λ<m,n,i,o>. o>)
```

APPENDIX Z: Hardware Representation

Hardware Representation :	Publication Language
a, b, c, ... :	α, β, γ, ...
a1, a', a'', ...:	α_1, α', α'', ...
aa, bb, aa1, ...:	a, b, a_1, ...
New, Lay, ... :	*New*, *Lay*,
AA, BB, AA1, ...:	$\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}_1$, ...
def, let, ... :	def, let, ...
N, Exp, ... :	N, Exp, ...
( )*? :	{ }*?
[ ] :	⟦ ⟧
" " :	" "
! :	!
° :	°
* :	*
|| :	‖
EE: [...] :	$\mathfrak{E} \in$ [...]

REFERENCES

[1] J. H. Morris, Jr.:
Lambda-Calculus Models of Programming Languages.
Ph.D. Thesis, M.I.T., 1968.

[2] R. Milner:
Implementation and Applications of Scott's Logic for Computable Functions.
SIGPLAN Notices 7:1, pp. 1-6 (1972);
(Proc. ACM. Conf. on Proving Assertions about Programs, New Mexico, 1971).

[3] C. B. Jones:
Formal Definition in Program Development.
In Springer Verlag Lecture Notes in Computer Science No. 23, 1975, on Programming Methodology.

[4] R. E. Milne:
The Formal Semantics of Computer Languages and their Implementations.
Ph.D. Thesis, Cambridge University, 1974.

[5] J. C. Reynolds:
Definitional Interpreters for Higher-Order Programming Languages.
Proc. 27th ACM Nat. Conf., 1972, pp. 717-740.

[6] D. Scott, C. Strachey:
Toward a Mathematical Semantics for Computer Languages.
Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, 1971; and Technical Monograph PRG-6, O.U.C.L., Oxford.

[7] C. Strachey, C. P. Wadsworth:
Continuations: A Mathematical Semantics for Handling Full Jumps.
Technical Monograph PRG-11, O.U.C.L., Oxford (1974).

[8] R. D. Tennent:
Mathematical Semantics and Design of Programming Languages.
Ph.D. Thesis, Univ. of Toronto, 1973.

[9] C. A. R. Hoare:
An Axiomatic Basis for Computer Programming.
Comm. ACM 12:10, pp. 576-583 (1969).

[10] C. A. R. Hoare, N. Wirth:
An Axiomatic Definition of the Programming Language PASCAL.
Acta Informatica 2:4, pp. 335-355 (1973).

[11] P. Lucas, K. Walk:
On the Formal Description of PL/I.
Ann. Rev. Automatic Programming 6, pp. 105-182 (1969).

[12] P. J. Landin:
The Mechanical Evaluation of Expressions.
Comp. J. 6:4, pp. 308-320 (1964).

[13] P. J. Landin:
A Correspondence between ALGOL 60 and Church's Lambda-Notation.
Comm. ACM 8:2, pp.89-101 and 8:3, pp.158-165 (1965).

[14] N. Wirth, H. Weber:
EULER - A Generalization of ALGOL and its Definition.
Comm. ACM 9:1, pp.13-23 and 9:2, pp.89-99 (1966).

[15] G. T. Ligler:
A Survey of Approaches to Programming Language Semantics.
M.Sc. Dissertation, Oxford University, 1973.

[16] D. Scott:
Outline of a Mathematical Theory of Computation.
Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems, 1970, pp. 169-176; and Technical Monograph PRG-2, O.U.C.L., Oxford.

[17] D. Scott:
The Lattice of Flow Diagrams.
Technical Monograph PRG-3, O.U.C.L., Oxford (1970).

[18] D. Scott:
Data Types as Lattices.
To appear in Springer Lecture Notes (1975).

[19] A. Evans, Jr.:
PAL - A Reference Manual and Primer.
Course Notes, Dept. Elec. Eng., M.I.T., 1969.

[20] D. Scott:
Continuous Lattices.
Proc. 1971 Dalhousie Conf., Springer Lecture Notes;
and Technical Monograph PRG-7, O.U.C.L., Oxford.

[21] A. Church:
The Calculi of Lambda-Conversion.
Ann. of Math. Studies 6, Princeton, 1951.

[22] C. P. Wadsworth:
Semantics and Pragmatics of the Lambda-Calculus.
D.Phil. Thesis, Oxford University, 1971.

[23] J. C. Reynolds:
On the Relation between Direct and Continuation Semantics.
Proc. Second Coll. on Automata, Languages and Programming, Saarbrucken, 1974.

[24] M. J. Richards:
The BCPL Reference Manual.
Technical Memo 69/1, Cambridge University Math. Lab.

[25] P. Naur (Ed.):
The Revised Report on the Algorithmic Language ALGOL 60.
Comm. ACM 6:1, pp. 1-17 (1963).

[26] C. Strachey:
An Assignment Language.
Unpublished Course Note, 1970.

[27] H. B. Curry, R. Feys, W. Craig:
Combinatory Logic.
North-Holland, 1968.

[28] J. Vuillemin:
Correct and Optimal Implementations of Recursion in a Simple Programming Language.
Rapport de Recherche No. 24, IRIA, 1973.

[29] T. Anderson, J. Eve, J. J. Horning:
Efficient LR(1) Parsers.
Acta Informatica 2:1, pp. 12-39 (1973).

[30] S. Ginsburg:
The Mathematical Theory of Context-Free Languages.
McGraw-Hill, 1966.

[31] H. Bekić, D. Bjørner, W. Henhapl, C. B. Jones, P. Lucas:
A Formal Definition of a PL/I Subset.
TR 25.139, I.B.M. Laboratory, Vienna (1974).

[32] J. Feldman, D. Gries:
Translator Writing Systems.
Comm. ACM 11:2, pp. 77-113 (1968).

[33] R. M. Burstall:
Proving Properties of Programs by Structural Induction.
Comp. J. 12:1, pp. 41-48 (1969).