A λ-calculus Interpreter and its Relationship to Algorithmic Languages

by

Ian Kirkland Livingstone

Thesis submitted for the degree of Doctor of Philosophy in the University of London.

Institute of Computer Science

A λ-calculus Interpreter and its Relationship to Algorithmic Languages

by

Ian Kirkland Livingstone

ABSTRACT

One particular interpreter for the λ-calculus is analysed in detail. The interpreter is presented both formally and informally: an informal discussion leading to a rigorous formal definition. The main results show the suitability of the λ-calculus and associated interpreter for explicating the semantics of high-level block-structure programming languages. ALGOL 60 is used as the model of high-level block-structure programming languages for the purpose of the analysis.

A method for specifying complex data structures is developed, the data structures being represented as graphs. This method is used for the formal definition of the λ-calculus interpreter.

Possible generalisations of the machine are considered, and directions for future research indicated.

ACKNOWLEDGEMENT

I express my appreciation to the staff of the Institute of Computer Science, especially to my supervisor W.L. Benedict Nixon, with whom I have had many interesting and fruitful discussions, and to Michael Bell, who introduced me to λ-expressions.

A special acknowledgement is due to Peter Landin, whose work in this field has been the inspiration for my own.

I also wish to express my gratitude to the Science Research Council for the financial support which made this research possible.

CONTENTS

	Page
ABSTRACT	2
ACKNOWLEDGEMENT	3
Introduction	6
Chapter 1. The λ-calculus	9
1.1. Informal Introduction	9
1.2. Basic Theory	17
1.3. λ-definability	31
1.3.1. Boolean Algebra	32
1.3.2. Number-Theoretic Functions	34
1.3.3. Data Structures	46
1.4. Recursive definitions	53
1.5. McCarthy Conditionals	66
1.6. μ-recursive function - λ-calculus equivalence	68
Chapter 2. Structure Definitions	76
2.1. Structure Definitions and Graphs	76
2.2. Interpreters	93
Chapter 3. The S.C.E. Machine	98
3.1. Notation and Machinery	98
3.2. Introduction to the Machine	104
3.2.1. The S.C. Machine	104
3.2.2. Introduction of E Component	112
3.2.3. Formal Definition of S.C.E. Machine	130
3.2.4. An Extension to the Basic Machine	134
3.2.5. Incorporation of Conditionals into the Machine	136

	Page
Chapter 4. The I.S.C.E.M.O. Machine	138
4.1. The S.C.E.M. Machine	139
4.2. Incorporation of I and O Components	143
4.3. Recursion on the I.S.C.E.M.O. Machine	146
4.3.1. ALGOL 60 Labels	153
4.3.2. ALGOL 60 Procedures	156
4.4. Examples of ALGOL 60 Programs	162
4.5. Remaining Constructs	167
4.5.1. Arrays	167
4.5.2. Switch, For Loop and Own Variables	170
4.6. Formal Definition of the I.S.C.E.M.O. Machine	174
Chapter 5. Developments	180
5.1. Simplifications of the Machine	180
5.2. Considerations Necessary for ALGOL 68	185
5.3. Formal Definition of the Machine	190
5.4. An Extension to the Machine	194
5.5. Directions for Future Research	195
5.5.1. Parallel Processing	195
5.5.2. Modes	198
5.5.3. The λ-calculus Revisited	199
5.6. Concluding remarks	200
APPENDIX A - figures A1-A43	203
APPENDIX B - figures B1-B14	266
References	272

Introduction

Many authors have used the λ-calculus to explicate the mechanisms of block-structure languages, and have demonstrated its relevance to programming language structure and design. An onlooker may ask "Why should we want a machine for evaluating λ-expressions; and supposing we do, why do we have to formalize it so precisely?". Well, real programming languages, infested as they are by syntactic and semantic details, are difficult objects to study. Some relief to such a study is afforded by the similarities to one another which many programming languages show. Some of these are more fundamental than mere syntactic similarities; they go right down to the foundations on which the languages are based. One would like to strip away the irrelevant syntactic trappings from programming languages, and study the resulting "core language" which underlies so large a class of programming languages.

Now, the λ-calculus is a mathematical system which models the processes of functional application and substitution. It can therefore be taken as a suitable approximation to the "core language": suitable because it is precisely defined, because several questions of deep theoretical importance about it have been answered, and because it is relatively well understood. The processes of functional application and substitution are fundamental to many programming languages, and so one may expect a study of the methods used in implementing the λ-calculus to yield real insights into them. Peter Wegner, in particular, has advocated (see e.g. W2) a study of the λ-calculus as an aid to understanding the principles underlying a wide spectrum of programming languages.

λ-calculus interpreters (which I call "λ-machines") help in this process by automating the reduction algorithms, thus enabling us to "see"

(literally, if our trace routines are effective enough) clearly what is involved in the reduction processes. We can then gain greater insight into a particular programming language by choosing a λ-machine whose operation mirrors the "real-life" evaluation strategy of the language under consideration. Thus, for ALGOL 60 we might consider a fixed-program λ-machine which uses a stack, and treats an occurrence of a variable bound to a λ-expression as a procedure call. Clement McGowan in M3 defines and proves correct several quite different λ-machines, including one of the type just mentioned.

The λ-calculus does more than just illuminate programming language design and semantics, it also gives insight into programming language implementation. As will be seen, Landin's S.E.C.D. machine[L1,L2], and my modifications of it, throw considerable light on such an implementation-dependent aspect as storage allocation, in languages as far apart as ALGOL and LISP.

I have chosen (a modification of) Peter Landin's S.E.C.D. machine[L1] as the basis for my work for two reasons. First, it is the premier λ-calculus interpreter; it has been the inspiration for all subsequent researches. The ideas embodied in it have been widely used elsewhere, notably by the VDL group[L5, but see also L4,L6,N1,W3]. Secondly, I believe it to show the workings of the λ-calculus in the most perspicuous manner possible. That is, I consider that my S.C.E. machine, although it is not a normal-order evaluation machine, shows all the essential properties of a λ-machine in a manner which is clearer than that of other types of machine such as stack and pointer machines. In particular, the interlinking of the environments is made quite explicit in my machine, while it is obscured in VDL-style machines which represent pointers as integer indexes, use dumps, or generate unique names for each newly encountered identifier.

In Chapter 1, I give an account of the λ-calculus, for the sake of logical completeness of the thesis. Some of the formulae in this chapter were arrived at by the author independently, although other researchers must have derived them also. Chapter 2 sets forth the basic data structure and function defining machinery used in Chapters 3 and 4. The data structure definitions used are a modification of Peter Landin's structure definitions, and allow an arbitrary graph as a data structure. This has some advantages over the pure trees approach of the VDL group. Chapter 3 introduces my basic λ-calculus interpreter, the S.C.E. machine, and Chapter 4 extends its power and develops it into the I.S.C.E.M.O. machine. Chapter 5 explores one possible generalization of the I.S.C.E.M.O. machine, and considers areas for future research.

Chapter 1

The λ-calculus

1.1.

Traditional notation for functions uses an expression such as 'f(x)' ambiguously, both as the name of a function, as in

$$'f(x) = x+x^2',$$

and the value of a function at a point, as in

$$'f(x)+x^2+y'.$$

This can lead to genuine ambiguities of meaning, as in the following example.

If we define $$P(f(x)) = \begin{cases} \frac{f(x)-f(0)}{x} & x\neq 0 \\ f'(0) & x=0, \end{cases}$$

then it is not clear what we mean by $P(f(x+1))$.

It could mean either

1) $P(f(x+1))$ is the function $P(g)$, where g is the function $f(x+1)$,

or 2) $P(f(x+1))$ is the function $h(x+1)$, where h is the function $P(f)$.

As a concrete example, if we let $f(x) = x^2$, then 1) gives

$$g = f(x+1) = (x+1)^2,$$

$$\text{so } P(g) = \frac{(x+1)^2-(1)^2}{x} = x+2,$$

while 2) gives $$h = P(f) = \frac{x^2-0^2}{x} = x,$$

so $h(x+1) = x+1$,

- two different results. Without an unambiguous notation, we are unable to decide what $P(f(x+1))$ means. We need to put into symbols a sentence such as "the function of x that $x+x^2$ is". The appropriate unambiguous notation for this was supplied by Alonzo Church in C1, who represents the above by

$$f = \lambda x.x+x^2, \tag{1}$$

the f being the <u>name</u> of the function, and $\lambda x.x+x^2$ being the <u>denotation</u> of the function. It is this use of the Greek letter lambda ("λ") to introduce a function that gives the lambda-calculus its name. The above two meanings of $P(f(x+1))$ now become

1) $P(\lambda x.f(x+1))$

and 2) $\lambda x.P(f(x+1))$,

- two different expressions.

We may similarly use the notation for functions of two or more arguments. For example, the function definition usually written as:

$$g(x,y,z) = x^2+y^2+z^2-3axyz$$

would now be written as:

$$g = \lambda(x,y,z).x^2+y^2+z^2-3axyz. \tag{2}$$

The right-hand-side of any expression of the form (1) or (2) is called a <u>λ-expression</u>. The variables appearing between the 'λ' and the '.' are called the <u>bound variables</u> of the λ-expression. Thus x is the bound variable in (1), and x,y and z the bound variables in (2). The

expression after the '.' is called the λ-body (or just body) of the expression. Thus $x+x^2$ is the λ-body of (1) and $x^2+y^2+z^2-3axyz$ the λ-body of (2). Any occurrences of the bound variables in the λ-body are called bound occurrences. Any variables which are not bound variables occurring in the λ-body are called free variables, and any such occurrence is called a free occurrence. Thus (1) has no free variables, and a is a free variable of (2). Another piece of notation I shall use is:

$$S^{x_1,\ldots,x_r}_{N_1,\ldots,N_r} M$$

which stands for the formula which results by substitution of $N_1, \ldots, N_r$ for $x_1, \ldots, x_r$, respectively, throughout a formula M.

I now give the rules for manipulating λ-expressions. Returning to the old notation for an illustration, we know that the following two expressions both define the same function:

$$f(x) = x+x^2$$

and

$$f(y) = y+y^2.$$

Clearly, renaming the variable used as argument of the function does not alter the function defined. This gives us the first rule of the λ-calculus: "The bound variables of a λ-expression may be consistently replaced by any other variables". In symbols, if $x_1, \ldots, x_n$ are the bound variables of a λ-expression M, and the variables $y_1, \ldots, y_n$ are all distinct from each other (though not necessarily from the x's) then the λ-expressions M and $S^{x_1,\ldots,x_n}_{y_1,\ldots,y_n} M$ are equivalent. Thus $\lambda x.x+x^2$ and $\lambda a.a+a^2$ are equivalent λ-expressions. Likewise, $\lambda(x,y,z).x^2+y^2+z^2-3axyz$, $\lambda(p,q,r).p^2+q^2+r^2-3apqr$ and $\lambda(z,x,t).z^2+x^2+t^2-3azxt$, are all equivalent. I use the word "consistently"

in the rule to indicate that all occurrences of a particular bound variable in a λ-expression must be changed to the same new variable. This rule is called α-conversion, and we write

$$\lambda x.x+x^2 >_\alpha \lambda y.y+y^2$$

to indicate α-conversion from $\lambda x.x+x^2$ to $\lambda y.y+y^2$. Since, however, we have equally that

$$\lambda y.y+y^2 >_\alpha \lambda x.x+x^2,$$

we write this dual relationship as

$$\lambda x.x+x^2 =_\alpha \lambda y.y+y^2.$$

We must use this rule with care, however. Consider the λ-expression

$$\lambda x.a+x. \qquad (3)$$

This is equal (or α-equivalent) to

$$\lambda y.a+y,$$

and so on. If we try to change the x to an a, however, we get

$$\lambda a.a+a, \qquad (4)$$

which is a completely different λ-expression. The function denoted by (3) adds 'a' to its argument, whereas the function denoted by (4) doubles its argument. Clearly, what has gone wrong is that the a, which is free in (3), has been "captured" in (4), and is no longer free. We must therefore modify our λ-conversion rule so that it reads: "The bound variables of a λ-expression may be consistently replaced by any variables distinct from the free variables of the λ-expression". In

symbols, if $x_1, \ldots, x_r$ are the bound variables of a λ-expression M and the variables $y_1, \ldots, y_r$ are all distinct from each other and the free variables of M (though not necessarily distinct from the x's), then the λ-expressions M and $S^{x_1,\ldots,x_r}_{y_1,\ldots,y_r}M$ are equivalent.

Going back to the old notation for another illustration, if

$$f(x) = x+x^2, \tag{1}$$

then

$$f(10) = 10+10^2,$$

and

$$f(a) = a+a^2.$$

Since in the λ-calculus (1) is written as

$$f = \lambda x.x+x^2,$$

then $f(10) = (\lambda x.x+x^2)\ (10)$. This is cumbersome and so in the λ-calculus we write (fx) instead of f(x). We also drop any outermost pair of parenthesis for convenience. Thus we have that

$$(\lambda x.x+x^2)10 = 10+10^2.$$

Likewise,

$$(\lambda x.x+x^2)a = a+a^2.$$

This gives us the second rule of the λ-calculus: "The arguments of a λ-expression may be consistently substituted for the bound variables of that λ-expression throughout it", or, in symbols, $(\lambda(x_1,\ldots,x_r).M)N_1 \ldots N_r$ may be replaced by $S^{x_1,\ldots,x_r}_{N_1,\ldots,N_r}M$. This rule is called <u>β-reduction</u>, and

we write

$$(\lambda x.x+x^2)10 >_\beta \quad 10+10^2.$$

As with α-conversion, we must be careful how we use this rule, for we have:

$$(\lambda x.(\lambda a.a+x))a >_\beta \quad \lambda a.a+a, \tag{5}$$

and also:

$$\begin{aligned}(\lambda x.(\lambda a.a+x))a &>_\alpha \quad (\lambda x.(\lambda b.b+x))a \\ &>_\beta \lambda b.b+a,\end{aligned} \tag{6}$$

two different results. Since α-conversion leaves λ-expressions equivalent, expressions (5) and (6) should be the same. The trouble here is that in (5) the a substituted for x has been "captured" by the 'λa' in 'λa.a+x'. Hence we must amend our β-conversion rule to read: "The arguments of a λ-expression may be consistently substituted for the bound variables of that λ-expression, provided that no free variables of the arguments are identical to bound variables in the λ-body of the expression". In symbols, $(\lambda(x_1,\ldots,x_r).M)N_1 \ldots N_r$ may be replaced by $S^{x_1,\ldots,x_r}_{N_1,\ldots,N_r}M$, provided that the bound variables of M are distinct from the free variables of $N_1, \ldots, N_r$. If the bound variables of M do coincide with the free variables of $N_1, \ldots, N_r$, then a preliminary α-conversion of M is necessary to remove the conflicts. It should be noted that the bound variables of a function do not necessarily have to appear in the function body. As examples, $\lambda(x,y).x$ chooses the first of two arguments and throws away the second, and $\lambda(x,y).y$ chooses the second of two arguments and throws away the first. If, in the rule above, we replace the phrases "bound variables" and "arguments" by

"formal parameters" and "actual parameters" respectively, then the parallel with the parameter-passing mechanism of ALGOL 68 (and 60) should be obvious.

In the strict λ-calculus, that is, a fully formalized λ-calculus, we are not allowed argument lists. Thus the two examples shown above have to be written as $\lambda x \lambda y.x$ and $\lambda x \lambda y.y$ respectively.

The next section introduces the pure λ-calculus along more formal lines; "pure" λ-calculus means λ-calculus without arithmetic (such as I have been using here). Alternative ways of writing the λ-expression

$$(\lambda x.M)N$$

which I shall sometimes use are

$$\begin{array}{l} \underline{\text{let}}\ x = N \\ \underline{\text{in}}\ M, \end{array}$$

and

$$M\ \underline{\text{where}}\ x = N.$$

These forms (the "let-where" notation") are called "sugarings" of $(\lambda x.M)N$, because some people find them more "palatable" than the strict form. I also "sugar" $(\lambda(x_1,x_2,\ldots,x_r).M)N_1N_2 \ldots N_r$

as

$$\begin{array}{l} \underline{\text{let}}\ x_1 = N_1 \\ \underline{\text{and}}\ x_2 = N_2 \\ \vdots \\ \underline{\text{and}}\ x_r = N_r \\ \underline{\text{in}}\ M, \end{array}$$

or

$$\begin{array}{l} M\ \underline{\text{where}}\ x_1 = N_1 \\ \underline{\text{and}}\ x_2 = N_2 \\ \vdots \\ \underline{\text{and}}\ x_r = N_r\ . \end{array}$$

Let-where expressions may also properly be considered as concrete examples of an abstract object rather than a sugaring of the "correct" λ-expression form. That is, we can consider the expressions

$$(\lambda x.M)N, \quad \underline{\text{let}}\ x=N\ \underline{\text{in}}\ M$$

and

$$M\ \underline{\text{where}}\ x=N$$

as three different concrete representations of the same abstract object, namely an "AE" (for Applicative Expression). The AE $(\lambda x.M)N$ would have:

$$\text{rator}((\lambda x.M)N) = \lambda x.M,$$

$$\text{rand}((\lambda x.M)N) = N,$$

$$\text{bvlist}(\text{rator}((\lambda x.M)N)) = (x),$$

and

$$\text{body}(\text{rator}((\lambda x.M)N)) = M.$$

1.2. In this section I present the λ-calculus as a formal system.

The calculus has as primitive symbols the three symbols "(", ")" and "λ", and a denumerable set of symbols $x_0, x_1, x_2, \ldots$ each different from "(", ")" and "λ", called variables. I shall use lower case letters to represent variables.

A finite sequence of instances of primitive symbols is termed a formula. I shall use upper case letters to represent formulas. A formula X is said to be equal to a formula Y, written $X \equiv Y$, if X and Y are (different instances of) the same sequence of symbols. A formula N is a part of a formula M if N can be obtained from M by erasing zero or more symbols from either or both ends of M. Thus, in particular, any formula is a part of itself.

A formula is said to be a well-formed formula (abbreviated "wff") if it follows from a finite number of applications of the following three rules, in which I also take the opportunity of saying what is meant by an occurrence of a variable in a well-formed formula being free or bound in that formula:

1. Every variable x is a well-formed formula, and x occurs free in x. No other variable occurs free in x.

2. If X and Y are well-formed formulas, then (XY) is a well-formed formula, and an occurrence of a variable x in (XY) is free in (XY) if and only if it occurs free in at least one of X and Y.

3. If X is a well-formed formula, and x any variable, then (λxX) is a well-formed formula, and an occurrence of a variable y in (λxX) is free in (λxX) if and only if it occurs free in X and is not the same symbol as x.

Finally, an occurrence of a variable in a well-formed formula is said to be a bound occurrence if and only if it is not a free occurrence. Hence, all occurrences of a variable x in a wff (λxX) are bound-occurrences. All free occurences of x in X are said to be bound to the x following the symbol λ.

A variable is said to be free in a wff if there is at least one free occurrence of that variable in the wff, and likewise a variable is said to be bound in a wff if there is at least one bound occurrence of that variable in the wff. It should be noted that a variable may occur both free and bound in the same wff, for example, the first occurrence of x in $(x(\lambda xx))$ is free, while the second and third occurrences are bound.

A wff of the form (XY) is termed a <u>combination</u>, and a wff of the form (λxX) is termed a <u>λ-expression</u>.

From now on, it is to be understood that unless otherwise stated, any formula represented by an upper case letter will be a well-formed formula.

It may be noted that those formulas designated as being well-formed by the three rules above are the terminal productions of $\langle\lambda exp\rangle$ in the following grammar:

<λexp> → <variable>

<λexp> → (<λexp>< λexp>)

<λexp> → (λ<variable>< λexp>)

<variable> → $x_0 | x_1 | x_2 | \ldots$.

I now define what is meant by a sub-formula (abbreviated "subwff") of a formula. A formula N is a subformula of another formula M if it is a well-formed part of M. It should be noted that this implies that N cannot begin with a variable which immediately follows an occurrence of the symbol λ. A formula N is a proper subformula of a formula M if N is a subwff of M and N≠M.

For all variables x and all wffs N, I define the substitution operator S_N^x by the following three rules:

1. If y is a variable, then $S_N^x y = \begin{cases} N & \text{if } y \equiv x, \\ y & \text{otherwise,} \end{cases}$

2. $S_N^x(XY) = (S_N^x X \; S_N^x Y)$,

3. $S_N^x(\lambda y Y) = \begin{cases} (\lambda y Y) & \text{if } y \equiv x, \\ (\lambda y \; S_N^x Y) & \text{otherwise.} \end{cases}$

In order to avoid writing large numbers of brackets, I simplify wffs by eliding redundant brackets. Thus I write ((XY)Z) as (XYZ) (left association of combinations), (λx(MN)) as (λxMN), (λx(λyZ)) as (λxλyZ) (right association of λ's), and (X(λyY)) as (Xλy Y). Further, I drop the outermost pair of brackets, if any. This means that I eliminate as many brackets as possible, while keeping the original wff recoverable from the simplified one. Note, though, the words of J.B. Rosser in R3:

"Note that the parentheses are not supposed really to be absent. They are actually there, but for convenience we have failed to write them.". Finally, to aid the eye, if X does not begin with the symbol λ, I write λxX as λx.X. Thus I write

$$(\lambda x(x(\lambda yy))) \text{ as } \lambda x.x\lambda y.y,$$

$$(\lambda x((\lambda yy)x)) \text{ as } \lambda x.(\lambda y.y)x,$$

and $$(\lambda x(\lambda y(yx))) \text{ as } \lambda x\lambda y.yx.$$

All wffs are now terminal productions of <λexp> in the following grammar, which is complicated by my insistence on eliminating as many brackets as possible:

<λexp> → <λexp'>|<λexp">

<λexp'> → λ<var>< λexp'>|λ<var>.<λexp">

<λexp"> → <comb>|<var>

<comb> → <comb' >< atom >

<comb'> → <comb'>< atom'>|<atom">

<atom> → <λexp'>|(<comb>)|<var>

<atom'> → (<λexp'>)|(<comb>)|<var>

<atom"> → (<λexp'>)|<var>

<var> → $x_0|x_1|x_2|\ldots$.

In practice, I often insert extra brackets and/or periods for clarity, or to emphasise a particular subwff. I therefore give a (double-level) context-free grammar which has as its terminal productions all wffs parenthesised in all ways between least-parenthesised (simplified scheme) and fully-parenthesised (strict form). I believe this to be the first time a double-level grammar has been put to this use.

Metagrammar.

<NOTION> → <λexp'>|<comb>|<comb">

<NOTION pack> → <NOTION>|(<NOTION>)

Grammar.

<λexp> → <λexp' pack>|<comb pack>|<var>

<λexp'> → λ<var><.option><λexp>

<comb> → <comb'><atom>

<comb'> → <comb" pack>|<atom">

<comb"> → <comb'><atom'>

<atom> → <λexp' pack>|(<comb>)|<var>

<atom'> → (<λexp'>)|(<comb>)|<var>

<atom"> → (<λexp'>)|<var>

<.option> → .|

<var> → $x_0|x_1|x_2|\ldots$.

The fully-parenthesised form is obtained by choosing each <.option> to be empty and each <X pack> to be parenthesised, while the least-parenthesised form is obtained by choosing each <.option> before a <comb> or <var> to be a ".", and all the rest empty, and each <X pack> to be unparenthesised.

I now give the two rules by which one wff may be converted into another.

1) α-conversion (Renaming of bound variables).

We may replace any subwff $(\lambda x N)$ of a wff M by

$$(\lambda y\ S^x_y\ N),$$

provided that in so doing, no free occurrence of a (possibly renamed) variable in N becomes bound. If M is converted to M' by this process, we write

$$M >_\alpha M'.$$

As examples, $\lambda x\lambda y.yx >_\alpha \lambda x\lambda z.zx >_\alpha \lambda y\lambda z.zy >_\alpha \lambda y\lambda x.xy$, but $\lambda x\lambda y.yx >_\alpha \lambda y\lambda y.yy$ is illegal (the free occurrence of x in $\lambda y.yx$ becomes a bound occurrence of y in $\lambda y.yy$), as is $\lambda x.xy >_\alpha \lambda y.yy$ (the free occurrence of y in xy becomes bound).

It is clear that, by the same rule, we may always replace the subwff $(\lambda yN')$ of M', where $N' = S^x_y N$, by $(\lambda x\, S^y_x N')$, and so we also have

$$M' >_\alpha M.$$

We therefore say that $M =_\alpha M'$, or that M and M' are equivalent up to renaming of bound variables. Since $M >_\alpha M$ trivially, $=_\alpha$ is an equivalence relation.

2) <u>β-reduction</u> (substitution of arguments for bound variables).

We may replace any subwff $((\lambda xM)N)$ of a wff P by

$$S^x_N M,$$

provided that in so doing no free occurrence of a variable in N becomes bound. If P is converted to P' by this process,

we write

$$P >_\beta P'$$

or occasionally, $P' <_\beta P$,

when we say that P is obtained from P' by a β-abstraction.

As examples,

$$(\lambda x\lambda y.xy)\lambda z.z >_\beta \lambda y.(\lambda z.z)y >_\beta \lambda y.y,$$

and $$(\lambda x.x\lambda x\lambda y.y)\lambda x.xz >_\beta (\lambda x.xz)\lambda x\lambda y.y$$
$$>_\beta (\lambda x\lambda y.y)z >_\beta \lambda y.y,$$

but $(\lambda x\lambda y.xy)\lambda x.xy >_\beta \lambda y.(\lambda x.xy)y$ is illegal (the free occurrence of y in $\lambda x.xy$ becomes bound).

If, on performing the substitutions, one of the free variables of N does become bound, α-conversion must be used to rename the bound variables of M suitably. Thus while the last example above was impermissible, we may have

$$(\lambda x\lambda y.xy)\lambda x.xy >_\alpha (\lambda x\lambda t.xt)\lambda x.xy >_\beta \lambda t.(\lambda x.xy)t$$
$$>_\beta \lambda t.ty >_\alpha \lambda x.xy,$$

and the α-conversion required to convert $\lambda t.(\lambda x.xy)t$ into $\lambda y.(\lambda x.xy)y$ is not permissible. From now on, I take a β-reduction to include any preliminary α-conversions, if necessary.

Corrigendum

The section on page 24 beginning "I have never ..." and ending with "... becomes bound thereby.", should be deleted and replaced by the following.

The rules given here are the least restrictive possible: they forbid substitution only if a free variable becomes bound thereby. Some authors define S_N^x as substituting N for all occurrences of x in a wff, whereas my S_N^x "shields" a subwff beginning with "$(\lambda x$" from further substitution. This simplifies the statement of the restrictions in the α and β conversion rules. While this definition of S_N^x is sufficient for my purpose, it is a "naive" definition, in that it does not perform β substitutions correctly. For example:

$$S_y^x(\lambda y.x) = (\lambda y.y),$$

and $(\lambda y.x)y >_\beta (\lambda y.y)$ is an impermissible β-reduction. Curry and Feys in C2 give a "correct" definition of the substitution operator, which renames bound variables when necessary.

Rosenbloom, in R2, asserts that the statement of the α and β conversion rules in several standard works is incorrect, and cites Church [C1] as giving a correct statement. However, even Church's statement of them is more restrictive than that given here. ~~I have never seen any statement of the conversion rules that was not unnecessarily restrictive. Part of the trouble stems from the fact that some other authors give S_N^x as substituting N for all occurrences of x in a wff, whereas my S_N^x "shields" a subwff beginning with "$(\lambda x$" from further substitution. This simplifies the statement of the restrictions. Still other authors build my "shielding" property into a 'Subst(N,x,=)' function, but unfortunately they build some unnecessary restrictions into it as well. The rules given here are the least restrictive possible: they forbid substitution only if a free variable becomes bound thereby.~~

A reduction sequence of a wff A into a wff B is a finite sequence of wffs $E_0, E_1, E_2, \ldots, E_n$, where $n \geq 0$, such that

$$A \equiv E_0 >_\beta E_1 >_\beta E_2 >_\beta \ldots >_\beta E_n =_\alpha B. \qquad n \geq 0.$$

We write $A \geq_\beta B$, or $B \leq_\beta A$, if there exists a reduction sequence which transforms A into B.

Two wffs A and B are interconvertible, written $A =_\beta B$, if either $A \geq_\beta B$ or $B \geq_\beta A$, or there exists a wff C such that $A =_\beta C$ and $C =_\beta B$.

A wff is in normal form (abbreviated "nf") if it contains no subwff of the form $(\lambda x M)N$, i.e. if it can be β-reduced no further. A wff B is a normal form of a wff A, or A has a normal form B if B is in normal form, and $A \geq_\beta B$.

For example, $\lambda x.zu$ is a normal form of $(\lambda x\lambda u.xzu)\lambda x\lambda y.xu$

since $$(\lambda x\lambda u.xzu)\lambda x\lambda y.xu >_\alpha (\lambda x\lambda t.xzt)\lambda x\lambda y.xu$$
$$>_\beta \lambda t.(\lambda x\lambda y.xu)zt >_\beta \lambda t.(\lambda y.zu)t$$
$$>>_\beta \lambda t.zu >_\alpha \lambda x.zu.$$

Not all wffs have normal forms, for example:

$$(\lambda x.xx)(\lambda x.xx) >_\beta (\lambda x.xx)(\lambda x.xx) >_\beta \ldots$$

and $$(\lambda x.xxx)(\lambda x.xxx) >_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) >_\beta \ldots$$

It can be shown to be undecidable as to whether a wff has a normal form or not (see e.g. C1).

By far the most important theoretical result about the λ-calculus is the Church-Rosser theorem: If a wff A has two normal forms B_1 and B_2, then $B_1 =_\alpha B_2$. What Church and Rosser <u>actually</u> proved in their theorem is that if $A =_\beta B$ then there exists a wff Z such that $A \geq_\beta Z$ and $Z \leq_\beta B$. From this the previous theorem follows immediately.

The proof of this theorem is long and tedious, and occupies the whole of chapter 4 of C2.

This theorem means that if we reduce a wff to normal form by one choice of reductions, then any other choice of reductions which lead to a normal form will lead to (essentially) the <u>same</u> normal form.

This guarantees that no contradictions can arise if we assign meanings to λ-expressions. Since if we assign a meaning to a λ-expression then such a meaning must be invariant under α and β conversion, inter-convertible λ-expressions must be assigned the same meaning. That is,

we must assign meanings consistently. We can look on β-reduction as a deduction of one meaningful wff from another. If we now reduce a wff A with a given meaning to normal form B, then the Church-Rosser theorem guarantees that B is unique (up to renaming of bound variables), or we cannot, from a wff A, deduce two different wffs in normal form with the same meaning. Thus the Church-Rosser theorem guarantees the consistency of the λ-calculus.

It is important to realise that the Church-Rosser theorem does not assert that if a wff has a normal form then every reduction sequence will lead to it. A simple counterexample is:

$(\lambda x.a)((\lambda x.xx)(\lambda x.xx))$ $\quad$ $(\lambda x.a)((\lambda x.xx)(\lambda x.xx))$

$>_\beta$ a which is in nf. $\quad$ $>_\beta (\lambda x.a)((\lambda x.xx)(\lambda x.xx))$

$>_\beta \ldots$ ad. inf.

A left-most reduction sequence is one where at each β-reduction the left-most subwff of the form $(\lambda xM)N$ is reduced. It can be shown that if a wff is reduced by a left-most reduction sequence, then the sequence is finite if and only if the wff has a normal form [C2].

If X and Y are wffs, then $X =_\alpha Y$ implies $X \geq_\beta Y$ and $Y \geq_\beta X$ trivially but the converse is not true. For example

$$(\lambda x.xx)((\lambda x\lambda y.xyy)\lambda u.u) >_\beta (\lambda x\lambda y.xyy)(\lambda u.u)((\lambda x\lambda y.xyy)\lambda u.u)$$

and $(\lambda x\lambda y.xyy)(\lambda u.u)((\lambda x\lambda y.xyy)\lambda u.u) >_\beta (\lambda y(\lambda u.u)yy)((\lambda x\lambda y.xyy)\lambda u.u)$

$$>_\beta (\lambda y.yy)((\lambda x\lambda y.xyy)\lambda u.u)$$

$$>_\alpha (\lambda x.xx)((\lambda x\lambda y.xyy)\lambda u.u)$$

and clearly $(\lambda x.xx)((\lambda x\lambda y.xyy)\lambda u.u) \neq_\alpha (\lambda x\lambda y.xyy)(\lambda u.u)((\lambda x\lambda y.xyy)\lambda u.u)$.

Since, for all wffs N, $(\lambda x(Mx))N >_\beta (MN)$ provided that x does not occur free in M, some authors allow a η-conversion rule, defined as follows.

3) η-reduction

We may replace any subwff $(\lambda x(Nx))$ of a wff M by

N,

provided that x does not occur free in N. If M is converted to M' by this process, we write

$$M >_\eta M'.$$

Owing to the above-mentioned equivalence, this rule is not of much importance. In the sequel there are only a few occasions when we might be sorry that we have not included η-conversion. A λ-calculus admitting η-reductions is called a βη-calculus.

An important enlargement of the λ-calculus is the addition of an indefinite class of rules, called δ-rules. We first of all include among the primitive symbols of the calculus a denumerable set of symbols $a_0, a_1, a_2, \ldots$ each different from "(", ")", "λ" and each of the variables, called constants. We then augment the definition of a wff to include the constants, by adding to the previous three clauses:

1'. Every constant a is a well-formed formula, and no variable occurs free in a.

The δ-rules are all of the following form.

4) δ-reduction

We may replace any subwff N of a wff M by the wff N', provided that:

i) N is of the form $((aN_1)...N_r)$ where a is a constant,

ii) N and N' contain no free variables, and

iii) no proper subwff of N is reducible by the β- and η-rules, or any δ-rule, N itself is not reducible by any other δ-rule, and N' is not reducible by any δ-rule.

If M is converted to M' by this process, we write

$$M >_\delta M'.$$

A λ-calculus admitting δ-rules is called a βδ-calculus. If if also admits η-reductions, it is called a βηδ-calculus, or full λ-calculus.

A set of δ-rules is always associated with the constants appearing in a particular system. I give two examples of the use of δ-rules, the first modified from C1 and the second from M6.

a) Church [C1] studied a specialised form of δ-rules, as follows. Just one constant is admitted, the symbol "δ". The two δ-rules are: If M and N conform to restrictions i) - iii) above, then

$((\delta M)N)$ may be replaced by $(\lambda x(\lambda yx))$ if $M =_\alpha N$,

and $((\delta M)N)$ may be replaced by $(\lambda x(\lambda yy))$ if $M \neq_\alpha N$.

b) Morris [M6] gives a system for dealing with the natural numbers. There are four constants, namely "O", "suc", "pred" and "zero". We define a wff to be a numeral if it has the form $(\text{suc}^n\ \text{O})$, i.e. is one of O, (suc O), (suc(suc O)), etc.. The (infinite) set of δ-rules are then the following:

If x is a numeral, then

i) (pred O) may be replaced by O,

ii) (pred(suc x)) may be replaced by x,

iii) (zero O) may be replaced by $(\lambda x(\lambda yx))$, and

iv) (zero(suc x)) may be replaced by $(\lambda x(\lambda yy))$.

The reason for the choice of the λ-expressions $(\lambda x(\lambda yx))$ and $(\lambda x(\lambda yy))$ in these reduction rules will become apparent in the next section.

Analogously to the previous definitions of reduction sequence, interconvertibility and normal form (now all properly prefixed by "β-"), we can define η- and δ- reduction sequences, interconvertibility and normal form. We can also give corresponding $\beta\eta$-, $\beta\delta$-, $\eta\delta$- and $\beta\eta\delta$-definitions, all of which have their obvious, intuitive meanings. In general the reductions are all done in one fixed calculus, and so we may just talk of reduction sequences and normal form, and drop the qualifying prefixes. The Church-Rosser theorem retains its central position of importance, since it remains true for an arbitrary full λ-calculus.

I do not consider δ-rules further in the sequel, since for my purposes all of the constants required (the integers, for example) can be defined within the β-calculus itself

It is the purpose of the next section to show how this may be done.

1.3.

As already mentioned, we may assign meanings to λ-expressions. That is, we can use the λ-calculus to express other branches of mathematics within the λ-calculus itself, for instance, boolean algebra and recursive function theory. Given the purely syntactic structures of the λ-calculus, we are at liberty to interpret them in any way we like, and provided our assignments are consistent, we can say that these formulas mirror within the λ-calculus the system under consideration. The Church-Rosser theorem guarantees that provided our original assignments are consistent, no inconsistencies can result.

In this section, various sequences of characters will be used ambiguously. As examples: the strings "suc", "and", etc. will be used to represent a function and also to abbreviate a certain λ-expression; the characters "0", "1" etc. will be used to denote integers, and also to abbreviate a certain λ-expression, etc.. This convention saves the use of excessive ugly underlining, or the appearance of a plethora of 'quote' symbols.

The basic idea behind the representation of any formal system within the λ-calculus is that of λ-definability. A function F (over some domain of interest) is said to be <u>λ-definable</u> if there exists a λ-expression F' such that if x_1', x_2', ..., x_n' are λ-expressions representing the arguments $x_1, x_2, \ldots, x_n$ of F, then $F'x_1'x_2'\ldots x_n'$ has a normal form x' (which represents x) if $Fx_1x_2\ldots x_n$ is defined and equals x, and has no normal form if $Fx_1x_2\ldots x_n$ is undefined. Again, the Church-Rosser theorem guarantees that no inconsistencies can arise from such a system.

1.3.1.

I start with boolean algebra. The ALGOL 60 conditional statement suggests a way of representing the boolean constants true and false in the λ-calculus. if b then s1 else s2; means that if b evaluates to true, then statement s1 is executed, otherwise statement s2 is executed. If we write this as: if b,s1,s2; then we see that true chooses the first element of a two element list, and false the second. I have already shown λ-expressions which do just that, namely λxλy.x and λxλy.y. Hence, I make the formal definitions:

$$T = \lambda x \lambda y.x$$

and

$$F = \lambda x \lambda y.y$$

Therefore, if s1 and s2 are any λ-expressions, we have that $T\ s1\ s2 >_\beta s1$ and $F\ s1\ s2 >_\beta s2$.

Now the λ-expression for the function not is easily derived, since not T must reduce to F, and not F must reduce to T. Hence

$$not = \lambda x.xFT,$$

or

$$not = \lambda x.x(\lambda x \lambda y.y)\lambda x \lambda y.x.$$

As a check: $not\ T = (\lambda x.xFT)T >_\beta TFT >_\beta F$

and $not\ F = (\lambda x.xFT)F >_\beta FFT >_\beta T.$

The most convenient representation for the function or is the McCarthy conditional form: $(x \rightarrow T,y)$. Hence (or T y) must select T, and (or F y) must select y. Therefore, we have that

$$or = \lambda u \lambda v.uTv,$$

i.e.

$$or = \lambda u \lambda v.u(\lambda x \lambda y.x)v.$$

Similarly, and is $(x \to y,F)$, and hence is rendered as a λ-expression as follows:

$$\text{and} = \lambda u \lambda v.uvF$$

i.e. $$\text{and} = \lambda u \lambda v.uv\lambda x\lambda y.y.$$

Implies follows in the same way: $\text{imp} = (x \to y,T)$. Hence,

$$\text{imp} = \lambda u \lambda v.uvT$$

i.e. $$\text{imp} = \lambda u \lambda v.uv\lambda x\lambda y.x.$$

As one last example in this sub-section, I derive the function nand, by two different methods.

Method (i). We have: $\text{nand} = (x \to \sim y,T)$. Hence,

$$\begin{aligned} \text{nand} &= \lambda u \lambda v.u(\text{not } v)T \\ &= \lambda u \lambda v.u((\lambda x.xFT)v)T \\ &>_{\beta} \lambda u \lambda v.u(vFT)T \end{aligned}$$

or $$\text{nand} = \lambda u \lambda v.u(v(\lambda x\lambda y.y)\lambda x\lambda y.x)\lambda x\lambda y.x.$$

Method (ii). By definition, nand x y = not(and x y). Therefore,

$$\begin{aligned} \text{nand } x\ y &= (\lambda x.xFT)((\lambda u \lambda v.uvF)xy) \\ &>_{\beta} (\lambda x.xFT)(xyF) \\ &>_{\beta} (xyF)FT \end{aligned}$$

Hence, by β-abstraction twice,

$$\text{nand} = \lambda u \lambda v.uvFFT$$

or $$\text{nand} = \lambda u \lambda v.uv(\lambda x\lambda y.y)(\lambda x\lambda y.y)\lambda x\lambda y.x.$$

It should be noted that the two λ-expressions given for nand are not interconvertible, since both expressions are in normal form, and one cannot be changed into the other by α-conversion only. I could have reached the same result by noting that both or and and are commutative, so that I also have or = λuλv.v(λxλy.x)u and and = λuλv.vuλxλy.y.

Since I constructed my functions to take the specific representations I have chosen for T and F as arguments, I can say nothing about the behaviour of not, or, and, imp etc. when applied to λ-expressions other than T or F.

My representation of true and false is not the only one possible, by any means. As already stated, we are at liberty to assign to any λ-expressions the meanings true and false, provided that such representations remain consistent. We may represent true by λxλy.y and false by λxλy.x, or represent true by λxλy.xy and false by λxλy.y. Church, in C1, represented true by λxλy.x(xy) and false by λxλy.xy. One can also investigate three-valued logics, perhaps representing true by λxλyλz.x, undefined by λxλyλz.y and false by λxλyλz.z.

1.3.2.

I turn now to functions of the non-negative integers. If I could find a representation of zero and the successor function suc in the λ-calculus, then suc 0 would represent 1, suc(suc0) would represent 2, and so on. But,

$$(\lambda f\lambda x.fx)\,\mathrm{suc}\ 0 >_\beta \mathrm{suc}\ 0 = 1,$$

$$(\lambda f\lambda x.f(fx))\,\mathrm{suc}\ 0 >_\beta \mathrm{suc}(\mathrm{suc}\ 0) = 2,$$

and so on. Also,

$$(\lambda f\lambda x.x)\,\mathrm{suc}\ 0 >_\beta 0.$$

This suggests representing zero as $\lambda f\lambda x.x$, one as $\lambda f\lambda x.fx$, two as $\lambda f\lambda x.f(fx)$, and so on. Accordingly, I make the definitions:

$$0 = \lambda f\lambda x.x,$$

$$n = \lambda f\lambda x.f^n x, \quad \text{where} \quad f^1 x = fx \quad \text{and} \quad f^{n+1} = f(f^n x), \quad n \geq 1.$$

It should be noted that here $f^n x$ means right association $f(f(\ldots(fx)))$.

I now find a λ-expression to represent suc. suc must be of the form $\lambda u\ldots.u\ldots$, and suc applied to n will therefore yield $\ldots(\lambda f\lambda x.f^n x)\ldots.$ To introduce an extra f, I need to remove the '$\lambda f\lambda x$' in n. This can be done as follows:

$$\mathrm{suc} = \lambda u\ldots.ufx\ldots$$

Therefore,

$$\mathrm{suc}\ n >_\beta \ldots(\lambda f\lambda x.f^n x)fx\ldots >_\beta \ldots f^n x\ldots$$

Now I can apply an extra f, and reintroduce the '$\lambda f\lambda x$'. So,

$$\mathrm{suc} = \lambda u\lambda f\lambda x.f(ufx).$$

As a final check, for all $n \geq 0$,

$$\begin{aligned}\mathrm{suc}\ n &= (\lambda u\lambda f\lambda x.f(ufx))\lambda f\lambda x.f^n x \\ &>_\beta \lambda f\lambda x.f((\lambda f\lambda x.f^n x)fx) \\ &>_\beta \lambda f\lambda x.f(f^n x) = \lambda f\lambda x.f^{n+1}x = n+1.\end{aligned}$$

It should be noted that I can introduce the extra f in another way, by substituting (fx) for x, giving a different version of suc:

$$\text{suc} = \lambda u \lambda f \lambda x.uf(fx).$$

As a check, for any $n \geq 0$,

$$\begin{aligned} \text{suc } n &= (\lambda u \lambda f \lambda x.uf(fx))\lambda f \lambda x.f^n x \\ &>_\beta \lambda f \lambda x.(\lambda f \lambda x.f^n x)f(fx) \\ &>_\beta \lambda f \lambda x.f^n(fx) = \lambda f \lambda x.f^{n+1}x = n+1. \end{aligned}$$

Hence, I have two easily-derivable but not interconvertible versions of suc (as I had for nand).

Now I find a λ-expression for the function zero, which yields <u>true</u> if applied to the number zero, and <u>false</u> if applied to any non-zero positive integer. To start with,

$$\text{zero} = \lambda u....u....$$

Now, $0 = \lambda f \lambda x.x$, and is identical in action to F, in choosing the second of two arguments. Therefore,

$$\text{zero} = \lambda u.uAT,$$

where A is some λ-expression still to be determined. zero applied to $n+1$ $(n \geq 0)$ gives us:

$$\begin{aligned} (\lambda u.uAT)n+1 &>_\beta (\lambda f \lambda x.f^{n+1}x)AT \\ &>_\beta A^{n+1}\ T = A(A^n T). \end{aligned}$$

Hence, I require A to be that function which throws away its argument and yields F regardless, i.e. $\lambda x.F$. Therefore,

$$\text{zero} = \lambda u.u(\lambda x.f)T,$$

i.e. $$\text{zero} = \lambda u.u(\lambda x \lambda x \lambda y.y)\lambda x \lambda y.x.$$

To construct a λ-expression for the function add, I note that we must have add = λuλv....u...v.... Therefore, if I apply add to n and m, I get:

$$\ldots(\lambda f\lambda x.f^n x)\ldots(\lambda f\lambda x.f^m x)\ldots$$

I would like to strip the 'λfλx' from the second term, and substitute the resulting $f^m x$ instead of x in the first term. Hence,

$$\text{add} = \lambda u\lambda v\lambda f\lambda x.uf(vfx).$$

As an example: $\text{add } n\ m >_\beta \lambda f\lambda x.(\lambda f\lambda x.f^n x)f((\lambda f\lambda x.f^m x)fx)$

$$>_\beta \lambda f\lambda x.(\lambda x.f^n x)(f^m x)$$

$$>_\beta \lambda f\lambda x.f^n(f^m x) = \lambda f\lambda x.f^{n+m}x = n+m.$$

Another representation of add can be found by noting that add n m can be defined as n applications of suc to m. Hence,

$$\text{add} = \lambda u\lambda v.u\ \text{suc}\ v,$$

and also,

$$\text{add} = \lambda u\lambda v.u\ \text{suc}(v\ \text{suc}\ 0).$$

Since add is commutative, this gives me a large number of different versions of add, when taken with the two versions of suc.

However, I do not have quite the freedom I appear to have at first sight. While there is nothing wrong with choosing any one of the large number of versions of add, and either of the versions of suc, I must take note of the fact that add 1 is the same function as suc. Consequently, while it is not necessary, it is somewhat pleasing to have add $1 =_\alpha$ suc.

If I choose add = $\lambda u\lambda v\lambda f\lambda x.uf(vfx)$, then

$$\text{add } 1 = (\lambda u\lambda v\lambda f\lambda x.uf(vfx))\lambda f\lambda x.fx$$

$$>_\beta \lambda u\lambda f\lambda x.f(ufx), \text{ the first verion of suc;}$$

if I choose add = $\lambda u\lambda v\lambda f\lambda x.vf(ufx)$, then

$$\text{add } 1 = (\lambda u\lambda v\lambda f\lambda x.vf(ufx))\lambda f\lambda x.fx$$

$$>_\beta \lambda u\lambda f\lambda x.uf(fx), \text{ the second version of suc.}$$

From now on, I take suc = $\lambda u\lambda f\lambda x.f(ufx)$, and

$$\text{add} = \lambda u\lambda v\lambda f\lambda x.uf(vfx).$$

Defining the function mult is a little more difficult. I would like to arrange it so that mult n m substitutes the f^m from m for every f in n. The following expression does just that:

$$\text{mult} = \lambda u\lambda v.u(vf).$$

I can also define mult n m as n applications of add m to zero. Hence, I also have:

$$\text{mult} = \lambda u\lambda v.u(\text{add } v)0.$$

Also, the function exp is λ-defined by:

$$\text{exp} = \lambda u\lambda v.vu$$

It should be noted that with this definition, n^0 is undefined for any n. That is, the normal form of exp n 0 does not represent an integer.

In fact, exp n 0 $>_\beta \lambda x.x$. This is one of the cases when I might be sorry that I have not included η-reductions, since

$$1 = \lambda f.(\lambda x.fx) >_{\eta} \lambda f.(f) =_{\alpha} \lambda x.x,$$

and hence

$$\exp\ n\ 0 =_{\beta\eta} 1.$$

However, exp n m is m applications of mult n to 1, so I also have:

$$\exp = \lambda u \lambda v.v(\text{mult } u)1.$$

With this definition of exp, $n^0 = 1$ for all n.

Having found a λ-expression to represent suc, I should like to find one to represent pred, the predecessor function. Introducing an extra f into $f^n x$ by suc was easy, but how am I to delete one f? Two possible methods come to mind. First, substitute for f some λ-expression of the form $\lambda y.yX$. Then fx reduces to xX. I therefore arrange to substitute for x some λ-expression which destroys X on a first application, and from then on adds one f on each application. Secondly, I could substitute some form of a counter for x, and allow f to increment this counter on each application except the first. I derive an expression for pred by both methods.

Method (i). I hope that pred = $\lambda u \lambda f \lambda x.uAB$, for some A and B. Therefore,

$$\text{pred } n >_{\beta} (\lambda f \lambda x.f^n x)AB$$
$$>_{\beta} A^n B.$$

Therefore, I must have $A = \lambda y.yA'$, so

$$AB >_{\beta} BA'$$

and $$A(AB) >_{\beta} A(BA') >_{\beta} BA'A', \text{ etc..}$$

Hence, I must have $BA' >_\beta Xx$, and $XxA' >_\beta X(fx)$,

$X(fx)A' >_\beta X(f(fx))$, etc..

Therefore, if $B = \lambda y\lambda z.zx$, $BA' >_\beta \lambda z.zx$,

and so $BA'A' >_\beta A'x$,

and I require $A'xA' >_\beta A'(fx)$.

Hence,

$$A' = \lambda y\lambda z.z(fy).$$

Tidying up the details after some experimentation, I have finally:

$$pred = \lambda u\lambda f\lambda x.u(\lambda y.y\lambda y\lambda z.z(fy))(\lambda y\lambda z.zx)\lambda y.y.$$

As a test, $pred\ n >_\beta \lambda f\lambda x.(\lambda y.y\lambda y\lambda z.z(fy))^n(\lambda y\lambda z.zx)\lambda y.y$

Now, $(\lambda y.y\lambda y\lambda z.z(fy))\lambda y\lambda z.zx$

$>_\beta (\lambda y\lambda z.zx)\lambda y\lambda z.z(fy)$

$>_\beta \lambda z.zx,$

and $(\lambda y.y\lambda y\lambda z.z(fy))\lambda z.zx$

$>_\beta (\lambda z.zx)\lambda y\lambda z.z(fy)$

$>_\beta (\lambda y\lambda z.z(fy))x$

$>_\beta \lambda z.z(fx),$

and in general, $(\lambda y.y\lambda y\lambda z.z(fy))\lambda z.z(f^n x)$

$>_\beta (\lambda z.z(f^n x))\lambda y\lambda z.z(fy)$

$>_\beta (\lambda y\lambda z.z(fy))f^n x$

$>_\beta \lambda z.z(f^{n+1}x)$

Hence, for all $n \geq 0$,

$$\begin{aligned} \text{pred } n+1 &>_\beta \lambda f\lambda x.(\lambda z.z(f^n x))\lambda y.y \\ &>_\beta \lambda f\lambda x.(\lambda y.y)f^n x \\ &>_\beta \lambda f\lambda x.f^n x = n \end{aligned}$$

It should be noted that pred 0 is undefined, since the normal form for it is not the representation of an integer (but it is η-reducible to 0).

Method (ii). I proceed as follows: I use $\langle a,b\rangle$ to represent $\lambda x.xab$. Then, $\langle a,b\rangle T >_\beta a$, and $\langle a,b\rangle F >_\beta b$. (I here use T and F rather than the better τ_2^1 and τ_2^2). Now, if A is any λ-expression whatever,

$$\begin{aligned} &(\lambda x.\langle \text{suc}(xT),(xT)\rangle)\langle n,A\rangle \\ &>_\beta \langle \text{suc}(\langle n,A\rangle T),(\langle n,A\rangle T)\rangle \\ &>_\beta \langle \text{suc } n,n\rangle \\ &>_\beta \langle n+1,n\rangle, \end{aligned}$$

i.e., for any A,

$$\begin{aligned} &(\lambda x.\langle \text{suc}(xT),(xT)\rangle)\langle n,A\rangle \\ &>_\beta \langle n+1,n\rangle \end{aligned}$$

Hence,

$$\begin{aligned} &(\lambda x.\langle \text{suc}(xT),(xT)\rangle)^n\langle 0,D\rangle \\ &>_\beta (\lambda x.\langle \text{suc}(xT),(xT)\rangle)^{n-1}\langle 1,0\rangle >_\beta \ \ldots \\ &>_\beta \langle n,n-1\rangle, \text{ where D is a dummy } \lambda\text{-expression.} \end{aligned}$$

Therefore,

$$\text{pred} = \lambda u.(u\lambda x.\langle \text{suc}(xT),(xT)\rangle\langle 0,D\rangle)F.$$

Also, pred $0 >_\beta D$, so therefore, if I require pred $0 = 0$, I may choose $D = 0$, if I require pred 0 to be undefined, I may choose $D = (\lambda h.hh)(\lambda h.hh)$, which has no normal form.

From now on, any mention of pred will mean that version of pred such that pred $0 = 0$.

Having defined pred, I can easily define the modified difference function mod-dif, such that mod-dif $n\ m = n-m$ if $n \geq m$ and mod-dif $n\ m = 0$ if $n<m$, as follows:

$$\text{mod-dif} = \lambda u \lambda v.v\ \text{pred}\ u.$$

I can define abs-dif, the absolute difference function, as follows:

$$\text{abs-dif} = \lambda u \lambda v.\text{add}(\text{mod-dif}\ u\ v)\ (\text{mod-dif}\ v\ u).$$

There is still one elementary function left to derive which I shall need in 1.5., namely equal, which is such that equal $n\ m =$ <u>true</u> if $n=m$ and equal $nm =$ <u>false</u> if $n \neq m$. One idea for defining equal is as follows: arrange for the f's in n and m to build up λ-expressions which will annihilate each other when applied to each other. This is done as follows:

$$\text{equal} = \lambda u \lambda v.u(\lambda x \lambda y.yx)A(v(\lambda x \lambda y.yx)B)$$

Then,

$$\text{equal}\ n\ m >_\beta (\lambda x \lambda y.yx)^n A((\lambda x \lambda y.yx)^m B),$$

and

$$\begin{aligned}(\lambda y.yA')(\lambda y.yB') &>_\beta (\lambda y.yB')A' \\ &>_\beta A'B' = (\lambda y.yA'')(\lambda y.yB'') \\ &>_\beta \cdots\end{aligned}$$

and so on.

We see that three cases arise:

(i) if n=m then equal n m $>_\beta$ AB

(ii) if n<m then equal n m $>_\beta$ A(λy.yB')

(iii) if n>m then equal n m $>_\beta$ (λy.yA')B $>_\beta$ BA'

The third case demands that B = λx.F, the second that A = λx.F or λx.x(λx.F). However, this reduces case (i) to (λx.F)(λx.F) $>_\beta$ F, which is not what I want. This leads one to consider my original expression for equal followed by λ-expressions C,D,E,.... Then I have:

(i) if n=m then equal n m $>_\beta$ ABCDE...

(ii) if n<m then equal n m $>_\beta$ A(λy.yB')CDE...

(iii) if n>m then equal n m $>_\beta$ BA'CDE...

Case (ii) requires A = λx.x, so that (i) and (ii) reduce as follows:

(i) equal n m $>_\beta$ BCDE...

(ii) equal n m $>_\beta$ CB'DE...

Hence, by (ii), C = λx....F, where I do not yet know how many x's will be needed. If I try to choose B = λxλy.y, to throw away the A' in (iii), then I have:

(i) $>_\beta$ DE...

(ii) $>_\beta$ CB'DE...

and

(iii) $>_\beta$ CDE...

Now, the C in (ii) has to throw away one more λ-expression than the C in (iii), which is not possible. Hence, I must choose B = λxλyλz.z. Then,

(i) $>_\beta$ E...

(ii) $>_\beta$ CB'DE...

(iii) $>_\beta$ DE...

Hence,I choose E = T, C = λxλxλx.F, D = λx.F, and I am done. Finally:

equal = λuλv.u(λxλy.yx)(λx.x)(v(λxλy.yx)(λxλyλz.z))
(λxλxλx.F)(λx.F)T.

It is clear that I can define the other five relational operators in much the same way as follows:

For greater, A = λx.F, B = λx.T.

greater = λuλv.u(λxλy.yx)(λx.F)(v(λxλy.yx)(λx.T)).

For lesseq (less than or equal), A = λx.T, B = λx.F.

lesseq = λuλv.u(λxλy.yx)(λx.T)(v(λxλy.yx)(λx.F)).

For neq (not equal), A = λx.x, B = λxλyλz.z, C = λxλxλx.T, D = λx.T, E = F.

neq = λuλv.u(λxλy.yx)(λx.x)(v(λxλy.yx)(λxλyλz.z))
(λxλxλx.T)(λx.T)F.

For less, A = λx.x, B = λxλyλz.z, C = λxλxλx.T, D = λx.x, E = F.

less = λuλv.u(λxλy.yx)(λx.x)(v(λxλy.yx)(λxλyλz.z))
(λxλxλx.T)(λx.x)F.

For greq (greater than or equal), $A = \lambda x.x$, $B = \lambda x\lambda y\lambda z.z$, $C = \lambda x\lambda x\lambda x.F$, $D = \lambda x.x$, $E = T$.

$$greq = \lambda u\lambda v.u(\lambda x\lambda y.yx)(\lambda x.x)(v(\lambda x\lambda y.yx)(\lambda x\lambda y\lambda z.z))(\lambda x\lambda x\lambda x.F)(\lambda x.x)T.$$

As far as I know, this analysis of the relational operators is completely original.

As with boolean algebra, one can choose other representations of the integers. For example, I can define

$$0 = \lambda f\lambda x.x$$

and $n = \lambda f\lambda x.xf^n$, where $xf^1 = xf$, and $xf^{n+1} = (xf^n)f$, $n \geq 1$

It should be noted that here xf^n means left association $(((xf)f)\ldots)f$, contrary to usual convention. In this representation, I have

$$suc = \lambda u\lambda f\lambda x.ufxf$$

and also $$suc = \lambda u\lambda f\lambda x.uf(xf).$$

pred is considerably simplified, since

$$pred = \lambda u\lambda f\lambda x.uf\lambda y.x,$$

but mult and exp are made more complex, since one can no longer define mult as $\lambda u\lambda v.u(add\ v)0$ and exp as $\lambda u\lambda v.v(mult\ u)1$.

As an exercise, I translate from one notation to the other:

$$(\lambda u\lambda f.u\lambda y.yf)\lambda f\lambda x.f^n x >_\beta \lambda f\lambda x.xf^n$$

and $$(\lambda u\lambda f\lambda x.u(\lambda y\lambda z.z(fy))(\lambda y.yx)\lambda y.y)\lambda f\lambda x.xf^n >_\beta \lambda f\lambda x.f^n x,$$

for all $n \geq 0$.

I shall give another representation of the integers in the next sub-section.

1.3.3.

A very important use to which any formalism may be put is the representation of complex data-structures. Here I show that one of the simplest data-structures, namely the list, is λ-definable. It will be clear that much the same method may be applied to any data-structure representable as a tree.

A first idea for representing the list $\langle a_1,a_2,\ldots,a_n\rangle$ in the λ-calculus is as follows:

$$\langle a_1,a_2,\ldots,a_n\rangle = a_1a_2\ldots a_n.$$

However, this representation has two deficiencies. First, a_1 may not be in normal form, and hence $a_1a_2\ldots a_n$ will be reducible, thereby destroying the list. Secondly, even if a_1 could be guaranteed to be in normal form, the entire expression $a_1a_2\ldots a_n$ acts as a single entity, so that any attempt to throw away a_1 results in throwing away the entire list.

A second attempt at representing lists is:

$$\langle a_1,a_2,\ldots,a_n\rangle = a_1(a_2(\ldots(a_{n-1}a_n)\ldots)).$$

However, this suffers from the same deficiencies as the first method, only more so, since each a_i is free to cause reduction.

In order not to throw away the entire list on throwing away a_1, I must substitute the selector into a combination, something applied to the list. I can do this by my next attempt:

$$\langle a_1, a_2, \ldots, a_n \rangle = \lambda x.xa_1a_2\ldots a_n.$$

This is all right as a tuple, and indeed I have already used 2-tuples in defining pred. Here, the problem of a_1 not being in normal form no longer arises since a_1 is not being applied to anything. However, the list selectors would need to be of the form $\lambda x_1\lambda x_2\ldots\lambda x_n.x_i$, so that all the unwanted elements of the list would be thrown away. Since the selectors must be of known length, they cannot be applied to lists of arbitrary length. Hence, I must reject this as a method of representing variable-length lists. However, I use this as a representation of a tuple, and denote the tuple selectors $\lambda x_1\lambda x_2\ldots\lambda x_n.x_i$ by τ_n^i. It should be noted that the tuple must be applied to the selector, and not vice-versa.

My problem seems to be solved by starting with $\lambda x.xa_1\ldots$ as for tuples. My final, and successful, attempt to represent lists in the λ-calculus is as follows:

$$\langle a_1, a_2, \ldots, a_n \rangle = \lambda x.xa_1\langle a_2, \ldots, a_n \rangle,$$

and $\langle a_n \rangle = \lambda x.xa_nU$, where U is a dummy function,

which I shall here take to be $\lambda x\lambda y.yU'$, where $U' = (\lambda h.hh)(\lambda h.hh)$, for reasons which will emerge later. It should be noted that I can select other forms for U, as I find convenient.

The selectors are readily seen to be

$$h = \lambda x\lambda y.x$$

$$t = \lambda x\lambda y.y$$

$$ht = \lambda x\lambda y.y\lambda x\lambda y.x$$

$$ht^{n+1} = \lambda x\lambda y.yht^{n}$$

$$hh = \lambda x\lambda y.x\lambda x\lambda y.x,$$

and so on.

N.B. ht^n is an abbreviation for the appropriate selector, and does NOT mean $h(t(t...(tt)))$.

Again, the list must be applied to the selectors and not vice-versa. I use the notation $\langle a_1,a_2,...,a_n\rangle$ to represent both lists and tuples; context, in the form of which type of selector is used, will make it clear which meaning is intended.

If Q represents any selector, one can "manufacture" the selectors Qh and Qt by means of

$$\text{cons-h} = \lambda u\lambda x\lambda y.xu$$

and

$$\text{cons-t} = \lambda u\lambda x\lambda y.yu,$$

respectively.

I now try applying $\langle a_1,a_2\rangle$ to ht^{n+2}. I get:

$$\langle a_1,a_2\rangle ht^{n+2} >_\beta ht^{n+2}a_1\langle a_2\rangle = (\lambda x\lambda y.yht^{n+1})a_1\langle a_2\rangle$$

$$>_\beta \langle a_2\rangle ht^{n+1}$$

$$>_\beta ht^{n+1}a_2U = (\lambda x\lambda y.yht^{n})a_2U$$

$$>_\beta Uht^{n} = (\lambda x\lambda y.yU')ht^{n}$$

Hence,

$$\langle a_1,a_2\rangle ht^{n+2} >_\beta Uht^n = (\lambda x\lambda y.yU')ht^n >_\beta \lambda y.y\ U',$$

which is undefined, since U' has no normal form. Here U acts as the list terminator nil, and it is clear that any undefined list selection will result in $\lambda y.yU'$.

As an example of list manipulation in the λ-calculus, I try to define a tupling operator T_2, such that

$$\langle\langle a\rangle,\langle b\rangle,\ldots\rangle T_2 = \langle\langle a,b\rangle,\ldots\rangle.$$

I must have that $\langle\langle a\rangle,\langle b\rangle,\ldots\rangle T_2 >_\beta T_2\langle a\ \langle b\rangle,\ldots\rangle$.
Now I must "unlist" $\langle a\rangle$ by applying it to h, at the same time applying $\langle\langle b\rangle,\ldots\rangle$ to something similar to T_2. I must take care not to apply a to anything in the meantime. This is best done by attaching a onto the end of the expression out of harm's way. Hence, I put

$$T_2 = \lambda u\lambda v.vX(uh)$$

so therefore,

$$\begin{aligned} T_2\langle a\rangle\langle b\rangle,\ldots\rangle &>_\beta \langle\langle b\rangle,\ldots\rangle X(\langle a\rangle h) \\ &>_\beta X\langle b\rangle\langle\ldots\rangle a. \end{aligned}$$

X clearly has much the same form:

$$X = \lambda u\lambda v.Yv(uh)$$

so
$$\begin{aligned} X\langle b\rangle\langle\ldots\rangle a &>_\beta Y\langle\ldots\rangle(\langle b\rangle h)a \\ &>_\beta Y\langle\ldots\rangle ba. \end{aligned}$$

Hence, putting:

$$Y = \lambda u\lambda v\lambda w\lambda x.x(\lambda x.xw\lambda x.xvU)u$$

completes T_2, since

$$(\lambda u\lambda v\lambda w\lambda x.x(\lambda x.xw\lambda x.xvU)u)<...>ba$$

$$>_\beta \lambda x.x(\lambda x.xa\lambda x.xbU)<...>$$

$$= \lambda x,x<a,b><...>$$

$$= <<a,b>,<...>>.$$

Finally, bringing all the formulae together,

$$T_2 = \lambda u\lambda v.vX(uh)$$

$$X = \lambda u\lambda v.Yv(uh)$$

$$Y = \lambda u\lambda v\lambda w\lambda x.x(\lambda x.xw\lambda x.xvU)u$$

Clearly I can construct T_3 such that

$$<<a>,<b>,<c>,<...>>T_3 = <<a,b,c>,<...>>$$

by analogy with T_2:

$$T_3 = \lambda u\lambda v.vX_1(uh)$$

$$X_1 = \lambda u\lambda v.vX_2(uh)$$

$$X_2 = \lambda u\lambda v.Yv(uh)$$

$$Y = \lambda u\lambda v\lambda w\lambda t\lambda x.x(\lambda x.xt\lambda x.xw\lambda x.xvU)u$$

and similarly for more complicated operators.

Since it is more natural to apply operators to lists rather than lists to operators, I would like to alter h, t, T_2, etc., so that I can apply them to lists rather than vice-versa. I can do this by prefixing each operator by "λu.u". Thus h becomes λu.uλxλy.x, t becomes λu.uλxλy.y, ht becomes λu.uλxλy.yλxλy.x, and so on. I can also do this for the tuple selectors τ_n^i. Thus I shall write the selectors either before or after lists and tuples, as I find most convenient. More complicated list operators, like adjoin, which splices two lists together, are more easily defined using the McCarthy conditional of 1.5.. Two list operators which will be needed in 1.5. are prefix and null, which are defined as follows:

$$\text{prefix a } \langle \ldots \rangle = \langle a, \ldots \rangle,$$

and null applied to nil yields true and if applied to any other list yields false. We see that

$$\text{prefix} = \lambda u \lambda v \lambda x.xuv,$$

and null is λ-defined as:

$$\text{null} = \lambda u.u(\lambda x \lambda x \lambda x.F)\ x.T,$$

since
$$\begin{aligned} \text{null}\langle a,\ldots\rangle &>_\beta \langle a,\ldots\rangle(\lambda x\lambda x\lambda x.F)\lambda x.T \\ &>_\beta (\lambda x\lambda x\lambda x.F)a\langle\ldots\rangle\lambda x.T \\ &>_\beta F, \end{aligned}$$

and
$$\begin{aligned} \text{null U} &>_\beta U(\lambda x\lambda x\lambda x.F)\lambda x.T \\ &= (\lambda x\lambda y.yU')(\lambda x\lambda x\lambda x.F)\lambda x.T \\ &>_\beta (\lambda x.T)U' >_\beta T. \end{aligned}$$

Hence the reason for the appearance of a "λy.y" in the definition of U.

It is customary in defining prefix to take it not as a dyadic function, but as a monadic function which when presented with an argument forms the function which prefixes that argument to a list. Thus

$$\text{prefix } a = \lambda u \lambda x.xau$$

I can also define unitlist, the function which takes a single argument and makes it into a list, as:

$$\text{unitlist } a = \text{prefix } a\ U,$$

i.e. $$\text{unitlist} = \lambda u \lambda x.xuU.$$

In I.5. such useful list operators as adjoin and zip will be defined.

Now follows the promised representation of the integers. One can use the list selectors as a representation of the integers as follows:

$$0 = h$$

and $$n = ht^n \quad \text{for all} \quad n \geq 1$$

suc has already been given, as cons-t:

$$\text{suc} = \lambda u \lambda x \lambda y.yu.$$

pred is easily defined as $\lambda x.xD\lambda x.x$, where one may choose $D = 0$ for $\text{pred}0 >_\beta 0$, and $D = (\lambda h.hh)(\lambda h.hh)$ for pred 0 undefined.

1.4.

In this section I investigate the nature of the recursive definition of functions. The discussion here appears essentially in Morris[M6], but was prepared completely independently of that source.

My standard example in this section will be the well-known definition of the factorial function:

$$f(n) = (n=0 \rightarrow 1, n \times f(n-1)),$$

which, by the results of the previous section, could be written as

$$f = \lambda n.\text{zero } n\ 1(\text{mult } n(f(\text{pred } n))).$$

In this section I shall write my functions in the informal McCarthy conditional form. It will always be possible to translate them into the pure λ-calculus form, however.

Does the above definition define any function at all? We know that it does, since the recursion will eventually "unwind", by taking the "1" option when n is zero. However, what function does the following definition define?

$$f(n) = (n=0 \rightarrow 1, f(n))$$

ANY function such that $f(0) = 1$ seems to fit this definition, but this needs to be made precise.

I consider the evaluation of the following expression:

$$\begin{array}{l} \underline{\text{let}}\ f(x) = 2 \times x \\ \quad \underline{\text{in}}\ \underline{\text{let}}\ f(n) = (n=0 \rightarrow 1, n \times f(n-1) \\ \quad\quad \underline{\text{in}}\ f5 \end{array}$$

I translate this into the λ-calculus:

$$\underline{\text{let}}\ f = \lambda x.2\times x$$
$$\underline{\text{in}}\ \underline{\text{let}}\ f = \lambda n.(n=0\to 1, n\times f(n-1))$$
$$\underline{\text{in}}\ f5$$

and so to:

$$(\lambda f.(\lambda f.f5)\lambda n.(n=0\to 1, n\times f(n-1)))\lambda x.2\times x$$

Which is evaluated as follows:

$$\begin{aligned}
&(\lambda f.(\lambda f.f5)\lambda n.(n=0\to 1, n\times f(n-1)))\lambda x.2\times x \\
&>_\beta (\lambda f.f5)\lambda n.(n=0\to 1, n\times(\lambda x.2\times x)(n-1)) \\
&>_\beta (\lambda n.(n=0\to 1, n\times(\lambda x.2\times x)(n-1)))5 \\
&>_\beta (5=0\to 1, 5\times(\lambda x.2\times x)(5-1)) \\
&>_\beta 5\times(\lambda x.2\times x)4 \\
&>_\beta 5\times(2\times 4) = 40,
\end{aligned}$$

which is not the answer one would like. The reason for this is in the "de-sugaring" of

$$\underline{\text{let}}\ f = \lambda n.(n=0\to 1, n\times f(n-1))\ \ \underline{\text{in}}\ \ f5$$

which is rendered as a λ-expression as follows:

$$(\lambda f.f5)\lambda n.(n=0\to 1, n\times f(n-1)).$$

Now one can see the trouble: the "scope" of the f in f5 is the λ-expression λf.f5; there is nothing to tell one that the f in λn.(n=0→1,n×f(n-1)) is the same f as the f in λf.f5. One can get over this in the let-where notation by writing rec before all recursive definitions, thus:

$$\underline{\text{let}}\ \underline{\text{rec}}\ f(n) = (n=0 \rightarrow 1, n \times f(n-1))$$

$$\underline{\text{in}}\ f5$$

I must now investigate how to "unsugar" the let-where notation into the λ-calculus.

In general, a recursive definition will contain many occurrences of the function being defined in the expression on the right-hand side of the equals sign. One can reduce these occurrences to just a single one by prefixing the right-hand side with "λ<function-name>", and applying it to the function. For example:

$$f = (\lambda f \lambda n.(n=0 \rightarrow 1, n \times f(n-1)))f$$

Now one can see that the function I require is such that it is unchanged by an application of $F = \lambda f \lambda n.(n=0 \rightarrow 1, n \times f(n-1))$. The expression $\lambda f \lambda n.(n=0 \rightarrow 1, n \times f(n-1))$ is termed a functional, or function transformer. That is, it takes a function as an argument and produces a function as a result. I try applying this particular transformer to several simple functions of one argument.

(i) C_0^1 (Note: $C_0^1(x) = 0$ for all x)

$$(\lambda f \lambda n.(n=0 \rightarrow 1, n \times f(n-1)))C_0^1 >_\beta \lambda n.(n=0 \rightarrow 1, n \times C_0^1(n-1))$$

$$>_\beta \lambda n.(n=0 \rightarrow 1, 0) = \overline{sg},$$ the signum-bar function.

(ii) suc

$$(\lambda f \lambda n.(n=0 \rightarrow 1, n \times f(n-1)))suc >_\beta \lambda n.(n=0 \rightarrow 1, n \times suc(n-1))$$

$$>_\beta \lambda n.(n=0 \rightarrow 1, n^2),$$ a "disturbed" square function.

(iii) C_1^1 (Note: $C_1^1(x) = 1$ for all x)

$$(\lambda f \lambda n.(n=0 \to 1, n \times f(n-1)))C_1^1 >_\beta \lambda n.(n=0 \to 1, n \times C_1^1(n-1))$$

$$>_\beta \lambda n.(n=0 \to 1, n),$$ a "disturbed" identity function.

(iv) U_1^1 (Note: $U_1^1(x) = x$ for all x)

$$(\lambda f \lambda n.(n=0 \to 1, n \times f(n-1)))U_1^1 >_\beta \lambda n.(n=0 \to 1, n \times U_1^1(n-1))$$

$$>_\beta n.(n=0 \to 1, n \times (n-1)).$$

In each case, the function is changed by the transformer F. However, when one applies F to the factorial function fact, one has:

$$(\lambda f \lambda n.(n=0 \to 1, n \times f(n-1)))\text{fact} >_\beta \lambda n.(n=0 \to 1, n \times \text{fact}(n-1))$$

$$= \lambda n.(n=0 \to 1, \text{fact}(n)) = \text{fact},$$

i.e. fact is unchanged by an application of F.

If H is any λ-expression, and G is such that $G = HG$, then G is said to be a fixed point of H, i.e. it is that λ-expression which is unchanged by an application of H. In my running example, fact is the fixed point of F. As an example, $\lambda x \lambda y.x$ is a fixed point of $\lambda x.xxx$, since

$$(\lambda x.xxx)\lambda x \lambda y.x >_\beta (\lambda x \lambda y.x)(\lambda x \lambda y.x)(\lambda x \lambda y.x)$$

$$>_\beta \lambda x \lambda y.x.$$

Similarly, $\lambda x \lambda y.y$ and $\lambda x.x$ are also fixed points of $\lambda x.xxx$, showing that a λ-expression may have more than one fixed point. Also, any λ-expression whatever is a fixed point of $\lambda x.x$.

The question is, given H, can one find a fixed point of H? Well, certainly

$$(\lambda h.H(hh))(\lambda h.H(hh)) >_\beta H(\lambda h.H(hh))(\lambda h.H(hh)),$$

i.e. for any λ-expression H, $(\lambda h.H(hh))(\lambda h.H(hh))$ is a fixed point of H. Now,

$$\begin{aligned}(\lambda h.H(hh))(\lambda h.H(hh)) &<_\beta (\lambda h.hh)(\lambda h.H(hh)) \\ &<_\beta (\lambda f.(\lambda h.hh)(\lambda h.f(hh))H\end{aligned}$$

So, if H is any λ-expression whatever, $(\lambda f.(\lambda h.hh)(\lambda h.f(hh))H$ is a fixed point of H. I abbreviate $\lambda f.(\lambda h.hh)(\lambda h.f(hh))$ as Y, and so I have the important result:

For all λ-expressions H, $YH =_\beta H(YH)$.

Y is called a fixed point finder. It is simple to confirm that Y has no normal form, and that $Y_1 = (\lambda h\lambda f.f(hhf))(\lambda h\lambda f.f(hhf))$ is also a fixed point finder.

I try using Y to find the fixed point of a few λ-expressions:

(i) $\lambda x.5$.

$$Y(\lambda x.5) =_\beta (\lambda x.5)(Y\lambda x.5) >_\beta 5.$$

Hence a fixed point of $\lambda x.5$ is 5.

(ii) $H = \lambda f \lambda n.(n=0 \to 1, n \times f(0))$

$$YH =_\beta H(YH) = (\lambda f \lambda n.(n=0 \to 1, n \times f(0)))(YH)$$

$$>_\beta \lambda n.(n=0 \to 1, n \times (YH)(0))$$

$$=_\beta \lambda n.(n=0 \to 1, n \times (H(YH)(0)))$$

$$= \lambda n.(n=0 \to 1, n \times ((\lambda f \lambda n.(n=0 \to 1, n \times f(0)))(YH)(0)))$$

$$>_\beta \lambda n.(n=0 \to 1, n \times (0=0 \to 1, 0 \times (YH)(0)))$$

$$= \lambda n.(n=0 \to 1, n \times (1))$$

$$= \lambda n.(n=0 \to 1, n)$$

(iii) $F = \lambda f \lambda n.(n=0 \to 1, n \times f(n-1))$

$$YF =_\beta F(YF)$$

$$>_\beta \lambda n.(n=0 \to 1, n \times (YF)(n-1))$$

$$=_\beta \lambda n.(n=0 \to 1, n \times ((n-1)=0 \to 1, (n-1) \times (YF)(n-2)))$$

$$= \lambda n.(n=0 \to 1, n=1 \to n, n \times (n-1) \times (YF)(n-2))$$

$$=_\beta \ \ldots$$

This process can be continued indefinitely, always pushing the Y deeper into the conditional. Eventually, for any input k to F, one will reach the conditional n=k... which will throw away the Y, giving the value k! to the expression.

I give another example. Suppose $F = \lambda x \lambda y \lambda z.xyz$. Then, when regarded as a function, YF is undefined, since left-most reduction fails to give a normal form:

$$YFab =_\beta F(YF)ab = (\lambda x \lambda y \lambda z.xyz)(YF)ab >_\beta (YF)ab$$

$$=_\beta \ldots \text{ and so on.}$$

However, $\lambda x\lambda y\lambda z.z$ is also a fixed point of F, since

$$(\lambda x\lambda y\lambda z.xyz)\lambda x\lambda y\lambda z.z >_\beta \lambda y\lambda z.(\lambda x\lambda y\lambda z.z)yz$$

$$\lambda y\lambda z\lambda z.z =_\alpha \lambda x\lambda y\lambda z.z,$$

but $\lambda x\lambda y\lambda z.z$ when regarded as a function is defined, since

$$(\lambda x\lambda y\lambda z.z)abc >_\beta c.$$

It can be proved (see M6) that if Q is any fixed point of a λ-expression F, i.e. $Q = FQ$, then if the value of $YFa_1...a_n$ is defined for λ-expressions $a_1,...,a_n$, then $Qa_1...a_n$ is defined and $YFa_1...a_n = Qa_1...a_n$. This theorem and the example shows that YF is the least, or minimal, fixed point of F, the intersection of all possible fixed-points of F.

Since a λ-expression H may have more than one fixed point, this raises the problem of which fixed point G one chooses as the solution of $G = HG$. I define the G I want as the minimal fixed point of H, i.e. I define $G = YH$. Hence one says that:

$$\underline{\text{let}}\ \underline{\text{rec}}\ f = M$$

$$\underline{\text{in}}\ N$$

means:

$$(\lambda f.N)(Y\lambda f.M).$$

This gives the "unsugared" version of an expression using rec.

It should be noticed that although one can now λ-define the factorial function as:

$$\text{fact} = Y\lambda f\lambda n.\text{zero } n\ 1(\text{mult } n(f(\text{pred } n))),$$

there may very well be a "simpler" non-recursive definition. For example, one also has

$$\text{fact} = \lambda u.u(\lambda v.\langle \text{mult}(v\tau_2^1)(v\tau_2^2), \text{suc}(v\tau_2^2)\rangle)\langle 1,1\rangle\tau_2^1$$

The second λ-definition of `fact above is in normal form, but the first has no normal form. This is all right since my definition of λ-definability does not insist that the defining λ-expression be in normal form. In this case, fact n has a normal form for any n, as pointed out in (iii) above.

I go through an example, to show Y at work.

$$\begin{aligned}
\text{fact}\,1 &= (Y\lambda f\lambda n.(n=0\to 1, n\times f(n-1)))1 \\
&= ((\lambda f.(\lambda h.hh)(\lambda h.f(hh)))\lambda f\lambda n.(n=0\to 1, n\times f(n-1)))1 \\
&>_\beta ((\lambda h.hh)(\lambda h.F(hh)))1
\end{aligned}$$

where F is my already used abbreviation for $\lambda f\lambda n.(n=0\to 1, n\times f(n-1))$,

$$\begin{aligned}
&>_\beta (\lambda h.F(hh))(\lambda h.F(hh))1 \\
&>_\beta (F((\lambda h.F(hh))(\lambda h.F(hh))))1 \\
&= ((\lambda f\lambda n.(n=0\to 1, n\times f(n-1)))((\lambda h.F(hh))(\lambda h.F(hh))))1 \\
&>_\beta \lambda n.(n=0\to 1, n\times((\lambda h.F(hh))(\lambda h.F(hh)))(n-1))1 \\
&>_\beta (1=0\to 1, 1\times((\lambda h.F(hh))(\lambda h.F(hh)))(1-1)) \\
&= 1\times((\lambda h.F(hh))(\lambda h.F(hh)))0 \\
&>_\beta 1\times F((\lambda h.F(hh))(\lambda h.F(hh)))0 \\
&= 1\times((\lambda f\lambda n.(n=0\to 1, n\times f(n-1)))((\lambda h.F(hh))(\lambda h.F(hh))))0 \\
&>_\beta 1\times(\lambda n.(n=0\to 1, n\times((\lambda h.F(hh))(\lambda h.F(hh)))(n-1)))0 \\
&>_\beta 1\times(0=0\to 1, 0\times((\lambda h.F(hh))(\lambda h.F(hh)))(0-1)) \\
&= 1\times 1 = 1.
\end{aligned}$$

I show how Y gives a meaning to the definition:

$$f(n) = (n=0\rightarrow 1, f(n)),$$

which has as a solution any function such that $f(0) = 1$. The function transformer is $H = \lambda f \lambda n.(n=0\rightarrow 1, f(n))$.

$$YH0 =_\beta H(YH)0 = (\lambda f \lambda n.(n=0\rightarrow 1, f(n)))(YH)0$$

$$>_\beta (0=0\rightarrow 1, (YH)(0)) = 1,$$

and

$$YH(n+1) =_\beta H(YH)(n+1) >_\beta (\lambda f \lambda n.(n=0\rightarrow 1, f(n)))(YH)(n+1)$$

$$>_\beta ((n+1)=0\rightarrow 1, (YH)(n+1))$$

$$= YH(n+1) \text{ for all } n \geq 0,$$

i.e. $(YH)n$ is undefined for all $n \neq 0$ and $(YH)0 = 1$. Again, one sees that YH is the minimal fixed point of H, the intersection of all possible functions such that $f(0) = 1$, namely the function undefined everywhere except at 0, where $f(0) = 1$.

I also show how Y affects a "non-recursive" recursive definition, that is,

$$\underline{\text{let}}\ \underline{\text{rec}}\ f=M$$

$$\underline{\text{in}}\ N$$

where f does not occur free in M.

This is equivalent to

$$\underline{\text{let}}\ f = Y\lambda f.M$$

$$\underline{\text{in}}\ N$$

But, $Y\lambda f.M =_\beta (\lambda f.M)(Y\lambda f.M)$; and f does not occur free in M, so

$$(\lambda f.M)(Y\lambda f.M) >_\beta M.$$

Thus the recursive definition in this case is equivalent to

let f=M

in N,

as one would hope.

With Y, I can now quite easily define more advanced functions in terms of my elementary functions, for example div and log. div n m is the largest integer not greater than (n/m), and is defined as:

div n m = zero(mod-dif(suc n)m)0(suc(div(mod-dif n m)m)).

The reason for the (suc n) will become apparent upon evaluating div n n with only n there. Hence,

div = Yλfλuλv.zero(mod-dif(suc u)v)0(suc(f(mod-dif u v)v)).

Similarly, log a n is the number of digits in n when expressed in base a, and is defined as:

log a n = zero(div n a)1(suc(log a(div n a))).

Hence;

log = Yλfλuλv.zero(div v u)1(suc(f u(div v u))).

One can in fact define a sequence of fixed point finders, starting with Y as follows:

Define a new fixed point finder Y' by:

$$\underline{\text{let}}\ \underline{\text{rec}}\ Y'(F) = F(Y'F)$$

i.e. $$\underline{\text{let}}\ \underline{\text{rec}}\ Y' = \lambda F.F(Y'F)$$

or $$Y' = Y\lambda Y'\lambda F.F(Y'F),$$

or, $$Y' = Y\lambda y\lambda f.f(yf).$$

Now, if one lets $G = \lambda y\lambda f.f(yf)$, one has that YG is another fixed point finder.

Hence, define $$Y_0 = Y$$

and $$Y_{n+1} = Y_nG \quad \text{for all} \quad n \geq 0.$$

Then, for all n, $Y_n =_\beta GY_n$, since for $n=0$ one has

$$\begin{aligned} Y_0 &\quad \lambda f.(\lambda h.hh)(\lambda h.f(hh)) \\ &>_\beta \lambda f.(\lambda h.f(hh))(\lambda h.f(hh)) \\ &>_\beta \lambda f.f((\lambda h.f(hh))(\lambda h.f(hh))) \\ &<_\beta \lambda f.f((\lambda h.hh)(\lambda h.f(hh))) \\ &<_\beta \lambda f.f((\lambda f.(\lambda h.hh)(\lambda h.f(hh)))f) \\ &= \lambda f.f(Y_0f) \\ &<_\beta (\lambda y\lambda f.f(yf))Y_0 \equiv GY_0, \end{aligned}$$

and for any $n>0$, if one supposes that $Y_{n-1} =_\beta GY_{n-1}$, then

$$\begin{aligned} Y_n &= Y_{n-1}G \quad \text{by definition of } Y_n \\ &= (GY_{n-1})G \quad \text{by induction hypothesis} \\ &= (\lambda y\lambda f.f(yf))Y_{n-1}G \\ &>_\beta (\lambda f.f(Y_{n-1}f))G \\ &>_\beta G(Y_{n-1}G) = GY_n \quad \text{by definition of } Y_n. \end{aligned}$$

With this lemma, I may now show that for all $n \geq 0$, $Y_n F =_\beta F(Y_n F)$ for any λ-expression F, i.e. each Y_n is a fixed point finder. For,

$$\begin{aligned} Y_n F &=_\beta G Y_n F \text{ by lemma} \\ &= (\lambda y \lambda f.f(yf)) Y_n F \\ &>_\beta (\lambda f.f(Y_n f))F \\ &>_\beta F(Y_n F). \end{aligned}$$

It should be noted that $Y_1 = Y_0 G = YG = (\lambda f.(\lambda h.hh)(\lambda h.f(hh)))(\lambda y \lambda f.f(yf))$

$$\begin{aligned} &>_\beta (\lambda h.hh)(\lambda h.(\lambda y \lambda f.f(yf))(hh)) \\ &>_\beta (\lambda h.hh)(\lambda h \lambda f.f(hhf)) \\ &>_\beta (\lambda h \lambda f.f(hhf))(\lambda h \lambda f.f(hhf)), \end{aligned}$$

which is the Y_1 I gave earlier.

It can be shown that Y_1 and Y are not interconvertible[B2], so one is faced with the question, are the fixed points found by Y_1 and Y the same? I leave the problem of whether the Y_n's are equivalent, since by definition, the only Y_i I am interested in is $Y_0 \equiv Y$.

I now look at simultaneous recursion. Suppose that one has f and g defined recursively in terms of each other. Then, one will have that

$$f = E_1$$

and

$$g = E_2,$$

where E_1 and E_2 are the λ-expressions defining the functions f and g, but containing occurrences of the functions being defined. As before, I can reduce these occurrences to just one as follows: Replace every (free) occurrence of f in E_1 and E_2 by $(h\tau_2^1)$, and every (free) occurrence of g in E_1 and E_2 by $(h\tau_2^2)$, to form D_1 and D_2

respectively. Then, the definitions above are equivalent to:

$$f = (\lambda h.D_1)\langle f,g\rangle$$

and

$$g = (\lambda h.D_2)\langle f,g\rangle.$$

Hence,

$$\begin{aligned}\langle f,g\rangle &= \langle(\lambda h.D_1)\langle f,g\rangle,(\lambda h.D_2)\langle f,g\rangle\rangle \\ &<_\beta (\lambda h.\langle(\lambda h.D_1)h,(\lambda h.D_2)h\rangle)\langle f,g\rangle \\ &>_\beta (\lambda h.\langle D_1,D_2\rangle)\langle f,g\rangle,\end{aligned}$$

which is now of the form $F = XF$, and hence:

$$\langle f,g\rangle = Y\lambda h.\langle D_1,D_2\rangle,$$

and so

$$f = Y(\lambda h.\langle D_1,D_2\rangle)\tau_2^1$$

and

$$g = Y(\lambda h.\langle D_1,D_2\rangle)\tau_2^2$$

It is clear that in a similar way one can λ-define the simultaneous recursive definition of any number of functions.

As an example, let

$$f(x) = \text{zero } x\ 1(\text{mult } x(g(\text{pred } x)))$$

and

$$g(x) = \text{zero } x\ 1(\text{mult } x(f(\text{pred } x))).$$

Then f and g both define the factorial function. In this case one has that

$$E_1 = \lambda u.\text{zero } u\ 1(\text{mult } u(g(\text{pred } u)))$$

and

$$E_2 = \lambda u.\text{zero } u\ 1(\text{mult } u(f(\text{pred } u))),$$

and so

$$D_1 = \lambda u.\text{zero } u\ 1(\text{mult } u((h\tau_2^2)(\text{pred } u)))$$

and $$D_2 = \lambda u.\text{zero } u\ 1(\text{mult } u((h\tau_2^1)(\text{pred } u))).$$

Hence,

$$f = (\lambda f.(\lambda h.hh)(\lambda h.f(hh)))(\lambda h\lambda x.x(\lambda u.\text{zero } u\ 1(\text{mult } u(h\tau_2^2 (\text{pred } u))))(\lambda u.\text{zero } u\ 1(\text{mult } u(h\tau_2^1(\text{pred } u)))))\tau_2^1,$$

and g is identical with f except that g ends with "$\lambda x\lambda y.y$" and not "$\lambda x\lambda y.x$".

1.5.

The McCarthy conditional is of the form $(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n, e_{n+1})$, where each p_i is a predicate (i.e. a function yielding <u>true</u> or <u>false</u>), and the e_i's are expressions. The p_i's are evaluated from left to right, and the e_i corresponding to the first true p_i is evaluated, and taken as the value of the McCarthy conditional. The e_i's may contain the function being defined, yielding a recursive definition. Let $P_1, P_2, \ldots, P_n$ λ-define the predicates $p_1, p_2, \ldots, p_n$ respectively, and $E_1, E_2, \ldots, E_{n+1}$ λ-define the functions $e_1, e_2, \ldots, e_{n+1}$, except for occurrences of the function being defined. Then,

$$f(x_1,\ldots,x_r) = (p_1(x_1,\ldots,x_r) \rightarrow e_1(x_1,\ldots,x_r), \ldots p_n(x_1,\ldots,x_r) \rightarrow e_n(x_1,\ldots,x_r), e_{n+1}(x_1,\ldots,x_r))$$

is (almost) rendered as a λ-expression as follows:

$$f = \lambda x_1 \ldots \lambda x_r . P_1 E_1 (P_2 E_2 (\ldots (P_n E_n E_{n+1})).$$

If the definition is not recursive, one is finished, and the right hand side is the λ-definition of f. Otherwise, if the definition is recursive, in the right hand side replace each occurrence of the function being defined by a variable not bound in any of the P's or E's, f, say, and prefix the right hand side with "Yλf.". This completes the λ-definition of f.

As an example, I λ-define adjoin, which concatenates two lists together. One has:

adjoin (l,m) = (null(l)→ m,prefix(hl)(adjoin(tl,m))).

Hence, as a first step,

adjoin = λuλv.null u v(prefix(hu)(adjoin(tu)v)),

and finally,

adjoin = Yλfλuλv.null u v(prefix (hu)(f(tu)v)).

Another example is zip, which joins two lists of the same length into one, applying a specified function to corresponding elements from each list:

zip(f,l,m) = (null(l)∧null(m)→ nil,
prefix(f(hl,hm))(zip(f,tl,tm))).

and hence

zip = Yλgλfλuλv.and(null u)(null v)U(prefix(f(hu)(hv))
(g f(tu)(tv))),

where U is the list terminator.

As one last example, I define the function prime, which is such that

prime n is true if n is a prime number, and prime n is false otherwise. I define prime in terms of an auxiliary function, auxfn, as follows:

$$\text{prime} = \lambda u.\text{auxfn } u\ 2,$$

where I have:

$$\text{auxfn}(n,m) = (n=m \rightarrow T, \text{divides}(m,n) \rightarrow F, \text{auxfn}(n,\text{suc } m)),$$

i.e. $$\text{auxfn} = Y\lambda f\lambda u\lambda v.\text{equal } u\ v\ T(\text{divides } v\ u\ F(f\ u(\text{suc } v))),$$

where divides m n is true if m divides n, and false otherwise:

$$\text{divides} = \lambda u\lambda v.\text{zero}(\text{abs-dif}(\text{mult}(\text{divide } v\ u)u)v).$$

A function f is called McCarthy computable over $\mathfrak{F}$ (a set of functions, called the base functions) if f can be defined in terms of the base functions using the McCarthy conditional. It can be shown that any μ-recursive function can be McCarthy computed using as base functions either zero, suc and pred, or suc and equal[M1]. Since I have λ-defined zero, suc, pred and equal, it follows that any McCarthy computable function is λ-computable, and hence that any μ-recursive function is λ-computable. Section 1.6. has a more direct proof of the equivalence of μ-recursiveness and λ-definability.

1.6.

This section is devoted to one half of the proof of the equivalence of μ-recursive functions and λ-definable functions. I prove here that every μ-recursive function is λ-definable. While this is by no means original, the work presented here was executed independently by the author. The compiler is certainly new.

The μ-recursive functions are constructed from base functions by a finite number of applications of function transformers. The base functions are suc, and C_k^n and U_k^n for all $n,k \geq 0$. C_k^n and U_k^n are such that $C_k^n(x_1,\ldots,x_n) = k$ and $U_k^n(x_1,\ldots,x_n) = x_k$. The function transformers are S_k^n, R^n and μ. S_k^n is such that $S_k^n(f,g_1,\ldots,g_k)(x_1,\ldots,x_n) = f(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n))$. R^n is such that $R^n(g,h)(x_1,\ldots,x_{n-1},0) = g(x_1,\ldots,x_{n-1})$, and $R^n(g,h)(x_1,\ldots,x_{n-1},x_n+1) = h(x_1,\ldots,x_{n-1},x_n$ $R^n(g,h)(x_1,\ldots,x_{n-1},x_n))$. μ is such that $\mu(r)(x_1,\ldots,x_n)$ = the least y such that $r(x_1,\ldots,x_n,y) = 0$.

I prove the theorem by showing that the base functions are λ-definable, and that the function transformers preserve λ-definability.

suc is λ-defined by $(\lambda u \lambda f \lambda x.f(ufx))$.

C_k^n is λ-defined by $(\lambda x_1 \ldots \lambda x_n.k)$, i.e. $(\lambda x_1 \ldots \lambda x_n \lambda f \lambda x.f^k x)$, where as usual, $f^k x$ is an abbreviation for $f(f(\ldots(fx)))$, with k f's.

U_k^n is λ-defined by $(\lambda x_1 \ldots \lambda x_n.x_k)$.

S_k^n is λ-defined as $(\lambda f \lambda g_1 \ldots \lambda g_k \lambda x_1 \ldots \lambda x_n.F(g_1 x_1 \ldots x_n)\ldots(g_k x_1 \ldots x_n))$.

R^n is a little more difficult to derive. Let F abbreviate $fx_1 \ldots x_{n-1}$, G abbreviate $gx_1 \ldots x_{n-1}$, and H abbreviate $hx_1 \ldots x_{n-1}$. Then, I need to find an f such that $FO = G$, and $F(\text{suc } x_n) = Hx_n(Fx_n)$. Consider:

$$N = N_0 = \langle 0,FO\rangle = \langle 0,G\rangle, \text{ by definition of } F,$$

and $$N_i = \langle i,Fi\rangle.$$

I require an M such that:

$$MN_i = \langle \text{suc } i, F(\text{suc } i)\rangle$$
$$= \langle \text{suc } i, Hi(Fi)\rangle \text{ by definition of } F.$$

Hence, M is $\lambda u.\langle \text{suc}(u\tau_2^1), H(u\tau_2^1)(u\tau_2^2)\rangle$, for:

$$MN_i = \lambda u.\langle \text{suc}(u\tau_2^1), H(u\tau_2^1)(u\tau_2^2)\rangle\langle i, Fi\rangle$$
$$>_\beta \langle \text{suc}(\langle i,Fi\rangle\tau_2^1), H(\langle i,Fi\rangle\tau_2^1)(\langle i,Fi\rangle\tau_2^2)\rangle$$
$$>_\beta \langle \text{suc } i, Hi(Fi)\rangle = \langle \text{suc } i, F(\text{suc } i)\rangle \text{ for all } i \geq 0.$$

Therefore,

$$M^0N = \langle 0, G\rangle,$$

and

$$M^nN = \langle n, Fn\rangle.$$

Hence,

$$(\lambda f\lambda x.f^n x)MN >_\beta \langle n, Fn\rangle,$$

or,

$$nMN >_\beta \langle n, Fn\rangle.$$

Therefore,

$$Fn = \langle n, Fn\rangle\tau_2^2 = (nMN)\tau_2^2.$$

Hence,

$$f = \lambda x_1 \ldots \lambda x_{n-1}\lambda x_n.x_n MN\tau_2^2,$$

where $M = \lambda u \lambda x.x\,\mathrm{suc}\,(u\tau_2^1)\,(hx_1 \ldots x_{n-1}(u\tau_2^1)(u\tau_2^2))$

and $N = \lambda x.x0(gx_1 \ldots x_{n-1})$.

Therefore, I finally have that

$$R^n = (\lambda g \lambda h \lambda x_1 \ldots \lambda x_n.x_n(\lambda u \lambda x.x((\lambda u \lambda f \lambda x.f(ufx))(u \lambda x \lambda y.x))$$
$$(hx_1 \ldots x_{n-1}(u \lambda x \lambda y.x)(u \lambda x \lambda y.y)))(\lambda x.x(\lambda f \lambda x.x)$$
$$(gx_1 \ldots x_{n-1}))\lambda x \lambda y.y).$$

μ may be derived as follows: I define the function h such that

$$\mu(r)(x_1, \ldots, x_n) = h(x_1, \ldots x_n, 0).$$

by $$h(x_1, \ldots, x_n, y) = \begin{cases} y & \text{if } r(x_1, \ldots, x_n, y) = 0 \\ h(x_1, \ldots, x_n, y+1) & \text{otherwise.} \end{cases}$$

Let H abbreviate $hx_1 \ldots x_n$, and R abbreviate $rx_1 \ldots x_n$. Then,

$$H = (\lambda y.Ry \ldots),$$

where the ... denotes λ-expressions to be determined. If $Rn >_\beta 0$, I want $Rn \ldots >_\beta n$, so therefore I must choose the ... = Ay, where A is still to be determined. If $Rn >_\beta m>0$, then I want $RnAy >_\beta H(\mathrm{suc}\ y)$. Hence, $A = \lambda x.H(\mathrm{suc}\ y)$. Therefore, I have so far that

$$H = (\lambda y.Ry(\lambda x.H(\mathrm{suc}\ y))y)$$
$$= (\lambda g.(\lambda y.Ry(\lambda x.g(\mathrm{suc}\ y))y))H$$

Hence,

$$HO = Y(\lambda g.(\lambda y.Ry(\lambda x.g(\text{suc } y))y))O,$$

i.e. $h = \lambda x_1 \ldots \lambda x_n.Y(\lambda g.(\lambda y.rx_1 \ldots x_n y(\lambda x.g(\text{suc } y))y))O,$

and so I have finally that

$$\mu = (\lambda r \lambda x_1 \ldots \lambda x_n.(\lambda f.(\lambda h.hh)(\lambda h.f(hh)))(\lambda g.(\lambda y.rx_1 \ldots x_n y$$
$$(\lambda x.g((\lambda u \lambda f \lambda x.f(ufx))y))y))\lambda f \lambda x.x).$$

Now one can easily translate any μ-recursive definition of a function into its λ-definition, by simply replacing each function or functional in the μ-recursive definition by its corresponding λ-expression, provided that the definition is written using the notation (fx) for a function applied to its arguments, rather than $f(x)$.

For example, the function add has a μ-recursive derivation:

$$R^2(U_1^1, S_1^3(\text{suc}, U_3^3)).$$

In my notation, this is

$$(R^2\ U_1^1(S_1^3\ \text{suc}\ U_3^3)).$$

Hence, add is λ-defined as:

$$((\lambda g \lambda h \lambda x_1 \lambda x_2.x_2(\lambda u \lambda x.x((\lambda u \lambda f \lambda x.f(ufx))(u \lambda x \lambda y.x))(hx_1$$
$$(u \lambda x \lambda y.x)(u \lambda x \lambda y.y)))(\lambda x.x(\lambda f \lambda x.x)(gx_1))\lambda x \lambda y.y)(\lambda x_1.x_1)$$
$$((\lambda f \lambda g_1 \lambda x_1 \lambda x_2 \lambda x_3.f(g_1 x_1 x_2 x_3))(\lambda u \lambda f \lambda x.f(ufx))(\lambda x_1 \lambda x_2 \lambda x_3.x_3))).$$

This derivation can be simplified, however. One can do this by performing some of the required substitutions as one compiles from a μ-recursive definition to a λ-definition. Correspondingly, I give a compiler which accepts a μ-recursive definition of a function, in reverse polish notation, and outputs the λ-definition of the same function. My only concession to the working of the compiler is to write μ as μ^n to specify the 'adicity' of the function produced. The compiler uses a stack 's':

on stack violation, set t←0 and goto finish;

set t←0;

loop:set x←input, at end goto finish;

if x=suc then set f←"λuλfλx.f(ufx)";

if $x=C_k^n$ then set f←"$\lambda x_1 \ldots \lambda x_n \lambda f \lambda x . f^k x$";

if $x=U_k^n$ then set f←"$\lambda x_1 \ldots \lambda x_n . x_k$".

if $x=S_k^n$ then set f←"$\lambda x_1 \ldots \lambda x_n . (s_t)((s_{t-k})x_1 \ldots x_n) \ldots ((s_{t-1})x_1 \ldots x_n)$" and set t←t-k-1;

if $x=R^n$ then set f←"$\lambda x_1 \ldots \lambda x_n . x_n (\lambda u \lambda x . x((\lambda u \lambda f \lambda x . f(ufx))(u\lambda x\lambda y . x))((s_{t-1})x_1 \ldots x_{n-1}(u\lambda x\lambda y . x)(u\lambda x\lambda y . y)))(\lambda x . x(\lambda f \lambda x . x)((s_t)x_1 \ldots x_{n-1}))\lambda x\lambda y . y$" and set t←t-2;

if $x=\mu^n$ then set f←"$\lambda x_1 \ldots \lambda x_n . (\lambda f . (\lambda h . hh)(\lambda h . f(hh)))(\lambda g . (\lambda y . (s_t)x_1 \ldots x_n y(\lambda x . g((\lambda u \lambda f \lambda x . f(ufx))y))y))\lambda f \lambda x . x$" and set t←t-1;

set t←t+1;

set s_t←f and goto loop;

finish:if t≠1 then output ("error in input string") otherwise output (s_1);

I exemplify by an example: add in reverse polish notation is

$$U_3^3 \text{ suc } S_1^3 U_1^1 R^2.$$

x	t	stack	output
	0	-	
U_3^3	1	f_1	$(f_1=)\ \lambda x_1\lambda x_2\lambda x_3.x_3$
suc	2	f_1f_2	$(f_2=)\ \lambda u\lambda f\lambda x.f(ufx)$
S_1^3	0	-	$(f_3=)\ \lambda x_1\lambda x_2\lambda x_3.('f_2')(('f_1')x_1x_2x_3)$
	1	f_3	
U_1^1	2	f_3f_4	$(f_4=)\ \lambda x_1.x_1$
R^2	0	-	$(f_5=)\ \lambda x_1\lambda x_2.x_2(\lambda u\lambda x.x((\lambda u\lambda f\lambda x.f(ufx))(u\lambda x\lambda y.x))$
			$(('f_3')x_1(u\lambda x\lambda y.x)(u\lambda x\lambda y.y)))(\lambda x.x(\lambda f\lambda x.x)$
			$(('f_4)x_1))\lambda x\lambda y.y$
	1	f_5	

where the quotes mean that the string contained by the variable is to appear in that position. The result printed out is f_5:

$$\lambda x_1\lambda x_2.x_2(\lambda u\lambda x.x((\lambda u\lambda f\lambda x.f(ufx))(u\lambda x\lambda y.x))((\lambda x_1\lambda x_2\lambda x_3.$$
$$(\lambda u\lambda f\lambda x.f(ufx))((\lambda x_1\lambda x_2\lambda x_3.x_3)x_1x_2x_3))x_1(u\lambda x\lambda y.x)(u\lambda x\lambda y.y)))$$
$$(\lambda x.x(\lambda f\lambda x.x)((\lambda x_1.x_1)x_1))\lambda x\lambda y.y,$$

which is a β-reduction of the λ-expression given earlier for add.

This expression can still be simplified, however, since

$$(\lambda x_1\lambda x_2\lambda x_3.x_3)x_1x_2x_3 >_\beta x_3$$

and

$$(\lambda x_1.x_1)x_1 >_\beta x_1.$$

(It ultimately reduces to

$$\lambda x_1 \lambda x_2 . x_2 (\lambda u \lambda f \lambda x . f(ufx)) x_1 ,$$

i.e. $\lambda u \lambda v . v \text{ suc } u .$)

This completes my exposition of the λ-calculus.

Chapter two

Structure Definitions

2.1

The concept of abstract syntax was introduced by J. McCarthy in M2. An abstract syntax for an object defines what the essential structure of the object is, but is not committed to any method of representing the object. In L1, P. Landin made the important contribution of a notation for specifying the abstract syntax of objects. This has been developed by Standish in S1. A structure definition (SD) defines the abstract syntax of an object, called a constructed object (CO). That is, it defines an object without committing itself to any concrete representation of that object, and it says how to create, decompose and recognise objects. The set of all objects satisfying a given SD (or the union of such sets) constitute what is known as a data space. An SD defines

a) predicates — for testing which of several alternative formats (if there are alternative formats) a CO may have,

b) selectors — for selecting the component parts of a CO once its format is known,

c) constructors — for constructing a CO of given format out of given components.

As an example, I give an (informal) structure definition of the class of arithmetic expressions or arexps.

An arexp has a leftoperand(lrand) which is either an identifier
or an arexp
and an operator(rator)
and a rightoperand(rrand) which is either an identifier
or an arexp.

From this SD I have

a) The predicates isarexp and isidentifier, which test an operand to see if it is an arexp or an identifier, respectively;

b) the selectors slrand and srrand which select the left operand and right operand of an arexp, respectively, and srator which selects the operator of an arexp;

c) the constructor carexp which constructs an arexp out of a given left operand, operator and right operand.

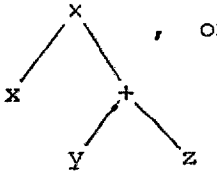
The predicates, selectors and constructors are related by the following identities:

$$\begin{aligned}
\text{isarexp}(\text{carexp}(x,f,y)) &= \text{true} \\
\text{isarexp}(x) &= \text{false} \\
\text{isidentifier}(\text{carexp}(x,f,y)) &= \text{false} \\
\text{isidentifier}(x) &= \text{true} \\
\text{slrand}(\text{carexp}(x,f,y)) &= x \\
\text{srrand}(\text{carexp}(x,f,y)) &= y \\
\text{srator}(\text{carexp}(x,f,y)) &= f \\
\text{carexp}(\text{slrand}(X),\text{srator}(X),\text{srrand}(X)) &= X
\end{aligned}$$

where in these equations x,f,y and X are such as to make both sides meaningful; i.e. x is an identifier, y and X are arexps and f is an operator.

In general, the predicates test for the entities the selectors

select and the constructors construct, and the selectors break open what the constructors put together.

A structure definition is similar in many respects to a context-free grammar, but has the added advantage of explicitly naming the predicates, selectors and constructors involved. Also, a context-free grammar is usually committed to some concrete representation, while a structure definition is not. To illustrate this, one may represent the arexp represented by carexp(x,×,carexp(y,+,z)) by x×(y+z), or xyz+×, or ×x+yz, or mult(x,add(y,z)), or 

, or by any one of many other conceivable alternatives.

A first step to formalising SDs is to write them with a small number of 'bold-face' words, as in

a list is either a null
or has a head (1)
and a tail which is a list

I can define a structure definition as follows:

a structure definition is
either a simple which
has a definiendum which is a identifier
and a definiens which is
either a elementary object description
or a structure definition
or a choice which
has a definiendum which is a identifier
and a definiens which is a structure definition
list of length at least two
or a compound which
has a definiendum which is a identifier

and a definiens which is a structure definition
list of length at least one.

The elements of the list defining a choice are called alternatives, and the elements of the list defining a compound are called components.

This definition of SDs forces me to be more precise in my definitions. Thus, the SD of a list becomes:

a list is either a null which is a elementary object
or a non-null which has a head which is a elementary object
and a tail which is a list.

The elementary objects (EOs) are not further defined, and form the 'base' data space from which all others are constructed. I assume the predicate of equality to be defined over the set of EOs. I allow a simple to be an EO description to allow complicated definitions in terms of functions and sets, etc.. For my convenience, I use 'where' to "factor out" common subdefinitions, especially where these are lengthy. For example,

a solely for the purposes of this example has a ap which is either a x
or a y
and a ip which is either a x
or a y
and a ptr which is either a x
or a y

where a x is a very long structure definition
and a y is a another very long structure definition.

Such uses of where will always be eliminable by textual substitution, however.

To have given this structure definition of SDs I must have already defined identifiers and structure-definition-lists. However, I need to use the structure definition to define identifiers and lists. This circularity of definition should not worry anyone; it besets the definition of all formal systems. One must eventually either leave the primitives undefined, or "define" them in terms of themselves. Such a process is not entirely pointless, since it may yield insight into the primitives themselves.

I can now give the predicates, selectors and constructors which are derived from a structure definition (and which I call the derived functions).

predicates(SD) = {concatenate("is",x)|xε{all identifiers of SD, except those which name a simple which is a structure definition}},

selectors(SD) = {concatenate("s",x)|xε{all identifiers of the components of any compound in SD}},

constructors(SD) = {concatenate "c",x)|xε{all identifiers naming a compound or an EO in SD}}.

The reason for these definitions is as follows. The identifier of a simple is just another name for an already existing object, so I do not want to manufacture new predicates and constructors from it, but I do want to be able to select it from a compound. Hence from the definiendum of a simple I manufacture only a selector, and then only if it is to select a component of a compound. I need to be able to

test for any type of CO, and so I must manufacture predicates from all identifiers of an SD naming an EO, choice or compound. The only COs which need to be constructed are compounds, out of sub-components, and EOs, out of "thin air". Consequently I manufacture constructors only from the identifiers naming compounds and EOs.

In order to aid the eye, hyphens are allowed to be inserted anywhere in an identifier; they are purely decorative. Thus, predicates are usually prefixed with "is-", selectors by "s-" and constructors by "c-". Always prefixing selectors with "s" or "s-" is however a little onerous at times, and so I also allow just the identifier as the selector, unless this leads to an ambiguity.

One can picture a structure definition as defining a tree-structure, with the predicates testing for the objects at the nodes of the tree, the constructors constructing the tree, and the selectors labelling the branches of the tree. It is this approach to structure definitions which directly influenced the development of the Vienna Definition Language (VDL)[L5]. I prefer my approach to that of the Vienna group, since I allow general rooted directed graphs as representations of COs, rather than merely trees. This allows some economy of description to be achieved when defining the λ-machines of chapters 3 and 4. Each arc of a graph is labelled with the appropriate selector, and thus no two arcs emanating from one node may have the same label. Each graph has an associated predicate, which tests for that type of graph. This includes the case of EOs, which are 1-node graphs. The constructors construct the graphs from given subgraphs by creating a new node and linking it to its arguments by directed arcs labelled with the appropriate selector names. Thus in particular one can create as many instances of an EO as one likes, by using the appropriate constructor to create the 1-node graph that is the EO.

Also, it is evident that constructors by themselves cannot create any structures other than trees. I shall give the mechanism by which general graphs may be constructed below. It is clear from this graphical interpretation that one can extend the predicate of equality from the set of EOs to the set of COs. How this may be done by a machine in any particular case will be sketched later. I shall assume from now on that this extended predicate of equality is available.

Since all derived functions are given by the identifiers of an SD, at any point one can break off the definition, and leave the identifier to be defined elsewhere. This allows mutually recursive definitions, which would otherwise be impossible to specify.

Using a VDL-like notation, I give a concise representation of SDs as follows. If the identifier "id" is a simple, defined to be a O, then I write

$$id = O.$$

If the 'O' is an EO to which I do not need to give a denotation (see below) but only a name, then I write

$$id = EO.$$

If it is a choice, defined in terms of the structure definitions $1,2,\ldots,n$, then I write

$$id = 1 \vee 2 \vee \ldots \vee n.$$

If it is a compound, defined in terms of the structure definitions $1,2,\ldots,n$, then I write

$$id = \langle 1,2,\ldots,n \rangle.$$

Brackets are used to resolve any ambiguities, and for convenience, I allow a colon ":" to be used in place of an "=".

Thus, the SD given earlier for lists would be more concisely written as

list = (null:EO)∨(non-null=<head:EO,tail:list>).

I use the convention that "=EO" may be omitted. Thus, if an identifier is not specified elsewhere, it is an elementary object, supposed distinct from all other EOs. Hence, I now have:

list = null∨(non-null=<head,tail:list>),

and from this are derived the predicates:

is-list, is-null, is-non-null, is-head (but not is-tail),

the selectors

s-head, s-tail,

and the constructors

c-null, c-non-null.

I use the convention that the last identifier of a list of alternatives may be omitted, since its associated functions are obvious from the preceding ones. This is particularly useful if a SD has only two alternatives in a choice. Thus my SD of a list finally looks like:

list = null∨<head,tail:list>,

which is superficially the definition (1).

Furthermore, if the identifier on the left-hand side of an "=" sign is the same as the identifier representing a structure definition on the right-hand side, then I omit the "=" sign and the identifier to its left. Thus, if the set of nni's has been defined, I contract an SD containing

'nni = nni'

to one containing only

'nni',

this being the right-hand occurrence of the identifier 'nni' in the original SD. The above definition may look peculiar, but it just means that the selector "s-nni" selects an object which is a nni.

If one needs to give a denotation for an EO, one may do it by underlining a group of symbols, e.g. 'nil', which is the denotation of the null list. One has to be careful when talking about denotations of EOs, because by hypothesis one is talking about abstract objects, which have no representation. It is sometimes convenient however to consider representations of COs (as directed graphs), and for this one needs denotations of abstract EOs.

In order to avoid the tedious repetition of definitions, I need the Van Wijngaarden type of meta-definition[W5], or SD schema. For example,

a X-list is either a null
or has a head which is a X
and a tail which is a X-list

defines an infinite class of structures, with their attendant predicates,

selectors and constructors, one for each substitution of an identifier for "X" in the schema. As in the Van Wijngaarden grammars[W5], each occurrence of a given meta-identifier in a SD schema must be replaced consistently throughout by the same identifier. One can use 'where' to qualify the strings which may replace a given meta-identifier. The SD schemata are at their most powerful when an instance of a schema is used in a set of SDs, with its meta-identifiers replaced by those very identifiers being defined.

I now give a few elementary structure definitions.

letter = a∨b∨c∨...∨x∨y∨z.

digit = 0∨1∨2∨...∨7∨8∨9.

alphanumeric = letter∨digit

By my convention of omitting "=EO", "a", "b", ..., "0", "1", ..., are all defined to be EOs.

X-string = empty∨<stem:X,stern:X-string>

where I have here made use of the convention of omitting the identifier of the last alternative of a choice.

identifier = <stem:letter,stern:alphanumeric-string>

I can now give the definition of a function such as 'concatenate':

concatenate: X-string×X-string → X-string,

concatenate(s,t) = (is-empty(s) → t,c-non-empty(stem(s),
concatenate(stern(s),t))).

Strictly speaking, I should write 'concatenate_X', a different concatenation function for each substitution for X. However, since the domain

of definition of any given 'concatenate' will usually be known from the context, I omit the subscripts X.

Using the full power of the Van Wijngaarden meta-syntax, I can define an 'X-list-of-length-at-least-n' in terms of an 'X-list' as follows:

X-list-of-length-at-least-N-plus-one = <head:X,tail:X-list-of-length-at-least-N>

where N∈L({N},{zero,one,plus},{N→zero,N→N-plus-one},N), (i.e. is a member of the set of terminal productions of the given grammar),

X-list-of-length-at-least-zero = nullv<head:X,tail:X-list>.

In practice, one can abbreviate one-plus-...-plus-one to a decimal integer.

Now, a structure definition is defined as

structure-definition =

(simple = <definiendum:identifier,definiens:elementary-object-descriptionvstructure-definition>)v

(choice = <definiendum:identifier,definiens:structure-definition-list-of-length-at-least-one-plus-one>)v

(compound = <definiendum:identifier,definiens:structure-definition-list-of-length-at-least-one>).

Before I can define the derived functions from the structure of an SD, I need a partial function, 'repr', say,

repr:{x|is-identifier(x)} → EO

which maps an identifier naming a function into that function. In particular,

repr("repr") = repr.

I will also need two list-processing functions. For convenience, I abbreviate "s-head" to "h" and "s-tail" to "t". Also, I give myself a special dispensation and leave out the brackets round the argument of a monadic function when this can cause no confusion:

elem(i) = λL.(i=1 → hL,elem(i-1)(tL)),

and length(L) = (isnull(L) → 0,length(tL)+1).

elem(i) yields the function which chooses the ith element of a list, and length yields the number of elements in a list.

I can now define, for a structure definition SD,

predicates(SD) = {repr(concatenate("is",x))|xεid-set(SD)}
where rec id-set(X) =
(is-simple(X) → (is-elementary-object-description(M) → {s-definiendum(X)},φ),
 is-choice(X) → {s-definiendum(X)}Uid-set(elem(i)(M))|1≤i≤length(M),
 is-compound(X) → {s-definiendum(X)}Uid-set(elem(i)(M))|1≤i≤length(M))
 where M = s-definiens(X),

selectors(SD) = {repr(concatenate("s",x))|xεid-set(SD)}
where rec id-set(X) =
(is-simple(X) → φ,
 is-choice(X) → U id-set(elem(i)(M))|1≤i≤length(M),
 is-compound(X) → {s-definiendum(elem(i)(M))|1≤i≤length(M)} U
 id-set(elem(i)(M))|1≤i≤length(M))
 where M = s-definiens(X),

and

constructors(SD) = { repr(concatenate("c",x)) | x∈ id-set(SD) }

where rec id-set(X) =

(is-simple(X) → (is-elementary-object-description(M) → {s-definiendum(X)},φ)

is-choice(X) → Uid-set(elem(i)(M)) | 1≤i≤length(M),

is-compound(X) → {s-definiendum(X)}Uid-set(elem(i)(M)) | 1≤i≤length(M))

where M = s-definiens(X).

In a similar manner, derived functions can be manufactured from an SD schema. For example, from the SD schema

X-list = nullv<head:X,tail:X-list>

I have, as before, the functions

is-null, is-non-null, is-head, s-head, s-tail,

and in this case the two functions

is-X-list and c-X-list.

These two functions give a different predicate and constructor for each substitution for X. Strictly, from an SD schema I should derive functionals, represented by identifiers possibly including upper case meta-letters. These functionals would then provide the appropriate derived functions upon being supplied with an identifier. For example, to manufacture the derived functions for a string-list, I would have

is-X-list("string") = is-string-list,

c-X-list("string") = c-string-list,

and also is-null("string") = is-null,

s-head("string") = s-head, etc..

From an SD schema containing n meta-identifiers would be derived functionals requiring n string arguments, one for each meta-identifier.

As in the string-list example, not all these arguments may be required. In general, context should remove any ambiguities between the use of a string as a functional and as a function (e.g. is-null).

In practice, I require the identifiers of an SD to be more general than strings of letters and digits (for instance, s-lst, is-γ), and so I allow an arbitrary string of characters (called a tag) to name a CO.

Two more fundamental structures are:

truth-value = true∨false

and

nni = zero∨<pred:nni> (non-negative-integer).

If one writes 'zero' for 'is-zero', 'pred' for 's-pred', 'suc' for 'c-non-zero' and '0' for 'c-zero', then these four functions (three, if one considers '0' as being a constant) are the functions shown by McCarthy in M1 to be equivalent to the μ-recursive functions. As a directed graph, a nni is just a single chain of edges.

A useful notational extension to structure definitions is to contract a compound containing several component definitions all of the same type into just one containing an indexed tag. For example, the SD

xyz = <ip:nni,ap:nni,ptr:nni,flag:truth-value>

may be contracted to

xyz = <field(i):nni | $1 \leq i \leq 3$,flag:truth-value>.

I now have the selectors s-field(1), s-field(2) and s-field(3) instead of s-ip, s-ap and s-ptr. A genuine extension to the power of structure definitions is to allow an arbitrary number of components of a compound. As a directed graph, this means allowing an arbitrary number of edges to leave a node. For instance, the definition of an X-group is:

$$\text{X-group} = \langle s(i):X \mid i \geq 1 \rangle.$$

In this case the selectors are s-s(i) for all integers i, but not all of them are required actually to select something from the object. That is, the edges leading from a node are not required to be numbered consecutively, or be infinite in number. I do not even require that the node have any edges leading from it at all. An extension of this allows me to define a set, by using s-set-el(x) as a selector for each element x I wish to belong to the set. Thus,

$$\text{X-set} = \langle \text{set-el}(x):x \mid \text{is-X}(x) \rangle.$$

I can define an n-tuple for arbitrary n as:

$$\text{N-tuple} = \langle \text{sel}(i) \mid 1 \leq i \leq \text{dec-repr}(N) \rangle$$

where $N \in L(\{N\},\{\text{zero},\text{one},\text{plus}\},\{N \to \text{zero}, N \to \text{N-plus-one}\},N)$

and $\text{dec-repr}(\text{zero}) = 0,$

$\text{dec-repr}(\text{N-plus-one}) = \text{dec-repr}(N)+1.$

The selectors of an n-tuple are s-sel(i) for $1 \leq i \leq n$; a 0-tuple has no selectors.

In a similar manner I can define an n-tuple, each component of which is an X as:

$$\text{X-N-tuple} = \langle \text{sel}(i):X \mid 1 \leq i \leq \text{dec-repr}(N) \rangle,$$

where N and dec-repr are as before.

If I do not wish to specify beforehand how many elements a tuple may have, I can define an X-tuple as:

$$\text{X-tuple} = \langle \text{sel}(i):X \mid 1 \leq i \leq n \rangle.$$

For constructed objects such as tuples which can have an arbitrary but invariant number of components, I take the appropriate constructor as accepting a list of the required length, and forming its elements into a tuple. Thus c-tuple(n) manufactures the constructor which accepts a list of length n and constructs an n-tuple from it.

For constructed objects which can have an arbitrary and varying number of components, there is no simple way of defining the action of the associated constructor, since functions cannot take an arbitrary number of arguments. I could get over this as for tuples by packing the arguments into a list, but not only is this aesthetically not pleasing, but it may not always be convenient to do so. I therefore take such a constructor as constructing the group (or set, etc.) with no elements, that is, a single node. Elements may then be added by the use of the assignment, or arc-swinging, operator, μ. If a is an arbitrary CO, b a compound, and s a selector, then the effect of

$$\mu(a,s,b) \qquad \text{("assign a to the s component of b")}$$

is as follows: If b has a component selected by the selector s, then the s component of b is made to be a. If b has no s component, but may have a variable number of components one of which could be selected by s, then b is made to have a as an s component. Otherwise, the result is undefined. In graphical germs, if the node b

has an arc labelled s leading from it, then s is "swung" to connect b to a. The object pointed to by s previously may be "lost" by this operation, in the sense that no path through the graph from the root to the object exists. On the other hand, if the node has no arc labelled s leading from it, but could have one, then one is created, joining or "assigning" a to b.

I shall use two abbreviated forms for SDs, as follows:

(i) id = <SD-1 SD-2 ... SD-n>-indicator,

which is an abbreviation of

SD-temp = id-indicator

id = SD-1vSD-2v...vSD-n

and (ii) id = <SD-1,SD-2,...,SD-n>-indicator,

which is an abbreviation of

SD-temp = id-indicator

id = <SD-1,SD-2,...,SD-n>,

where 'indicator' is "list", "group", "set", etc..

This is more than a notational convention; all the derived functions are formed from "id", even those strictly requiring to be formed from "SD-temp". I can allow this because it cannot lead to ambiguity.

2.2

The structure of the λ-machines introduced in chapters 3 and 4 will be given by an SD. The semantics will be given by a function 'transform', which may be considered as a machine for mapping abstract structures into abstract structures. The "programming language" in which each 'transform' is written is roughly the purely functional part of PAL (RPAL)[E1], or Landin's ISWIM[L3], augmented by the use of μ. This uses the λ-calculus for function definition, "sugared" by the use of let and where, together with the symbol rec to indicate a recursive definition. The McCarthy conditional is used for conditional evaluation. I shall use the convention that

$$(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ \ldots \ , p_n \rightarrow e_n)$$

is an abbreviation of

$$(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ \ldots \ , p_n \rightarrow e_n, \text{undefined})$$

Usually the p_i's will be collectively exhaustive, and so the "undefined" option will never be taken. Thus any program written in this language can be immediately transformed into a λ-expression, and evaluated on a λ-machine, provided the action of μ can be simulated. Thus the semantics of the λ-calculus reduction algorithm defined by 'transform' is ultimately defined by the λ-calculus itself. While it must be possible to simulate the action of μ in the λ-calculus, the exact details are tiresome, and I do not intend to pursue them here.

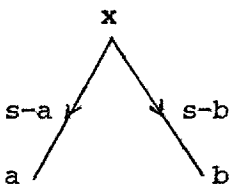
Just as VDL uses an interpreter to run programs specifying transformations of Vienna Objects, so I need an interpreter to run the programs defining 'transform'. I call such an interpreter the "defining machine", since it is this machine that defines how actual parameters

are passed to formal parameters in a function call, or what the result is of applying a selector (μ) to a compound not containing (which cannot contain) the required component, etc..

A very important function of the defining machine is to specify how actual parameters are bound to formal parameters in a function call. Different results will emerge according as the actual parameters to a function are copied or not copied. For example, if one has the structure definition

$$x = \langle a,b \rangle$$

and the structure



then the result of

$$cx(a(x),b(x))$$

(where I have here dropped the "s" from the selectors) will be as in figure 2.1 if the arguments are copied and as in figure 2.2 if the arguments are shared (not copied).

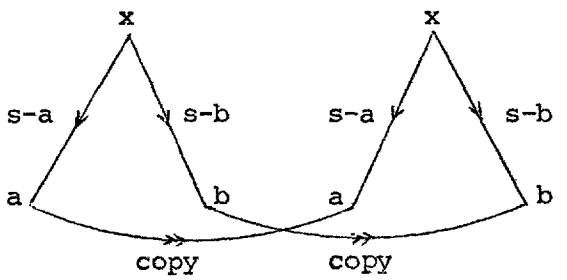


fig 2.1

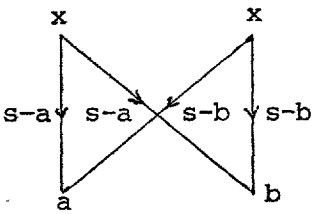


fig 2.2

Peter Landin in L2 achieves the effect of copying by using the function 'separate'. Since the constructors always yield a new node, without μ there is no way of "updating" an already existing node in a graph. Thus there is no way of finding out whether a given λ-machine copies or shares its arguments. This is well known from the LISP language: in pure LISP, there is no way of discovering how storage is allocated, but as soon as assignments are allowed in the system a programmer can discover his implementor's storage strategy.

Clearly, I could give several different defining machines, depending on how I want my programs to be interpreted, and what transformations of data spaces I wish to specify. For my definition of 'transform', I will take the defining machine as sharing all arguments. I shall also take the result of applying a selector to a compound not containing the required component as being undefined, etc.. That is, the defining machine ceases interpretation of the function transform at that point, and halts displaying some error message. One may specify different machines according to the type of programming language one wishes to use: for instance, my ISWIM-PAL machine, or one allowing sequential instructions as on the VDL pattern.

Another important function of the defining machine is to specify whether arguments are passed evaluated (call by value) or unevaluated (call by name). For my functions, I will take arguments as being passed evaluated. The evaluated/unevaluated choice for argument passing is independent of the copied/shared choice. Thus there are four methods of passing arguments to functions. For emphasis, ALGOL 60 passes its name parameters by name/sharing and its value parameters by value/copy (value parameters act as local variables which can be assigned to). On the other hand, ALGOL 68 passes all parameters by

value/sharing. It is because the parameters are passed by value that copying is not necessary.

Copying the arguments if they are a tree structure causes no problems. The defining machine, in order to be able to effect the interpretation, must do some "hidden" housekeeping operations, invisible to the function transform and the associated data spaces. I can therefore suppose that the defining machine keeps associated with each node a list of the selectors labelling the edges leaving that vertex. It is then simple to trace recursively down a tree until the terminal nodes are reached, and then construct a copy of the tree. If the data structure is not a tree, but has more than one edge incident on a node (a directed acyclic graph), then a direct copy cannot be performed. For instance, a direct copy of the graph of figure 2.3 gives figure 2.4.

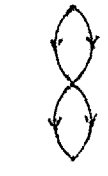


fig 2.3

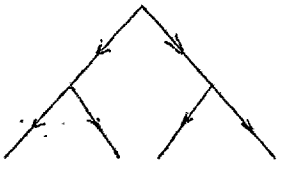


fig 2.4

I can get round this by supposing that when the defining machine interprets a constructor, it also attaches to the created node a unique generation number. Then, during the copying operation, each node can be checked to see if it has been copied previously, and if so, then the already existing copy can be used. This presupposes an extra mechanism to keep track of already existing copies, but there is no great difficulty in this. This copying method will also work for cyclic graphs, since upon reaching a previously copied node, one proceeds no further along that path. By a similar mechanism, the defining machine could implement the predicate of equality over COs, tracing through a graph

and checking the equality of corresponding nodes. To be able to do this, the defining machine does not even have to be able to implement the full range of EOs; when interpreting a constructor constructing an EO, it need only create a new node and associate with the node an integer index, each class of similar EOs being allocated the same index. Then, the predicate of equality between constructed EOs reduces to determining the equality of two integer indexes. In Chapter 4, one of the McCarthy conditionals used has as a predicate an expression which evaluates to one of the truth-values true or false. Thus my defining machine must evaluate a test for equality between two EOs (ultimately, all predicate tests reduce to tests for equality between EOs) to one of the values true or false. In this sense, the truth-values are more fundamental than any other type of EO.

Chapter three

The S.C.E. Machine

3.1.

In this first section of chapter three, I describe most of the notation and machinery which will be used later.

I have already given the definition of lists in section 2.1; I shall need a concrete representation of lists other than as directed graphs, and the one I shall use is as follows. The result of c-null (the null list) is represented by an empty bracket pair "()", or by "nullist", two expressions which may be used interchangeably. I represent a non-null list by the elements of the list in a sequence, separated by commas, and enclosed by brackets. Thus

$$\text{c-list}(x_1,()) = (x_1)$$

$$\text{c-list}(x_2,(x_1)) = (x_2,x_1)$$

$$\text{c-list}(x_3,(x_2,x_1)) = (x_3,x_2,x_1),$$

and so on. It should be noted that

$$\text{c-list}((),x) = (\text{nullist},x)$$

and is therefore entirely different from

$$\text{c-list}(x,()) = (x).$$

As before, I write "h" for "s-head" and "t" for "s-tail". I denote the composition of two functions by a dot, i.e. $f \cdot g = \lambda x.f(g(x))$. Since $f \cdot (g \cdot h)$ and $(f \cdot g) \cdot h$ both have the same meaning, the brackets

may be omitted without ambiguity. I denote $f \cdot f$ by f^2, $f \cdot f \cdot f$ by f^3, and so on. Functional composition binds more tightly than functional application; thus $t \cdot h(L)$ means $(t \cdot h)(L)$, and not $t \cdot (h(L))$. Indeed, $(t \cdot h)(L)$ is only defined if $h(L)$ is a list, and $t \cdot (h(L))$ is only defined if $h(L)$ is a function. I use this to save writing brackets, which always produce confusing clutter in a complicated expression. Thus, $h(t(h(L)))$ would be written as $h \cdot t \cdot h(L)$, or even as $h \cdot t \cdot h\ L$, where as before I drop the brackets round the argument of a monadic function. For convenience, I give myself a special dispensation and also write functions consisting solely of h's and t's without their infixed dots. Thus the above expression would be written still more simply as hthL.

I now define several functions which operate on lists, in addition to elem and length, which were defined previously, in section 2.1.

1) $\quad 1st(S) = hS,$

$\quad 2nd(S) = htS,$

$\quad 3rd(S) = ht^2S,$

and in general $nth(S) = ht^{n-1}S$, for all $n \geq 1$.

One may regard 'nth' as an abbreviation of 'elem(n)'. Likewise, I consider t^nS to be an abbreviation of $tl(n)(S)$, where

$$tl(n) = \lambda S.(n=0 \rightarrow S, t \cdot tl(n-1)(S)).$$

2) $\quad prefix(x) = \lambda L.clist(x,L).$

prefix is a function-producing function of one argument (x), which when applied to x produces that function of one argument which prefixes its argument with x. Thus

$$prefix(x)(L) = c\text{-}list(x,L).$$

In order to save myself writing, I shall denote prefix(x)(L) by x:L.

Note that x:nullist = c-list(x,()) = (x).

Further to save myself writing, I shall suppose that x:y:L means x:(y:L), and to obtain the effect of (x:y):L, I must write the brackets explicitly. I take ":" as binding less tightly than either functional application or composition.

3) append(X) = λL.(isnull(X) → L,hX:append(tX)(L)).

The following example shows how this function of one argument works.

append((x,y,z))(L) = (x,y,z,L).

Contrast this with:

prefix((x,y,z))(L) = ((x,y,z),L).

prefix adds a new head to a list, append concatenates two lists together.

4) postfix(x) = λL.(isnull(L) → x:(),hL:postfix(x)(tL)).

postfix is a function similar in action to prefix, but its argument is postfixed to a list.

5) rev(L) = (isnull(L) → nullist,postfix(hL)(rev(tL))).

The function rev reverses a list.

6) makelist(n) = λL.(n=0 → nullist,hL:makelist(n-1)(tL)).

makelist(n) constructs a list of length n from the first n elements of its argument. I shall abbreviate 'rev·makelist(n)' by 'list^n'.

Since I am supposing the predicate of equality to be defined over the set of COs, I have in particular the predicate of equality between lists. Were it not to be available, it could be defined as follows:

7) $$X=Y = (\text{islist}(X)\wedge\text{islist}(Y) \rightarrow hX=hY\wedge tX=tY,$$
$$\sim\text{islist}(X)\wedge\sim\text{islist}(Y) \rightarrow X=Y,\text{false}).$$

The first, third and fourth occurrences of "=" denote the predicate being defined, the second occurrence of "=" is the = required in the definition, and the fifth occurrence of "=" denotes the predicate of equality between list elements.

As a digression, it may be observed that the meaning of any monadic function over the components of a data space can be extended to the complete data space by means of a universal function 'extend'. If one specifies a function d to decompose an element of the data space into a list of its components, a predicate p to recognise the components, and a function c for combining the separate operations on the components, then one can give extend as

$$\text{extend}(p,d,c) = \lambda f.\phi$$
$$\underline{\text{where}}\ \underline{\text{rec}}\ (\phi,\psi) = (\lambda a.(p(a) \rightarrow f(a),\psi(d(a))),$$
$$\lambda L.(\text{is-null}(tL) \rightarrow \phi(hL),c(\phi(hL),\psi(tL)))).$$

Thus, for example, if a logarithm function 'log' is defined only for primes, one can extend it to the domain of all non-zero integers by an application of the function

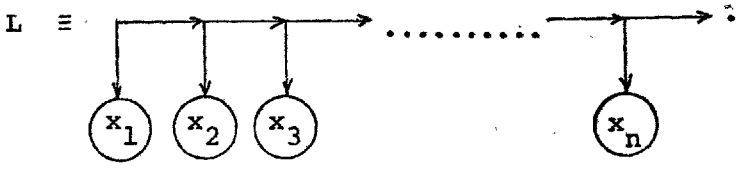
$$\text{extend}(\text{isprime},\text{factorise},+).$$

It should be noted that the exact factorisation provided by 'factorise' is immaterial, since the final factorisation will be unique.

Unfortunately, it is by no means an easy task to extend dyadic functions such as '=' in a similar manner.

This completes the set of list-processing functions for the present; any others needed will be defined as required.

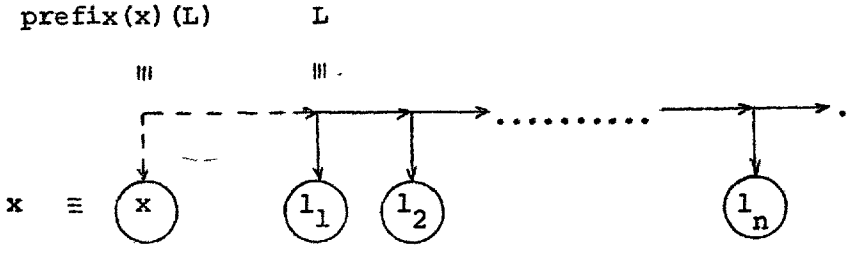
In the representation of lists as directed graphs, the list $L = (x_1, x_2, \ldots, x_n)$ is represented as:

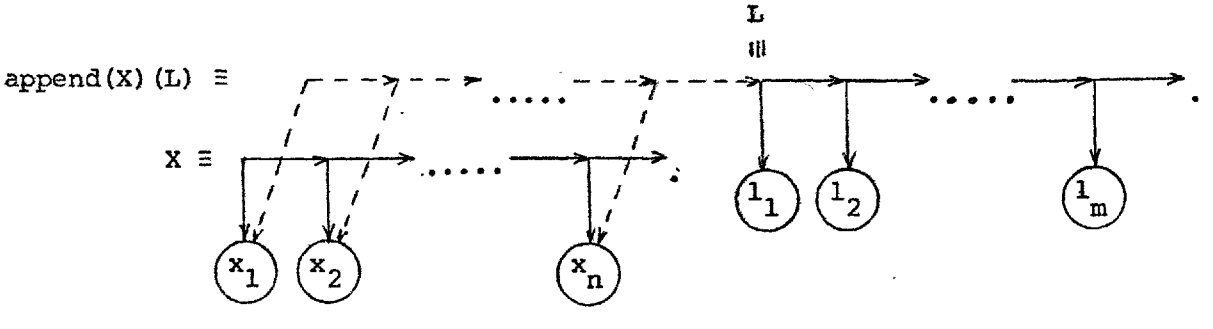


I represent the null list by a dot:

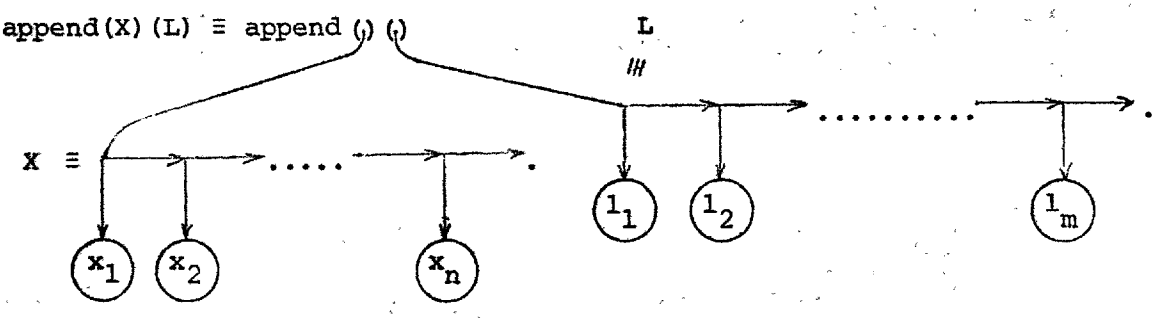
nullist ≡ .

My list-processing functions are now transformations of graphs as follows:

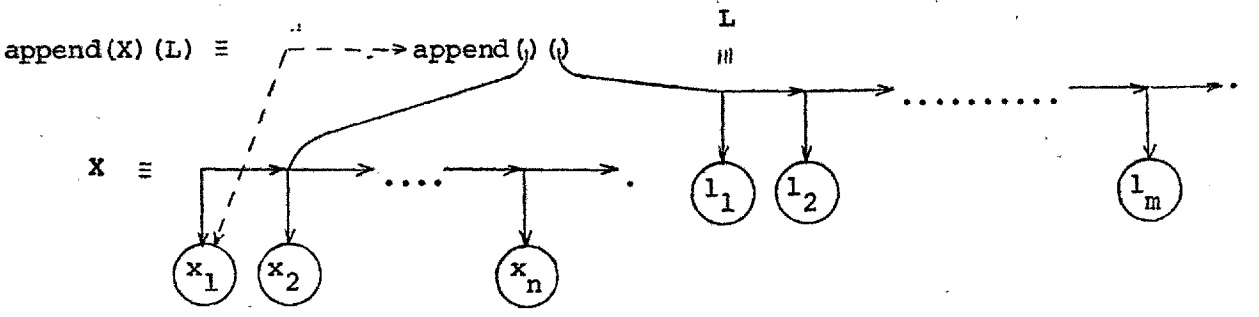




It is worth while looking at append in "slow motion". Initially:



is null(X) is false, so hX:append(tX)(L) is formed:



And so on, until eventually isnull(X) is true, and the chain being formed is linked onto L.

It should be noted that in all these processes the original lists X and L remain "intact", they have not been modified. The only way that either X or L could have been altered is by using the μ operator.

3.2.

In this section I introduce my λ-calculus interpreter, the S.C.E. machine. The machine presented in this section represents just one approach to an implementation model of the λ-calculus; it is the purpose of Chapter 4 to show the relevance of this method of implementation to "procedure-oriented" "block structure" languages. Having informally described my machine, I go on to give a completely formal definition. This is necessary, to provide an absolute standard of reference against which answers to all questions may ultimately be judged. More important, however, is that the function 'transform' given in section 3.2.3., although not explicitly stated in that form, may be written as a single λ-expression. Thus I have the entire definition of the S.C.E. machine in one λ-expression. This is of particular importance when I come to extensions of the S.C.E. machine, which enable it to handle a larger class of data-structures than λ-expressions. Although my machine will then no longer be restricted to interpreting λ-expressions only, the definition of the machine will still be a single λ-expression, and thus the larger class of data-structures will ultimately be defined in terms of the λ-calculus.

3.2.1.

I shall start off by describing a simplified machine, the S.C. machine, which will evaluate arithmetical expressions in reverse polish notation. Since arithmetic expressions do not form part of the λ-calculus directly, one may look upon them as being either abbreviations for certain λ-expressions, or as added δ-rules, as one pleases. The S.C.

machine is so called because it uses a Stack and Control in its operation. I will develop the S.C. machine with my eye on future developments. Because of this, I do not always take the most obvious method of doing things. The best way of introducing the machine is by way of an example. I evaluate 2+5×3-7 on the S.C. machine. In reverse polish, this expression is 2 5 3 × + 7 −, and I load this string into the Control of the machine as follows:

Stack	Control
	2 5 3 × + 7 −

Now, the head of Control is a number, so it is deleted and placed on the Stack:

2	5 3 × + 7 −

and again:

2 5	3 × + 7 −

and again:

2 5 3	× + 7 −

Now the head of Control is an arithmetic operator; I delete it from Control, and apply it to the 5 and 3 on the top of the Stack:

2 15	+ 7 −

and similarly for the +:

17	7 −

Now I carry over the 7:

17 7	−

and now apply the -:

10

Now Control is empty, the computation is finished, and the result, 10, resides in the Stack. And, indeed, 2+5×3-7 = 10.

Having informally introduced my machine, I must now define an abstract machine of which the pencil-and-paper one is a realization. I shall represent the Stack and Control by a list, and shall give a function 'transform' such that one application of transform to the list composed of S and C will advance the computation through one step. I defer giving a structure definition for reverse polish arithmetic expressions until I discuss the S.C.E. machine, and here merely suppose that I have the following four predicates and functions:

(i) isnumber which yields true when applied to an arithmetical constant, and false otherwise,

(ii) is function which tests for an arithmetical function or operator,

(iii) adicity which determines the number of arguments a given function requires, e.g. adicity(+) = 2, adicity(sin) = 1, also adicity(adicity) = 1, and

(iv) apply which applies a function to its argument list.

The function transform is given by:

transform((S,C)) = (isnumber(hC) → (hC:S,tC),
is function(hC) → (apply(hC,list^{n}(S)):t^{n}S,tC)
where n = adicity(hC)).

This function completely specifies my abstract machine, provided that the action of the basic list manipulation functions, isnumber, isfunction, adicity and apply are understood.

If one requires a function that yields the final result of the computation, then such a function is:

transform*((S,C)) = (isnull(C) → hS,transform*(transform((S,C)))).

So far, this is only the normal evaluation method for arithmetic expressions in reverse polish notation. It is however rather unsatisfactory to have the two predicates 'isnumber' and 'isfunction', since in a more complicated machine one may want to treat numbers and (ALGOL) identifiers the same way, and also one may require an actual operator to be put on the Stack, for instance, so that it may be assigned to a variable. These problems are overcome by having the S.C. machine treat numbers and functions in the same way, merely copying each from the head of Control to the head of the Stack. Then, functional application of a function to its arguments occurs when a special symbol (which cannot occur in any other context), namely "γ", appears at the head of Control. I now have to compile any reverse polish expressions into a form suitable for use by the machine. Thus a second attempt at evaluating 2+5×3-7 (reverse polish now $2\ 5\ 3\ \times\ \gamma\ +\ \gamma\ 7\ -\ \gamma$) is:

Stack	Control
	2 5 3 × γ + γ 7 − γ
2	5 3 × γ + γ 7 − γ
2 5	3 × γ + γ 7 − γ
2 5 3	× γ + γ 7 − γ
2 5 3 ×	γ + γ 7 − γ
2 15	+ γ 7 − γ
2 15 +	γ 7 − γ
17	7 − γ
17 7	− γ
17 7 −	γ
10	

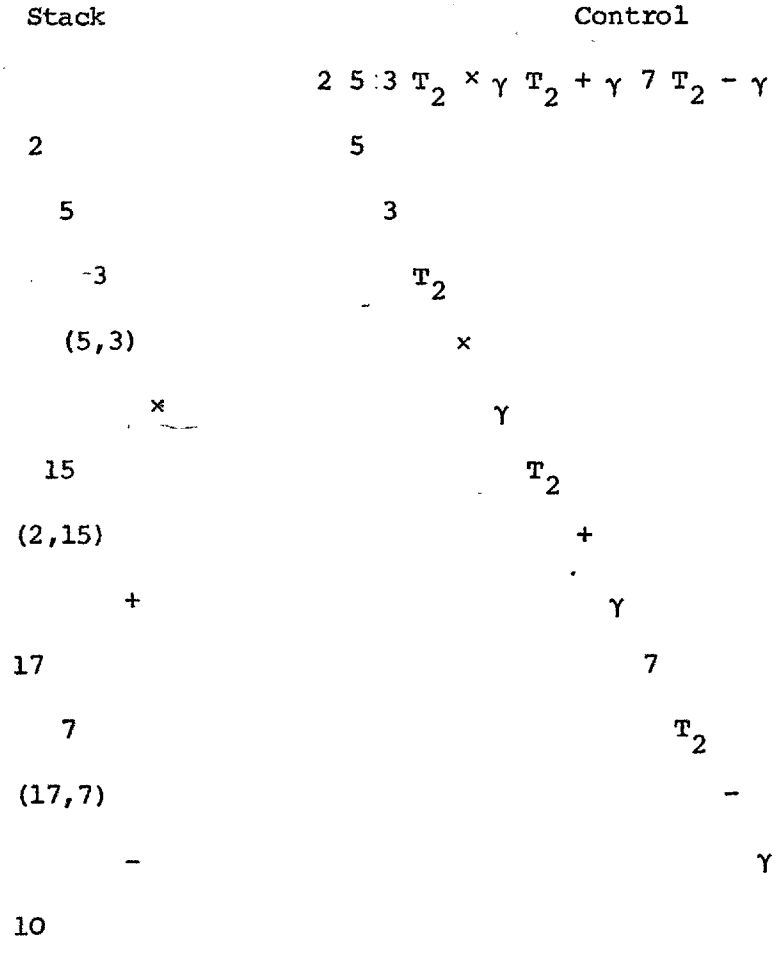
This is better, but now a difficulty appears. When a γ appears at the head of Control, the S.C. machine assumes that the head of the Stack is an operator, and that its arguments lie beneath it. But how many arguments? The operator could be monadic, dyadic, triadic, and so on, or even niladic (i.e. has no arguments, for instance a parameterless ALGOL procedure). As it stands, the S.C. machine has no idea of how many arguments to apply the top of the Stack to when a γ is encountered in Control. There are at least three solutions to this problem. The first is to suppose as before that each function is amenable to a function 'adicity', which when applied to a function returns the number of arguments that function expects. Then, when a γ reaches the head of Control, the adicity of the function on the ead of the Stack is determined, and the function applied to that many arguments. Perhaps a more sensible idea is that one γ should cause the head of the Stack to be applied to the second element on the Stack, rather than the 2nd, 3rd, and so on. Thus a

second solution is to treat + (to take one particular function as an example) as a monadic function which produces a monadic function when applied to an argument. Thus, + applied to 15 produces the "+15" function, a function which adds 15 to its argument. Then (+15) applied to 2 yields 17. This solution is equivalent to defining + as λxλy.x+y. My example would now run as follows, where from now on for convenience I write the head of each list only, all other elements beneath it being unchanged.

Stack	Control
	2 5 3 × γ γ + γ γ 7 - γ γ
2	5
5	3
3	×
×	γ
(×3)	γ
15	+
+	γ
(+15)	γ
17	7
7	-
-	γ
(-7)	γ
10	

It seems best, however, if only one γ is needed for each function. Hence the third solution is to treat every function as accepting one argument, that argument being a tuple. Thus + would have one argument, a pair of numbers. Similarly, a "triadic function" would have one argument, that argument being a triple. Now, in order to create a

tuple on the Stack in readiness as arguments for a function, I need tupling operators T_2, T_3, T_4, ... and so on. Here I have a problem, as the tupling operators are functions and hence will only accept one argument, that argument being a tuple! The way round this is to allow the tupling operators the same status as γ, so that each T_i is a special operator (which cannot occur in any other context and) which turns the top i elements of the Stack into an i-tuple and is applied as soon as it reaches the top of Control. Thus I do not need a γ to apply a T_i to its arguments, since each T_i "incorporates its own γ". It should be noted that T_1 turns the top element of the Stack into a list whose only element is that element, and T_0 places the null list on the Stack. This solution is equivalent to defining + as $\lambda(x,y).x+y$. My running example now goes as follows:



This third version of my machine is the one I shall develop.

I now suppose that I have the following three predicates and two functions:

(i) isconstant which yields true when applied to any arithmetical constant or function and false otherwise,

(ii) isγ which recognises a γ,

(iii) istupler which recognises a T_i,

(iv) tuplicity which delivers the number of arguments a tupler requires, i.e. tuplicity(T_n) = n, and

(v) apply which applies a function to its argument list;

and I can now give my (new) function 'transform':

transform((S,C)) = (isconstant(hC) → (hC:S,tC),
isγ(hC) → (apply(hS,htS):t^2S,tC),
istupler(hC) → (list^n(S):t^nS,tC)
where n = tuplicity(hC)).

For reference, the definitions of transform for the other two versions of the machine are as follows:

for machine 1:

transform((S,C)) = (isconstant(hC) → (hC:S,tC),
is$_\gamma$(hC) → (apply(hS,list^n(tS)):t^{n+1}S,tC)
where n = adicity(hS))

and for machine 2:

transform((S,C)) = (isconstant(hC) → (hC:S,tC),
isγ(hC) → (apply(hS,htS):t^2S,tC))

where apply and adicity are the appropriately defined functions for each case.

3.2.2.

Even with this second definition of the S.C. machine, one would still like to know just how the machine "knows" how to apply +, say, to its argument list. Also, the objects appearing on the Control are just symbols, and not values; for example, VI and ADD would do just as well on the Control as 6 and +. This fits in with a more advanced machine which will have to deal with (ALGOL) identifiers, but leaves me with the problem of how the machine knows that 6 or VI means the number six. The solution to how the machine "knows" how to apply + will not be completely clear until an "extension" of the S.C.E. machine is given in section 3.2.4.

Now, any λ-expression introduces a new level of notation, and indicates a sub-computation to be performed. In any λ-expression containing a sub-expression of the form (λx.L)M, the main computation is to be broken off, and the sub-expression L is to be evaluated, but with the added piece of information that x is to be replaced everywhere by M. Consequently, in evaluating an expression such as (λx.L)M, L is evaluated as on the S.C. machine but with an entry in a lookup list giving the current value (M) of the bound variable x. This entry in the lookup list is called an environment layer, and the lookup list for an evaluation of a λ-expression forms an object termed the environment. This solves the earlier problem, since I can suppose that the machine has a 'primitive environment', an entity in which the values of such 'primitive identifiers' as 6 and + are found. This

turns my S.C. machine into the S.C.E. machine, for I have now added an Environment component. When the machine comes across, say, the identifier 2, it looks it up in the primitive environment, and finds the corresponding value or 'primitive object', and it is this value which is put on the Stack. The form of the 'value' corresponding to a given primitive identifier is not defined; one is not allowed to break open the primitive environment (hereafter abbreviated PE) to see how it works. Indeed, there is little interest in doing so; one need only suppose that the machine has some consistent representation of primitive objects, and can manipulate these representations. One can build machines with different PEs for different purposes, e.g. one for arithmetic, one for list-processing, one for string-handling, etc..

I consider now the λ-expression $(\lambda a.a+a^2)10$. This is to be evaluated in the current environment, i.e. the PE. During this computation, $a+a^2$ is to be evaluated, and this must be done in a new environment, the old environment (PE) plus the layer consisting of the information 'a=10'. I put this computation onto the S.C.E. machine as follows. The reverse polish form of $a+a^2$ is $aaaT_2\times\gamma T_2+\gamma$, and hence the reverse polish form of $(\lambda a.a+a^2)10$ is $10(\lambda a.aaaT_2\times\gamma T_2+\gamma)\gamma$ (since in reverse polish notation, $(\lambda x.L)M$ is $M(\lambda x.L)\gamma$, the function $\lambda x.L$ applied to its argument M). I represent this in what is known as "control syntax" as

$$10\lambda_1\gamma$$

where the λ-marker λ_1 is given by

$$\lambda_1 \equiv \lambda a.aaaT_2\times\gamma T_2+\gamma.$$

This is run as shown in figure A1. Initially, the Stack contains the environment-marker (ε-marker) ε_0, corresponding to the primitive environment, marked '0' in the Environment column. The Control contains the string to be evaluated, $10\lambda_1\gamma$, appended to ε_0. The Environment contains only the PE, no other layers having been added yet. At line 1, a 10 is found at the head of Control. This identifier is looked up in the current environment, which in this case is the PE, and its value found. This value, which is 10, is put on the Stack. At line 2, I find a λ-marker at the head of Control. This represents a body of text to be evaluated at an appropriate time. It must, however, be evaluated in the current environment, augmented by a new environment layer giving the value of the bound variable. Hence, the λ-marker, plus the most recent ε-marker, are joined together to form an object called a λ-closure, and this object is put on the Stack. Unlike an identifier, this involves no looking up of any kind. A λ-closure, plus the argument it is applied to, contains all the information necessary to perform a sub-computation. At line 3, I come to apply the λ-closure to its argument, 10. The λ-closure and its argument are both deleted from the Stack, and a new ε-marker, ε_1, is put on the Stack (it is customary to number ε-markers and the corresponding environment layers from 0 upwards). The γ is deleted from the Control, and the new ε-marker placed in Control. The new, derived, environment is formed by adding a new layer, formed by giving 'a' (the bound variable of the λ-marker part of the closure) the value '10' (the argument on the Stack beneath the Closure). This new environment layer is linked back (by the arrow) to the environment given by the ε-marker part of the closure, which is the environment in which λ_1 must be evaluated, namely the environment 0 (the PE). Finally, the expression to be evaluated in this sub-computation is loaded onto the

Control. All this is shown at line 4. At line 4, the identifier a is encountered at the head of Control, is looked up in the current environment, namely 1, and its value, 10, found in the top-most layer. This value is put on the Stack. Similarly at lines 5 and 6. At line 7, the tupler causes the first and second elements of the Stack to be tupled into a pair. At line 8, the identifier × is encountered at the head of Control, and is looked up in the current environment (1). But, the top-most environment layer of the current environment contains only the information a=10. Hence, the search is continued in the environment to which 1 is linked, namely 0. This is the PE, where the value of × is found. This value is put on the Stack. At line 9, the × is applied. Lines 10, 11 and 12 follow similarly for the +. At line 13, the ε-marker ε_1 is at the head of Control, and is the second element on the Stack. This indicates that the current sub-computation is complete, so the two ε-markers are deleted, and the result of the sub-computation (110) copied down into the new head of Stack. At line 14, the PE marker is encountered, indicating that the computation is finished. Both markers are deleted, and the result left on the top of the Stack at line 15.

Apart from λ- and ε-markers, tuplers and γ, all objects in Control are identifiers, with no intrinsic meaning. The corresponding objects on the Stack, however, are values, and have an intrinsic meaning divorced from any physical representation. Because of this, the primitive objects on the Stack are underlined with a wavy underline in figure A1, to denote that what occurs there is not a representation of an object, but the object itself.

I now evaluate the λ-expression corresponding to

f(7) where f(x) = 2×x+1.

The λ-expression is

$$(\lambda f.f7)(\lambda x.2\times x+1),$$
$$\quad 1 \qquad 2$$

and in reverse polish this is

$$\lambda_2\lambda_1\gamma$$

$$\lambda_1 \equiv \lambda f.7f\gamma$$

$$\lambda_2 \equiv \lambda x.2xT_2\times\gamma 1T_2+\gamma.$$

I number the λ's in the λ-expression from left to right, to help me in constructing the corresponding reverse polish control syntax form. The computation is shown in figure A2. Initially, the control string is loaded on to the Control, on top of the PE marker, which is also on the Stack. At line 1 the λ-closure of the λ-marker λ_2 and the ε-marker ε_0 is formed and put on to the Stack. Similarly at line 2, with λ_1. At line 3, the head of Stack is applied to the element below it. Thus a new environment is set up, whose top environment layer has f (the bound variable of the λ-marker part of the closure) coupled with the value $\lambda_2\varepsilon_0$. Thus the identifier f has as value a body of text plus the environment in which it is to be evaluated. This new layer is linked back to the environment denoted by the ε-marker part of the closure. The new ε-marker is placed on the Stack and Control, and the body of the λ-marker, namely 7fγ, is loaded into the Control. At line 4, the identifier 7 is looked up in environment 1, and is not found in the top-most layer, so the search continues in the environment to which 1 is linked, namely 0, and the value is found there (in the PE). At line 5, the value of f is found immediately in the top-most layer of environment 1, and is put on the

Stack. At line 6 a λ-closure is applied, and a new environment set up, as before. At line 7, the identifier 2 is looked up in the top-most layer of environment 2, and is not found, so the search continues in environment 0. Environment 1 is bypassed in this lookup process; in fact environment 1 is completely inaccessible from environment 2. At lines 7-15 the computation of $2\times x+1$ proceeds, and at line 16 the machine exits from the completed sub-computation; likewise at line 17; and at line 18 the entire computation is completed.

I have always been careful to distinguish between an environment layer, which is a single piece of information, and an environment, which is a chain of environment layers linked together.

At any stage of the S.C.E. machine computation I can mechanically reconstruct the λ-expression represented by that stage of the computation. The best points for such reconstruction are just after entering a new environment, since here a new layer has been added but the subcomputation has not yet been initiated. For example, in figure A1, at line 1 $10\lambda_1\gamma\varepsilon_0$ is in Control, and since λ_1 is an abbreviation of $\lambda a.aaaT_2\times\gamma T_2+\gamma$, this represents

$$(\lambda a.a+a\times a)10,$$

the original expression. At line 4 $aaaT_2\times\gamma T_2+\gamma\varepsilon_1\varepsilon_0$ is in Control, and this therefore represents

$$a+a\times a \text{ where } a=10$$

or $10+10\times 10$;

I expand the meaning of the ε-marker ε_1 by suffixing the expression obtained with "where" followed by the information held in the environment. The where expression is only an intermediate form, since the substitution indicated by it must be performed to obtain the final expression. The expression 10+10×10 is the original λ-expression after one β-reduction.

In figure A2 at line 4 $7f\gamma\varepsilon_1\varepsilon_0$ is in Control, which represents

f7 where f=λx.2×x+1,

or (λx.2×x+1)7.

At line 7 $2\times T_2{}^{\times}\gamma 1 T_2{}^{+}\gamma\varepsilon_2\varepsilon_1\varepsilon_0$ is in Control, which represents

2×x+1 where x=7

where f=λx.2×x+1,

or 2×7+1;

I must expand both ε-markers of Control even though the information in one of them is not required.

Figure A3 shows the computation corresponding to the expression

let g(x)(y) = x+y

in f(3)×g(4)(5)

where f = g2,

which is $(\lambda g.(\lambda f.f(3)\times g(4)(5))(g2))(\lambda x.\lambda y.x+y)$ (1 2 3 4) as a λ-expression, or, in reverse polish control syntax:

$\lambda_3\lambda_1\gamma$

$\lambda_1 \equiv \lambda g.2g\gamma\lambda_2\gamma$

$\lambda_2 \equiv \lambda f.3f\gamma 54g\gamma\gamma T_2{}^{\times}\gamma$

$\lambda_3 \equiv \lambda x.\lambda_4$

$\lambda_4 \equiv \lambda y.xyT_2{}^{+}\gamma.$

This computation shows the machinery of the S.C.E. machine in full swing. Note that at both lines 8 and 25 a λ-closure is yielded as the result of a sub-computation. Also, at line 19 the current sub-computation in environment 4 finishes, and the 5, 4 and g at lines 20, 21 and 22 respectively are looked up in environment 3.

This computation allows more interesting conversions to the computed λ-expressions. For example, at line 11 $3f\gamma 54g\gamma\gamma T_2{}^{\times}\gamma\varepsilon_3\varepsilon_1\varepsilon_0$ is on Control, which represents

(f3)×g(4)(5) where f=(λy.x+y where x=2)
where g=λxλy.x+y
where g=λxλy.x+y,

or ((λy.2+y)3)×((λxλy.x+y)(4)(5));

here I have to expand the ε-marker part of the λ-closure bound to f, and I also have to follow the arrow from environment 3 to environment 1 while expanding the ε_3 marker on Control, before expanding the ε_1 marker on Control. At line 24, $\lambda_4\varepsilon_5\gamma T_2{}^{\times}\gamma\varepsilon_3\varepsilon_1\varepsilon_0$ is on Control, and 5 5 on the Stack, and so this represents

$5\times((\lambda y.x+y$ where $x=4)5)$ where $f=(\lambda x.x+y$ where $x=2)$
where $g=\lambda x\lambda y.x+y$
where $g=\lambda x\lambda y.x+y$,

or $5\times((\lambda y.4+y)5)$;

here, when a γ which has no second operand is met with on Control, the required value is found on the Stack. Strictly, there has been no need in these examples to expand the ε_1 on Control, since there are no identifiers between the ε_3 and ε_1 on Control.

This mechanical method of reconstructing λ-expressions from the S.C.E. machine-states representing them becomes a necessary stage in the proof that the S.C.E. machine implements β-reductions correctly, using the technique of Clement McGowan in M3.

It is interesting to see in figure A3 how the various environments get "hidden", then "reappear" again. For example, in environment 4, one can find only the values of primitive identifiers, and the identifiers x and y, whereas as soon as the machine exits to environment 3 at line 19, one can find only the values of primitive identifiers, and the identifiers f and g. The way the environment-markers are nested on the Stack and Control is called the dynamic nesting of the environments, and this nesting forms the dynamic tree of the computation. The dynamic tree is so called because it reflects the way the computation proceeds dynamically, the way the various components of the computation interact at run-time. The way the environment-markers are nested on the Environment is called the static nesting of the environments, and this nesting forms the static tree of the computation. The static tree is so called because it reflects the way the program is written, the way the various components of the program interrelate at compile-time. The computations of figures A1, A2 and A3

demonstrate different degrees of dynamic tree - static tree development. The trees for figures A1, A2 and A3 are shown in figures 3.1, 3.2 and 3.3 respectively.

fig. 3.1

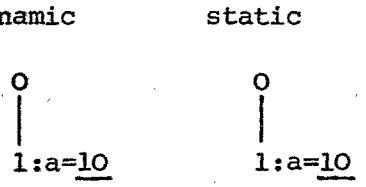


fig. 3.2

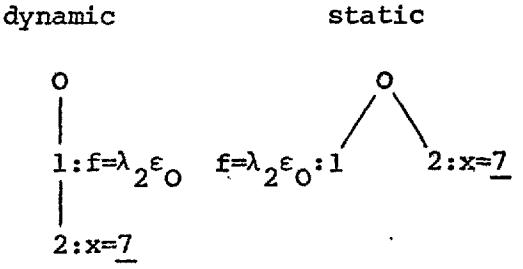
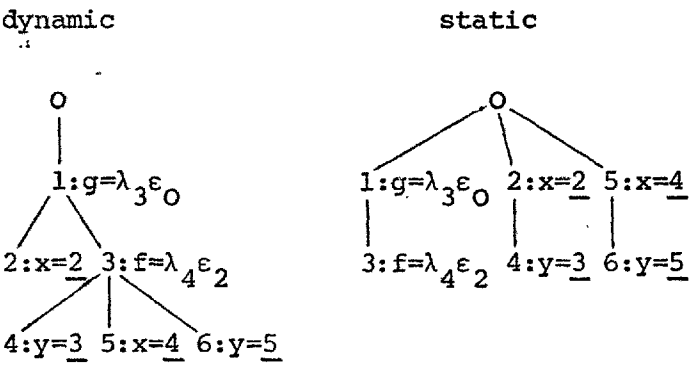


fig. 3.3



In Figure 3.1, the dynamic and static trees are identical. In figure 3.2, the static tree is a "subtree" of the dynamic tree, in the sense that two nodes directly joined in the static tree are connected by a chain of nodes in the dynamic tree. This is the case in ALGOL, and allows a stack-oriented storage regime. Because of this stack orientation, in ALGOL the static and dynamic trees are called the static and dynamic chains. In figure 3.3, the two trees are quite general, and appear to be in no way related. This generality is a feature of the λ-calculus and programming languages like PAL[E1] which are based on it. It comes about because a λ-closure can be carried out of the environment in which it was created, as at lines 8 and 25 in figure A3. Thus, via the λ-marker in the closure, one can gain access

to an environment which in ALGOL would have disappeared forever. Because of this, any storage regime needs run-time garbage collection. Some LISP implementations use a run-time data structure known as a chained stack, to allow for this generality.

Several authors (e.g. Peter Wegner[W1] and Clement McGowan[M3]) have developed so called "stack" λ-calculus interpreters. However, a true stack implementation is impossible, for the reasons cited above. One is constrained to allocating space at the top of the stack, but not being able to release it. In fact, all "stack" λ-machines which have been developed do not deallocate space at the top of the stack, and so implement storage allocation correctly. Eventually, "holes" (in the form of inaccessible environments) will appear in the stack, and stack compaction, the equivalent of garbage collection for this storage regime, can take place. As a matter of interest, Michael Fisher has shown in F1 that for every program which uses a retention strategy there exists a corresponding program which uses a deletion strategy. Furthermore, this second program can be derived from the first in such a way that the result of the theorem is true under all interpretations of free function symbols, i.e. independent of the PE chosen. The basis of this method is the use of Van Wijngaarden's device (see W4 and M7): no procedure is ever delivered as a result of a procedure call; when the procedure is about to be delivered, it is not, but is passed as a parameter to a procedure call. Thus one never exits from a procedure, except at the end of a program, when all procedure exits are performed at once. Again, one marches up the stack relentlessly, never deallocating anything.

A remark on the differences between the S.C.E. machine and S.E.C.D. machine[L1] is in order here. When the S.C.E. machine places ε-markers on the Stack and Control, the S.E.C.D. machine clears its Stack, Control

and Environment components by pushing them down onto a Dump, the old Dump also being pushed down as a component of the new Dump. Thus, the Stack and Control components hold only the current sub-computation. Also, each environment component holds only the values relevant to the current subcomputation, that is, the static chain. The dynamic chain is held in the succession of Environment components in the Dump. Also, there are no tuplers, all functions are monadic; nor are there any appliers, the S.E.C.D. machine automatically supplies appliers when breaking open a combination in order to evaluate the operator and operand. Expressions therefore need not be compiled, as on my machine. Thus the S.E.C.D. machine is in some sense "purer" than mine, but only by choice; I have already investigated and rejected having all functions monadic, and eliminating appliers is straightforward, since it only entails keeping the λ-body as a combination.

Two last full examples of the machine are given. The first is

selfapply twice suc 0

where selfapply = λh.hh

and twice = λfλx.f(fx)

and suc = the successor function

and 0 = zero.

This is $(\lambda h.hh)(\lambda f\lambda x.f(fx))\,suc\ 0$, or, in control syntax,
1 2 3

$$0\ suc\lambda_2\lambda_1\gamma\gamma\gamma$$

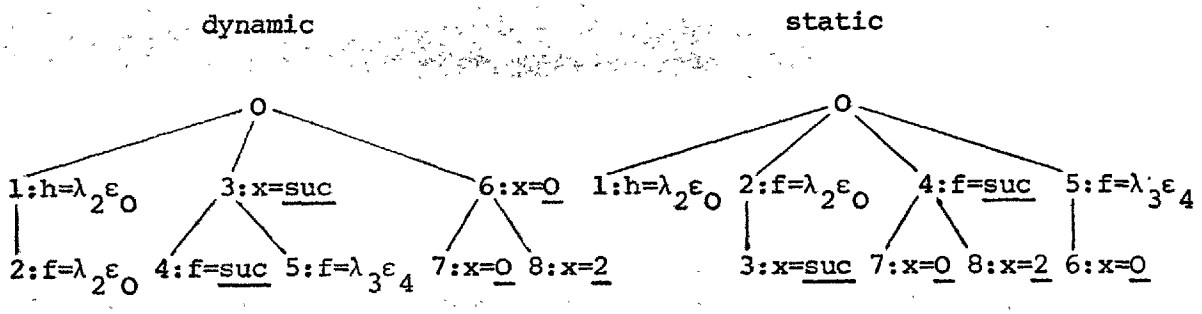
$$\lambda_1 \equiv \lambda h.hh\gamma$$

$$\lambda_2 \quad \lambda f.\lambda_3$$

$$\lambda_3 \equiv \lambda x.xf\gamma f\gamma.$$

The computation is shown in figure A4, and the static and dynamic trees in figure 3.4.

fig. 3.4



In order to discover in which order my machine performs β-reductions, I shall translate every step of this computation to its corresponding λ-expression.

At line 1: 0 suc $\lambda_2\lambda_1\gamma\gamma\gamma\varepsilon_0$ on Control.

This represents: (λh.hh)(λfλx.f(fx))suc 0 original expression

At line 6: $hh\gamma\varepsilon_1\gamma\gamma\varepsilon_0$ on Control, 0 suc on Stack.

Represents: (hh where h=λfλx.f(fx))suc 0

or (λfλx.f(fx))(λfλx.f(fx))suc 0 only reduction possible

At line 9: $\lambda_3\varepsilon_2\varepsilon_1\gamma\gamma\varepsilon_0$ on Control, 0 suc on Stack.

Represents: (λx.f(fx) where f=λfλx.f(fx)

where h=λfλx.f(fx))suc 0

or (λx.(λfλx.f(fx))((λfλx.f(fx))x))suc 0 only reduction possible

At line 13: $xf\gamma f\gamma\varepsilon_3\gamma\varepsilon_0$ on Control, 0 on Stack.

Represents: (f(fx) where x=suc

where f=λfλx.f(fx))0

or (λfλx.f(fx))((λfλx.f(fx))suc)0 leftmost reduction

At line 16: $\lambda_3\varepsilon_4 f\gamma\varepsilon_3\gamma\varepsilon_0$ on Control, 0 on Stack.

Represents: (f(λx.f(fx) where f=suc) where x=suc

where f=λfλx.f(fx))0

or (λfλx.f(fx))(λx.suc(suc x))0 rightmost reduction

At line 20: $\lambda_3\varepsilon_5\varepsilon_3\gamma\varepsilon_0$ on Control, 0 on Stack.

Represents: (λx.f(fx) where f=(λx.f(fx) where f=suc)

where x=suc

where f=λfλx.f(fx))0

or (λx.(λx.suc(suc x))((λx.suc(suc x))x))0 only reduction possible

At line 24: $xf\gamma f\gamma\varepsilon_6\varepsilon_0$ on Control, 0 on Stack.

Represents: (f(fx) where x = 0

where f=(λx.f(fx) where f=suc)

or (λx.suc(suc x))((λx.suc(suc x))0) leftmost reduction

At line 27: $xf\gamma f\gamma\varepsilon_7 f\gamma\varepsilon_6\varepsilon_0$ on Control.

Represents: f(f(fx) where x=0

where f=suc) where x=0

where f=(λx.f(fx) where f=suc)

or (λx.suc(suc x))(suc(suc 0)) rightmost reduction

At line 35: $xf\gamma f\gamma\varepsilon_8\varepsilon_6\varepsilon_0$ on Control.

Represents: f(fx) where x=2

where f=suc

where x=0

where f=(λx.f(fx) where f=suc)

or suc(suc 2) only reduction possible

Thus the S.C.E. machine reduces 'selfapply twice suc 0' to 'suc(suc(suc(suc 0)))' in just 8 steps, whereas rightmost reduction also requires 8 steps, but leftmost reduction requires 9 steps. The reduction order used by the S.C.E. machine is called "blackboard order", because the old name for the S.C.E. machine was the blackboard machine, so called because it was used for teaching purposes using a blackboard[B1]. Hence the rules followed by blackboard order reduction are as follows:

1) Evaluate the operand of a combination before evaluating the operator.
2) Substitute the evaluated actual parameter of a λ-expression for the formal parameter before evaluating the λ-body.

It is noticeable that at several points of the computation, e.g. lines 28, 30 and 36, the value of the identifier f eventually retrieved from the environment is not the one most recently placed there, but an earlier one. For example, at line 28, the value of f is looked up, but the value retrieved is not the value most recently associated with f, namely that at line 20, but the value at line 16. This phenomenon has been known variously as 'The FUNARG binding problem', 'The environment problem', or 'The "Most recent" error', and considerable effort has been expended on solving this "problem". Indeed, articles about it still appear in the literature occasionally[M5]. Some versions of LISP have been incorrect due to retrieving the value most recently associated with a variable, and not chaining back through the static tree (or some equivalent mechanism). This same phenomenon also occurs in more advanced macro generators.

The final full example is one of string manipulation:

```
(let u(y)(z) = twice(adj(y))(rev(z))
 in twice(u("a"))("bc"))
 where twice(f)(x) = f(fx).
```

Here, adj is a function which has the same relationship to strings as prefix has to lists, and rev reverses a string. Thus,

adj(a) = λS.c-string(a,S),

and rev("drawer") = "reward".

The quotes enclose a string denotation. Thus quotes are retained on Control, which holds identifiers, but are not retained on the Stack, which holds values. This expression is

```
(λtwice.(λu.twice(u("a"))("bc"))(λyλz.twice(adj(y))(rev(z))))
 1       2                        3 4

                                          (λfλx.f(fx))
                                           5 6
```

or $\lambda_5\lambda_1\gamma$

$\lambda_1 \equiv \lambda$twice.$\lambda_3\lambda_2\gamma$

$\lambda_2 \equiv \lambda$u."bc""a"uγtwiceγγ

$\lambda_3 \equiv \lambda$y.λ_4

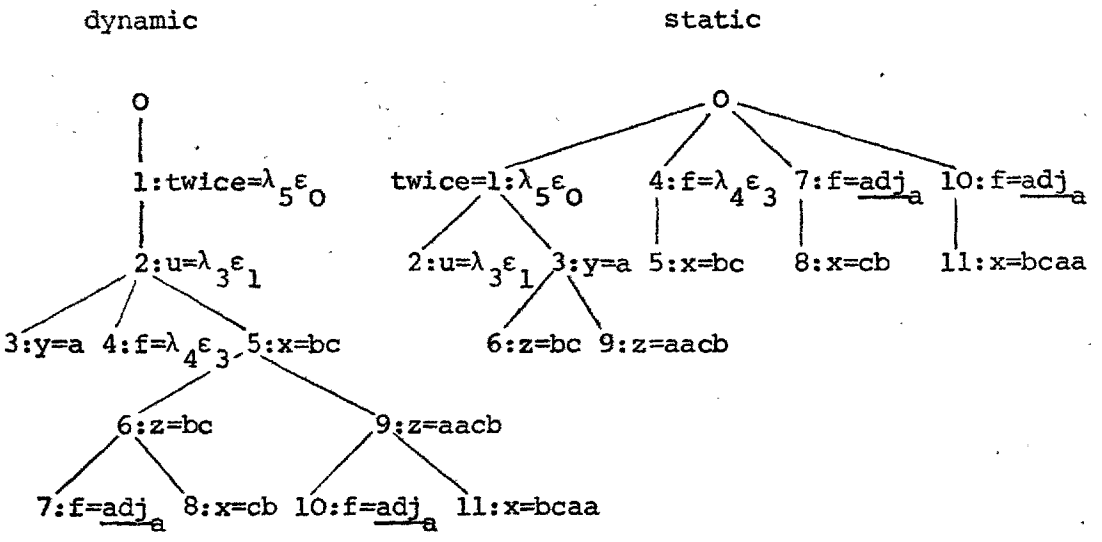
$\lambda_4 \equiv \lambda$z.z revγy adjγtwiceγγ

$\lambda_5 \equiv \lambda$f.λ_6

$\lambda_6 \equiv \lambda$x.xfγfγ

in control syntax. The computation is shown in figure A5, and the static and dynamic trees in figure 3.5.

fig 3.5



Because my machine is a reverse polish machine, it evaluates operands before operators. With reference to the expression $(\lambda x.L)M$, this corresponds to evaluating M once and for all, then looking up the value of M every time an x is encountered in L, rather than reevaluating M every time. This corresponds to the difference between call by value and call by name in ALGOL 60, and is likely to lead to a very much more efficient computation on the S.C.E. machine than on a normal-order evaluation machine. As an example, I consider the evaluation of the λ-expression

$$(\lambda f.f(f(7-2)))(\lambda u.u+u\times u).$$

In normal-order evaluation, the expression 7-2 is evaluated 9 times, with 8 subsequent arithmetic operations, but on the S.C.E. machine it is evaluated only once, with 4 subsequent arithmetic operations. In general, the number of operations performed by a normal-order machine increases exponentially with the length of the expression to be evaluated, whereas on the S.C.E. machine the number of operations increases arithmetically with the length of the expression: in the

expression $(\lambda f.f^n(7-2))(\lambda u.u+u\times u)$, a normal-order evaluation machine evaluates $7-2$ 3^n times, with 3^n-1 subsequent arithmetic operations, but the S.C.E. machine evaluates $7-2$ only once and performs only 2n subsequent arithmetic operations. The computation is shown in figure A6, where, from now on, I perform several lines of the computation at once, to save myself writing; I also dispense with the wavy underlining.

In all these examples only a small percentage of lines of the computation are actually calculation, the majority being "housekeeping" rearrangements. This is not atypical of the S.C.E. machine.

The S.C.E. machine may fail to compute a result from a legal input which has a value, for one of two reasons. The first is that if the result of a computation is a λ-expression, the machine will halt prematurely, with a λ-closure on the Stack. This λ-closure will be a λ-expression plus the environment in which it is to be evaluated. Thus on the computation of $(\lambda a.(\lambda x.x+(\lambda y.y\times a)3))2$ (see figure A7), the S.C.E. machine halts with $\lambda_2\varepsilon_1$ on the Stack, which represents

$$\lambda x.x+(\lambda y.y\times a)3 \text{ where } a=2,$$

or $\lambda x.x+(\lambda y.y\times 2)3.$

Thus the machine does not fully evaluate the expression to $\lambda x.x+6$. The reason for this is that in order to continue the evaluation to the subexpression '$(\lambda y.y\times 2)3$', the machine would have to break open the closure by a pseudo-apply operation, and output the string "λx." on an output tape. Another shortcoming in this area is the inability of the machine to handle a free variable. This can be overcome by using a unique-name generator, and putting a new, unique name on the Stack

whenever a free variable is encountered on Control. This proposal will only be of use if an output tape is added to the S.C.E. machine so that it can output λ-closures correctly. These additions have been made to Peter Landin's S.E.C.D. machine[L1] by C. McGowan[M4] to form the Modified S.E.C.D., or M.S.E.C.D., machine. I do not consider making such a modification here, since I am primarily concerned with λ-expressions which evaluate to constants which are not functions. I am surprised that some researchers have discovered these shortcomings of the machine only during attempts to prove it correct. The second and much more serious reason for the S.C.E. machine to fail to compute a result is that since my machine is not a normal-order evaluation machine, some computations which have a value can be expected not to terminate on it. I consider the computation of YF, for any λ-expression F which has a value on the S.C.E. machine, where Y is the fixed-point finder $\lambda f.(\lambda h.hh)(\lambda h.f(hh))$. The computation is shown in figure A8, and the machine is clearly in an inescapable loop. This is in some senses a disaster, since Y is the λ-expression I use to implement recursion. A solution to this problem will be given in Chapter 4.

3.2.3.

I now set about giving the function 'transform' which defines the S.C.E. machine. I must first define the objects which can appear on the Stack, Control and Environment of the machine. At any stage of the computation, the contents of the Stack, Control and Environment determine uniquely the state of the machine: the total configuration of the machine at this point, and all the information necessary to

continue the computation uniquely. Accordingly, I give a structure definition of a state, and a function 'transform', which maps a state into another state, called the next-state. I also give a function' 'initialise', which, given a λ-expression as argument, creates an 'initial state' of the machine, a state which is not the next-state of any state. Successive applications of transform to this initial state advance the computation step by step until, provided every step is defined (see below), the control component of the current state is null, at which point the computation is completed. Definition of a state:

```
state = <stack = <ε-markervstack-item>-list,
         control = <ε-markervcontrol-item>-list,
         environment = <top:<identifier,primitive-object>,
                        static-chain:environment,
                        dynamic-chain:environment>vPE>.
stack-item = primitive-objectv
             λ-closure = <λ-marker-part:λ-marker,
                          ε-marker-part:ε-marker>v
             stack-item-list.
control-item = identifierv
               λ-markerv
               applierv
               tupler = <tuplicity:nni>.
ε-marker = environment.
λ-marker = <bound-variable:identifier,
            λ-body:control-item-list>.
```

I thus have the following predicates, selectors and constructors, with abbreviations I shall use in brackers: is-environment(isenv),

is-stack-item, is-control-item, is-identifier(isid), is-primitive-object(isprimob), is-PE, is-λ-closure(isclosure), is-λ-marker(isλexp), is-applier(isγ) and is-tupler; s-stack, s-control, s-environment, s-top, s-identifier(id), s-primitive-object(primob), s-static-chain(sch), s-dynamic-chain(dch), s-λ-marker-part(λpart), s-ε-marker-part(εpart), s-tuplicity, s-bound-variable(bv) and s-λ-body(body); c-state, c-environment(c-env), c-top and c-λ-closure(c-closure). I shall avail myself of the facility for leaving the initial "s" off of selectors. Since one may have different machines with different PEs, I have a function 'val' which, when applied to a PE, gives a function from (primitive) identifiers to values (primitive objects) in that PE. I also have a function 'apply', which applies a primitive function to its argument list.

I can now give:

```
transform(state) =
let S = stack(state)
and C = control(state)
and E = environment(state)
in
(isid(hC) → c-state(lookup(hC,E):S,tC,E)
            where rec lookup(I,E) =
                        (isPE(E) → val(PE)(I),
                         id·top(E)=I → primob·top(E),lookup(I,sch(E))),
isλexp(hC) → c-state(c-closure(hC,E):S,tC,E),
istupler(hC) → c-state(list^n(S):t^nS,tC,E)
                        where n = tuplicity(hC),
isenv(hC) → c-state((htS=hC → hS:t^2S),tC,dch(E)),
```

```
isγ(hC) → (isprimob(hS) → c-state(apply(hS,htS):t²S,tC,E),
           isclosure(hS) → c-state(E':t²S,append(body·λpart·hS)(E':tC),E')
           where E' = c-env(c-top(bv·λpart·hS,htS),εpart·hS,E))).
```

And also:

```
initialise(X) = c-state(PE:(),makecontrol(X):PE:(),PE)
```

The function 'makecontrol' is not formalised, but takes a λ-expression and turns it into my reverse polish control syntax form.

The function transform may sometimes be undefined, for instance if isenv(hC) is true, and htS≠hC. Also, if a primitive object is "inapplicable" to its arguments, or if the arguments are not tupled into a list, the result of apply will be undefined. The clever programmer will doubtless be able to find many ways in which to sabotage the machine. This undefinedness corresponds to any physical realisation of the S.C.E. machine halting and displaying some error message at that point.

In order to demonstrate the compactness of representation afforded by not restricting myself to trees, I show transform in action on the example of figure A3, in figures B1 to B14. Not all the computation is shown, but enough to demonstrate the main points. In order to reduce the number of long arrows, an arrow marked with a λ- or ε-marker is to be considered as terminating on the node marked with the same marker. It is interesting to see how the static and dynamic trees are formed in the structure of the Environment. In all this activity, the only nodes which are created, rather than "shared", are those which are created to contain primitive objects.

3.2.4.

I now give an "extension" of the basic machine, which, as will appear, is already a part of the S.C.E. machine, but only in my treatment of primitive functions. I now allow a λ-expression to have a list of bound variables, and hence to take a list of arguments; the argument list must of course be the same length as the bound variable list. I allow the bound variable list and argument list of a λ-expression to be null. The alterations necessary to the S.C.E. machine are as follows. I must now tuple the arguments of a λ-expression before applying the corresponding λ-closure, even if the function is monadic or niladic (c.f. T_1 and T_0). This process is already catered for on the machine. I must also alter my definition of a λ-marker so that it reads:

λ-marker = <bound-variable-list:identifier-list,
λ-body:control-item-list>,

and of the top of an environment so that it reads:

top:<identifier,primitive-object>-list.

I now have the selector s-bound-variable-list(bvlist) instead of s-bound-variable.

Where before in the environment I had a name/value pair, I now have a list of name/value pairs (which may be null). Finally, I must alter the definitions of lookup and E' in transform. To give E' I will need one new list-processing function, 'unzip', defined as follows:

```
unzip(X,Y) = (isnull(X)∧isnull(Y) → (),
                  islist(X)∧islist(Y) → c-top(hX,hY):unzip(tX,tY)).
```

As examples of unzip at work:

```
unzip(nullist,nullist) = (),
```

and `unzip((x,y,z,w),(1,2,3,4)) = (<x,1>,<y,2>,<z,3>,<w,4>),`

representing a 'top' by angle brackets.

My new definitions are then:

```
lookup(I,E) = (isPE(E) → val(PE)(I),
               isnull·top(E) → lookup(I,sch(E)),
               id·h·top(E)=I → primob·h·top(E),
                               lookup(c-env(t·top(E),sch(E),dch(E)))),
```

and

```
E' = c-env(unzip(bvlist·λpart·hS,htS),εpart·hS,E).
```

This definition of lookup works properly, even when searching an environment containing a null environment layer.

I am now treating λ-closures in the same way as I treat primitive functions, tupling the arguments before applying. Thus one sees at last that applying a primitive function to its arguments is "really" applying the function's defining λ-closure to the argument list. A new environment is set up as usual, and the result, when (and if) computed, is copied down on to the head of the Stack when the machine exits from that environment. Since all this is hidden there is no way the machine can reenter that environment after exit, so there is no need for any working to be shown on the pencil and paper S.C.E. machine.

3.2.5.

I now extend the S.C.E. machine to deal with conditional statements. A first attempt to render $(p \rightarrow e_1,e_2)$ as a λ-expression might be to suppose that I have a function <u>if</u> such that for all expressions a_1 and a_2, $if(true,a_1,a_2) = a_1$ and $if(false,a_1,a_2) = a_2$. Then, $(p \rightarrow e_1,e_2)$ would be rendered in control syntax as

$$p\ e_1\ e_2\ T_3\ if\ \gamma.$$

However, this is not an adequate translation of $(p \rightarrow e_1,e_2)$, since both e_1 and e_2 are evaluated prior to choosing the required arm of the conditional. This can be disastrous, as in:

$$(x>0 \rightarrow sqrt(x),sqrt(-x)),$$

which will always be undefined if one evaluates both arms. Hence I must "freeze" each arm of the conditional, and "unfreeze" the success arem after it has been chosen. Since λ-markers are not evaluated when they appear at the head of Control, but are formed into a λ-closure, I can do this by turning each arm of the conditional into a λ-expression with a null bound variable list. I can then unfreeze later by applying the λ-closure to the null argument tuple. Instead of using the triadic function 'if' given above, in my second attempt I allow an 'if' to appear on Control, with the same status as γ and the tuplers. When an if is at the head of Control, then if the head of the Stack is true, it chooses the second element on the Stack and throws away the third, otherwise if the head of the Stack is false, it chooses the third element on the Stack and throws away the second. So, if I represent the null bound variable list by "π", then $(p \rightarrow e_1,e_2)$ is translated into reverse polish as:

$$T_0\ (\lambda\pi.e_2)\ (\lambda\pi.e_1)\ p\ \text{if}\ \gamma.$$

I run $\lambda x.(x>0 \rightarrow xxx,x-1)5$ just to illustrate conditional statements on the S.C.E. machine. The expression in control syntax is

$$5T_1\lambda_1\gamma$$

$$\lambda_1 \equiv \lambda x.T_0\lambda_3\lambda_2 x O T_2 > \gamma \text{if} \gamma$$

$$\lambda_2 \equiv \lambda\pi.xxT_2{}^x\gamma$$

$$\lambda_3 \equiv \lambda\pi.xlT_2{}^-\gamma.$$

The computation is shown in figure A9.

This extension requires two more lines in the definition of the S.C.E. machine, which I now give. In the definition of a control-item I must add:

...vchoicev...

which gives the predicate is-choice(isif). In the definition of transform, I must include the line:

"isif(hC) → c-state((hS=true → htS,hS=false → ht^2S):t^3S,tC,E),".

When isif(hC) is true, I make sure that hS is a truth-value by testing it for equality with both true and false.

Chapter four

The I.S.C.E.M.O. machine

4.

In this chapter, I extend the S.C.E. machine to the I.S.C.E.M.O. machine. A method of translating ALGOL 60 programs into the λ-calculus will be given informally, mostly by example, and I shall show how to run these programs on the I.S.C.E.M.O. machine. For those people who already have an understanding of the λ-calculus, this should shed light on the mechanisms of ALGOL 60; contrariwise, for those people who have a thorough knowledge of ALGOL 60, this should illuminate the processes of the λ-calculus.

The one feature of ALGOL 60 which I do not consider at all, and allow only this mention on the doorstep, is the apparent possibility of entering a compound statement by a goto, and exiting from that statement naturally. It will in due course become obvious that this causes complications far beyond what one would expect. These difficulties add to my belief that the above-mentioned construction should not be part of ALGOL 60 at all. I support this claim with part of section 4.1.3 from the 'Revised report on ALGOL 60', namely "A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest enclosing block, ...". This coupled with the first two paragraphs of section 5. would then make it illegal to enter a compound statement by a goto. In all events, jumping past a begin is against the whole spirit of the language.

4.1.

The most prominent feature of any programming language compared with the λ-calculus is the possibility of assignment to a variable. Despite the obvious advantages of a system including assignment, it does not however seem to be fundamental, for

```
begin integer x;
      x := 5;
      x := 2×x;
      print(x+1)
end
```

corresponds well enough to

```
let x=5
in let x=2×x
   in print(x+1),
```

and so to (λx.(λx.print(x+1))(2×x))5,

where here each "assignment" establishes a new environment. This removal of assignment statements can be carried to extreme lengths, indeed, assignment seems not to be necessary at all (see B3). However, such methods do not lead to efficient computations on the S.C.E. machine; furthermore, they make it hard to see the wood for the trees. That is, ALGOL programs are not made more perspicuous by eliminating all assignment statements. Hence, any machine which attempts to model real programming languages must have some provisions for a memory in which values may be stored and retrieved. Consequently, I add to my machine a Memory component, which is an indefinitely extensible sequence of

locations with addresses l_1, l_2, l_3, ... Since the only (non-primitive) identifiers used on the S.C.E. machine are bound variables, it need raise storage only when a λ-closure is applied. Thus, when a λ-closure is applied on the (now) S.C.E.M. machine, the bound variables are associated with addresses of locations, and in those locations are stored the values of the arguments. I illustrate this by translating my first ALGOL 60 program into control syntax, and running it on the S.C.E.M. machine.

Consider the program:

```
begin boolean b;
      b := false                                    (1)
end.
```

Unlike other forms of declaration in ALGOL 60, the declaration of b is uninitialised. Since there must be an argument to apply a λ-closure to, I initialise b to the undefined object, which I represent by "Ω". Also, I let the "type" of b look after itself; alternatively, b may be considered to be of type general, or of mode union(bool,int,real c etc. c). I use "←" on Control as the assignment operator. A ← at the head of Control has two effects. Firstly, a "side effect" occurs: the head of the Stack is assigned to the location whose address is second on the Stack. A side effect is defined to be a transformation of any component other than S or C. Secondly, a result is produced, just as in any functional application: the head of the Stack is copied down into the second position, overwriting the previous contents. Now, (1) can be rendered as

let b=Ω.

in b←false,

and so as (λb.b←false)Ω,

and hence in control syntax (1) is

$$\Omega T_1 \lambda_1 \gamma$$

$$\lambda_1 \equiv \lambda b.b\ \text{false}\leftarrow.$$

This trivial program is run in figure A10. At line 2, the λ-closure is to be applied. I enter the new environment in the normal way, except that in the environment I associate b with the address l_1 rather than with the value Ω. In the Memory column I create the location with address l_1, and initialise this location to Ω, ignoring the fact that this line is crossed through, for the moment. I now continue in the normal way, until a ← is at the head of Control, at line 5. I then assign the head of the Stack to the location whose address lies second on the Stack. This is done by crossing out the existing line "l_1 =..." and writing in a new one "l_1=false". Furthermore, the head of Stack is copied down one place in the Stack, overwriting the address l_1 (this copes with multiple assignments). I illustrate further with the program:

```
begin integer x,y;
      x := 0;                                    (2)
      y := x := x+1
end.
```

This shows multiple assignment at work, and also the necessity for a "contents of" operator (ς), and the necessity for a "voiding" operator (ν). The program corresponds to

let (x,y)=(Ω,Ω)

in x := 0; y := x := x+1,

and so to (λ(x,y).x := 0; y := x := x+1)(Ω,Ω).

Hence, in control syntax it is

$$\Omega\Omega T_2\lambda_1\gamma$$

$$\lambda_1 \equiv \lambda(x,y).x0\leftarrow\nu yxx\mathfrak{c}1T_2+\gamma\leftarrow\leftarrow.$$

This is run in figure A11. At line 5, ← is at the head of Control, and 0 is assigned to 1_1. At line 6, the zero left by the previous instruction is not wanted, so it is thrown away by the voider ν. At line 10, the contents-of operator $\mathfrak{c}$ looks up the value currently in location 1_1, and this value is placed on the Stack in place of the address. The multiple assignment is shown at lines 13 and 14. This example shows how long control strings become when one translates several ALGOL statements at once. To overcome this, I translate (2) as

$$\Omega\Omega T_2\lambda_1\gamma$$

$$\lambda_1 \equiv \lambda(x,y).x0\leftarrow\nu\delta_2$$

$$\delta_2 \equiv yxx\mathfrak{c}1T_2+\gamma\leftarrow\leftarrow,$$

where I have now added δ-markers to my list of possible control items. When a δ-marker reaches the head of Control, it is simply replaced by its body, and evaluation continued. This computation is shown in figure A12. At line 6 the δ-marker is at the head of Control, and is replaced by its body, shown in line 7.

4.2.

So far, my programs have not seemed to do much. That is, they accepted no input, and gave no output. Quite a considerable amount of work was done, however, if one just considers the transformation of the four components S, C, E and M. From now on, I link my machine to the outside world by the addition of an Input and Output component, turning it into the I.S.C.E.M.O. machine. The Input and Output components are lists of concrete representations of internal objects, and are altered by the "read" and "print" operators. The occurrence of read at the head of Control causes the head of the Input to be removed, and the corresponding internal object placed on the Stack. The occurrence of print at the head of Control causes the head of the Stack to be removed, and a corresponding concrete representation of that primitive object placed on the Output. I illustrate by running the program:

begin print(("result = ",read+read)) end (3)

with input (534,486).

I consider (3) as

begin $_1$print("result = ");
$_2$print(read+read)
end

and hence in control syntax as

$$\delta_1 \equiv \text{"result = "print}\delta_2$$

$$\delta_2 \equiv \text{read read } T_2+\gamma\text{print},$$

where I gave the beginning of each command an integer suffix for ease of translation into control syntax. This program is run in figure A13, where for my convenience I show the contents of the Input only once, and the components of the Output only as they are printed. The next position to be read from or printed to is shown by an upward arrow; the end of the input is shown by " " and the beginning of the output by " ". I now have virtually all the machinery I need to model the whole of ALGOL 60 on my I.S.C.E.M.O. machine. For example, the following program involves no new principles:

```
1begin 2integer a,b;
       b := a := read;
       3begin 4integer a;
              a := read;                    (4)
            5b := b+a
       end;
       6print(b+a)
end.
```

I give subscripts to the statements according to the following scheme: Number (from 1) the start of each block and the start of each statement, except the first in a block (this gives δ-markers), and each "type declaration" (this gives λ-markers). Now, (4) is equivalent to

```
1let 2(a,b)=(Ω,Ω)
 in b := a := read;
   3let 4a=Ω
    in a := read;
      5b := b+a;
   6print(b+a);
```

and to $_1(_2\lambda(a,b).b := a := read;\ _3(_4\lambda a.a := read;\ _5b := b+a)\Omega;$

$_6print(b+a))(\Omega,\Omega)$,

and so to $\delta_1 \equiv \Omega\Omega T_2\lambda_2\gamma$

$\lambda_2 \equiv \lambda(a,b).b\ a\ read\leftarrow\nu\delta_3$

$\delta_3 \equiv \Omega T_1\lambda_4\gamma\delta_6$

$\lambda_4 \equiv \lambda a.a\ read\leftarrow\nu\delta_5$

$\delta_5 \equiv b\ b\mathfrak{C}a\mathfrak{C}T_2+\gamma\leftarrow\nu$

$\delta_6 \equiv b\mathfrak{C}a\mathfrak{C}T_2+\gamma print.$

With a little practice, one can go straight from a subscripted ALGOL program to the control syntax form without any intermediate steps. I shall do this in future. The program is run with input (3,5) in figure A14. It is of crucial importance that the λ-calculus machinery automatically takes care of some of the semantic features of ALGOL 60. In the outer block, all references to a are to the a declared there, and not to the a in the inner block. Likewise in the inner block. Also, even if a were not declared in the outer block, the a in the inner block would still only be accessible from inside the inner block.

I have already dealt with conditionals in section 3.2.5., so it is easy to translate the ALGOL if-statement into control syntax.

if condition then expression1 else expression2;

translates into

$$T_0\lambda_2\lambda_1\text{condition if}\gamma,$$

$$\lambda_1 \equiv \lambda\pi.\text{expression1}$$

$$\lambda_2 \equiv \lambda\pi.\text{expression2},$$

and

if condition *then* expression;

translates into

$$T_0\lambda_2\lambda_1\text{condition if}\gamma,$$

$$\lambda_1 \equiv \lambda\pi.\text{expression}$$

$$\lambda_2 \equiv \lambda\pi.$$

I illustrate by running

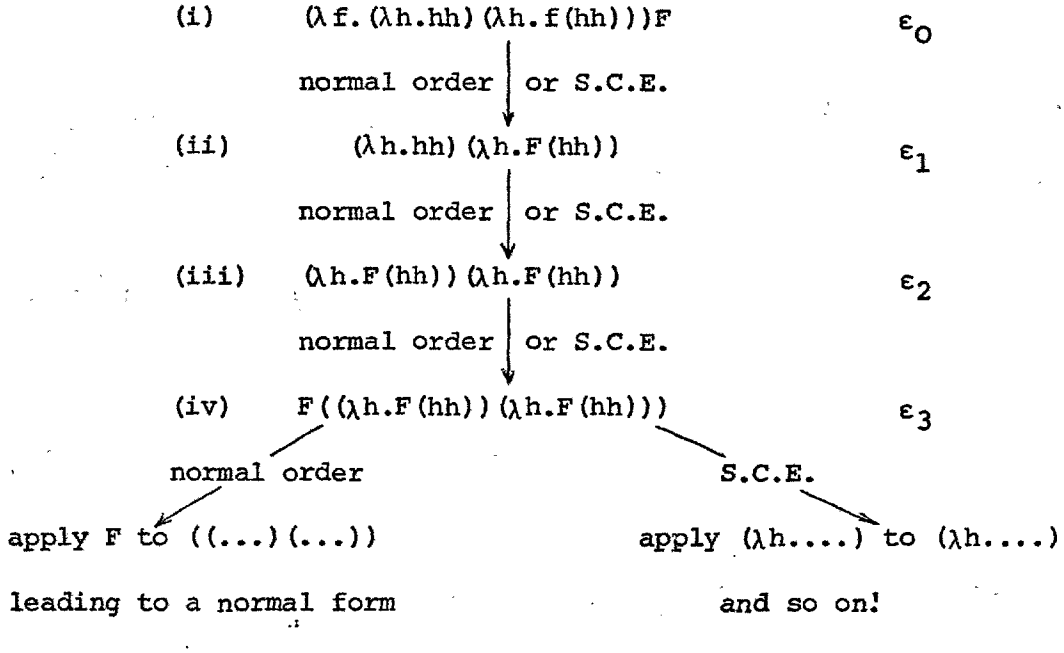
$$_1\underline{\text{begin}}\ \text{print}(\underline{\text{if}}\ \text{read}\ \underline{\text{then}}_2\text{read}\ \underline{\text{else}}_3 0)\ \underline{\text{end}},$$

with inputs (false,2) and (true,2) on the I.S.C.E.M.O. machine in figures A15a and A15b, where, from now on, I show only those components of the machine actually involved in a computation.

4.3.

I now come to the recursive definition of functions, and how to implement functions so defined. As already seen in the previous chapter, the computation of YF for any λ-expression F, where Y is the fixed-point finder $\lambda f.(\lambda h.hh)(\lambda h.f(hh))$, always fails to terminate on the S.C.E. machine, and hence also on the I.S.C.E.M.O. machine. The computation on the I.S.C.E.M.O. machine is shown in figure A16.

I contrast the normal order and S.C.E. methods of reduction:-



A first attempt to accommodate recursion on the I.S.C.E.M.O. machine is to prevent the machine from becoming trapped in the infinite loop by altering Y to Y', where Y' = λf.(λh.hh)(λh.f(λx.hhx)). These two λ-expressions are equivalent, because for any λ-expression Q, $(\lambda x.hhx)Q =_{\beta} (hh)Q$. Now, line (iv) above becomes

$$F(\lambda x.(\lambda h.F(\lambda x.hhx))(\lambda h.F(\lambda x.hhx))x),$$

and so the entire expression to the right of the first F is frozen by the λx., and F is applied to (λx.(...)(...)) as in normal-order reduction. This new Y', however, although it deals with most cases will still cause the I.S.C.E.M.O. machine to loop indefinitely with certain forms of F. For instance, if F is the factorial function disguised as:

$$\underline{\text{let rec}}\ f = (\lambda n.(n=0 \rightarrow 1, c \times n \times f(n-1)))\ \underline{\text{where}}\ c = f0),$$

then, abbreviating $\lambda c\lambda n.(n=0 \rightarrow 1, c\times n\times f(n-1))$ by G, $\lambda h.F(\lambda x.hhx)$ by λ_1, $\lambda x.\lambda_1\lambda_1 x$ by λ_2 and $S^f_{\lambda_2} G$ by G', then

$$F = \lambda f.G(f0)$$

and

$$\begin{aligned} Y'F &>_\beta (\lambda h.hh)(\lambda h.F(\lambda x.hhx)) = (\lambda h.hh)\lambda_1 \\ &>_\beta \lambda_1\lambda_1. \end{aligned}$$

Now,

$$\begin{aligned} \lambda_1\lambda_1 &= (\lambda h.F(\lambda x.hhx))\lambda_1 \\ &>_\beta F(\lambda x.\lambda_1\lambda_1 x) = F\lambda_2 \\ &= (\lambda f.G(f0))\lambda_2 \\ &>_\beta G'(\lambda_2 0) = G'(\lambda_1\lambda_1 0). \end{aligned}$$

But now, the I.S.C.E.M.O. machine tries to evaluate the operand, so one has

$$\begin{aligned} Y'F &>_\beta G'(\lambda_1\lambda_1 0) \\ &>_\beta G'(G'(\lambda_1\lambda_1 0)0) \\ &>_\beta \ldots \text{ and so on.} \end{aligned}$$

I can cure this in this case by using Y", where

$$Y'' = \lambda f.(\lambda h.hh)(\lambda h.f(\lambda x\lambda y.hhxy)).$$

However, the factorial function disguised now as the 2-place function

$$\underline{\text{let}}\ \underline{\text{rec}}\ f = (\lambda n_1\lambda n_2.(n_1=0 \rightarrow 1, c\times n_1\times f(n_1-1,n_2))\ \underline{\text{where}}\ c = f00)$$

will cause the computation to loop indefinitely.

In general, if one uses $Y^{(k)}$, where

$$Y^{(k)} = \lambda f.(\lambda h.hh)(\lambda h.f(\lambda x_1 \ldots \lambda x_k.hhx_1 \ldots x_k),$$

then the factorial function disguised as the k-place function

$$\underline{\text{let rec}}\ f = (\lambda n_1 \ldots \lambda n_k.(n_1=0 \rightarrow 1, c \times n_1 \times f(n_1-1, n_2, \ldots, n_k))\ \underline{\text{where}}\ c = f0 \ldots 0)$$

causes the computation to loop indefinitely.

As a second attempt to accommodate recursion on the I.S.C.E.M.O. machine, I try an imperative form of Y. That is, I try 'Ω' as a first "guess" to the fixed point YF of f, and then assign the true fixed point to f. Thus I define Y* by

$$Y^*(F) = \underline{\text{let}}\ f = \Omega\ \underline{\text{in}}\ (f := F);\ f,$$

i.e. I replace

$$\underline{\text{let rec}}\ f=F\ \underline{\text{in}}\ N$$

by $$\underline{\text{let}}\ f=(\underline{\text{let}}\ f=\Omega\ \underline{\text{in}}\ f := F;\ f)\ \underline{\text{in}}\ N,$$

or $$(\lambda f.N)(Y\lambda f.F) \equiv (\lambda f.N)((\lambda f.f := F;\ f)\Omega).$$

This, again, deals with most cases, but some forms of F will still cause the I.S.C.E.M.O. machine to loop. If, for instance, F contains f outside of a λ-marker, then an attempt will be made to look up the value of f before the assignment has been made, i.e., while the location coupled with f still holds Ω. For instance, using the F of the previous example,

$$\underline{\text{let rec}}\ f = G(f0)\ \underline{\text{in}}\ f2$$

becomes

$$(\lambda f.f2)((\lambda f.f := G(f0);\ f)\Omega),$$

or $$((\lambda f.f := G(f0);\ f)\Omega)2.$$

This is $((_1\lambda f.f := (_2\lambda c_3\lambda n.(n=0 \rightarrow 1, c\times n\times f(n-1)))(f0);\ f)\Omega)2,$
and the computation is run in figure A17. The necessity of computing f0 causes f to be looked up before the value $G(f0)$ $(\lambda_2\varepsilon_1)$ can be assigned to it.

As a third attempt, I look again at the possibility of freezing the operand that causes all the trouble: $\lambda h.f(hh)$. I may transform this into $\lambda h.f(\lambda\pi.hh)$, but now, unlike the previous two attempts, I must also alter the function under consideration, by replacing all occurrences of f there by $T_0f\&\gamma$, to unfreeze the $(\lambda\pi.hh)$. I show the computation of

<u>let rec</u> f = $\lambda n.(n=0 \rightarrow 1, n\times f(n-1))$
<u>in</u> f1,

i.e. $(\lambda f.f1)(Y\lambda f\lambda n.(n=0 \rightarrow 1, n\times f(n-1)))$,

or $_1(_2\lambda f.(_3\lambda h.hh)(_4\lambda h.f(_5\lambda\pi.hh)))(_6\lambda f_7\lambda n.(n=0 \rightarrow {}_8 1, {}_9 n\times f(n-1)))1,$

in figure A18.

Of the three methods considered so far, only the third has been able to guarantee a non-looping computation when one exists. Also, all methods are tediously inefficient. Since I have lost the full power of the λ-calculus in choosing an efficient order of evaluation, I may as well see if I can alter the evaluation mechanism of the machine to accommodate recursion, rather than try to use a Y-like λ-expression. I try running the program

let rec f = λn.(n=0 → 1,n×f(n-1))

in f2,

i.e. $_1(_2\lambda f.f2)(_3\lambda n.(n=0 \rightarrow {}_4 1, {}_5 n\times f(n-1)))$,

without bothering with recursion, to see what goes wrong on the machine. I know in advance that it will try to interpret the f in the definiens of rec f = λn.(n=0 → 1,n×f(n-1)) as an already known f, different from the f of the definiendum. The computation is shown in figure A19. Here, at line 12, I try to look up the value of f, and can find no such variable in the current environment. In environment 1, I have the information $f=1_1$, and $1_1=\lambda_3\varepsilon_0$; thus, when I come to apply f at line 4, the new environment (2) gets linked back to environment 0, whereas what I need is environment 0 plus the value of f, $f=1_1$. This persuades me to use a special type of λ, λ_{rec}, which causes the ε-part of a λ-closure to be updated to the current environment. The computation is repeated in figure A20, and this time all goes well; the λ-closure bound to f not only contains the environment in which all non-local variables of f must be evaluated, but also the environment layer containing f itself.

Changing the state of the I.S.C.E.M.O. machine at the line marked (*) back to the λ-expression it represents, I have

n×f(n-1) where n = 2

where f = (λn.(n=0 → 1,n×f(n-1))

where f = (λn.(n=0 → 1,n×f(n-1))

where f = (λn.(n=0 → 1,n×f(n-1))

where f = ...

Here, ε_1, the ε-marker part of the λ-closure $\lambda_3\varepsilon_1$, is expanded indefinitely. In order to obtain a finite representation of this infinite sequence of substitutions, I suppose that I have a λ-expression X such that

$$X \equiv \lambda n.(n=0 \rightarrow 1, n\times f(n-1)) \text{ where } f = (\lambda n.(n=0 \rightarrow 1, n\times f(n-1)) \text{ where } f = \ldots.$$

Then, $X \equiv (\lambda n.(n=0 \rightarrow 1, n\times f(n-1)) \text{ where } f = X)$,

or, $X \equiv (\lambda f \lambda n.(n=0 \rightarrow 1, n\times f(n-1)))X$.

Hence, the λ-expression representing X is none other than

$$Y\lambda f\lambda n.(n=0 \rightarrow 1, n\times f(n-1)),$$

and so the machine state represents the λ-expression

$$2\times(Y\lambda f\lambda n.(n=0 \rightarrow 1, n\times f(n-1)))1,$$

which is the original λ-expression $(\lambda f.f2)(Y\lambda f\lambda n.(n=0 \rightarrow 1, n\times f(n-1)))$ after three blackboard reductions, discounting $YF =_\beta S^f_{(YF)}(Ff)$ as a reduction, and counting choosing the success arm of a conditional as one blackboard reduction. Thus this "cheat" correctly implements recursion on the I.S.C.E.M.O. machine (and hence also on the S.C.E. machine). The theoretical reduction rule corresponding to λ_{rec} would be:

$$(\lambda_{rec} x M) >_\beta S^x_{(\lambda_{rec} x M)} M,$$

after a possible α-conversion to remove bound/free variable clashes.

4.3.1.

In ALGOL 60, a label constitutes an implicit declaration of a variable. The variable is initialized to the line of text immediately following the label. Since this line of text may contain references to the label preceding it, all label declarations are recursive. I handle this on the I.S.C.E.M.O. machine by initialising a label to a δ-closure, which is a package similar to a λ-closure, but containing a δ-marker in place of a λ-marker. Just as a λ-closure contains all the information necessary to apply a λ-expression, so a δ-closure contains all the information necessary to continue the computation from a δ-marker. I now require δ-markers to appear in the Control as operands. I use capital δ's (Δ) in the Control, as an ordinary δ would be replaced by its body on reaching the head of Control. A jump instruction is represented on Control by 'J'. The effect of a J at the head of Control is demonstrated in the following example.

```
begin integer a;
      a := 0;
   1:if read then goto k;
     begin boolean b;
           b := read;                    (5)
           a := a+if b then 1 else 2;
       k:if b=read then goto k else goto 1
     end;
   k:print(a)
end.
```

I amend my method of giving subscripts to the statements as follows. I first of all add declarations for all labels in the program. Then I number (from 1) the start of each block and the start of each statement, except the first in a block. I fill out each one-armed conditional with a dummy arm, and subscript each arm. I also number each set of non-recursive declarations, and each set of recursive declarations. Last of all, I turn each declaration into an initialised declaration. The program (5) then becomes:

```
1begin2integer a=Ω; 3label (k,1)=(δ16,δ4);
        a := 0;
     1:4if read then5goto k else6;
       7begin8boolean b=Ω;  9label k=δ13;
             b := read;
          10a := a+if b then11 1 else12 2;
        k:13if b=read then14goto k else15goto 1
       end
    k:16print(a)
end.
```

I can now give the control syntax for (5):

$\delta_1 \equiv \Omega T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda a.\Delta_{16}\Delta_4 T_2 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec}(k,1).a0 \leftarrow \nu\delta_4$

$\delta_4 \equiv T_0 \lambda_6 \lambda_5$ read if$\gamma\delta_7$

$\lambda_5 \equiv \lambda\pi.k \& J$

$\lambda_6 \equiv \lambda\pi.$

$\delta_7 \equiv \Omega T_1 \lambda_8 \gamma$

$\lambda_8 \equiv \lambda b.\Delta_{13} T_1 \lambda_9 \gamma$

$\lambda_9 \equiv \lambda_{rec}k.b\ read{\leftarrow}\nu\delta_{10}$

$\delta_{10} \equiv aa\mathfrak{C}T_0\lambda_{12}\lambda_{11}b\mathfrak{C}if\gamma T_2{+}\gamma{\leftarrow}\nu\delta_{13}$

$\lambda_{11} \equiv \lambda\pi.1$

$\lambda_{12} \equiv \lambda\pi.2$

$\delta_{13} \equiv T_0\lambda_{15}\lambda_{14}b\mathfrak{C}readT_2{=}\gamma if\gamma$

$\lambda_{14} \equiv \lambda\pi.k\mathfrak{C}J$

$\lambda_{15} \equiv \lambda\pi.1\mathfrak{C}J$

$\delta_{16} \equiv a\,\mathfrak{C}\ print.$

I run this program with input (false,false,false,true,true) in figure A21. At line 3, I find a Δ-marker at the head of Control. This is removed, and the corresponding δ-closure placed on the Stack. Likewise, with the Δ-marker Δ_4 at line 4. At line 6 a λ-closure is applied. The λpart of this closure is introduced by a λ_{rec}, so the εparts of the δ-closures are updated to the current environment. Again, at line 19 a λ-closure is applied, and the δ-closure updated. The first J appears at the head of Control at line 40, when the head of Stack is the δ-closure $\delta_{13}\varepsilon_5$. The effect is to remove everything above the ε-marker ε_5 in the Stack and Control, and then to install the body of the δ-marker δ_{13} in the Control. A similar action occurs when a J is at the head of Control at lines 47 and 53.

4.3.2.

The ALGOL 60 procedure also constitutes an initialised recursive declaration of a variable. The natural way to render an expression such as

```
procedure f(x); value x; integer x;
   print(x+1);
```

in control syntax is to consider the declaration to be equivalent to

```
let f = λx.print(x+1)
in ....
```

I try running the program

```
begin integer a,b;
      procedure f(x); value x; integer x;
         b := x+2↑x;
      a := read;                                   (6)
      f(a×a);
      print(b)
end
```

on the I.S.C.E.M.O. machine. I first of all subscript the program, and initialise all declarations.

```
1begin2integer (a,b)=(Ω,Ω);
      3procedure f=4λx.b := x+2↑x;
       a := read;
      5f(a×a);
      6print(b)
end
```

The control syntax for (6) is then

$\delta_1 \equiv \Omega\Omega T_2 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda(a,b).\lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec} f.a\ read \leftarrow \nu \delta_5$

$\lambda_4 \equiv \lambda x.bx\mathfrak{G}2x\varepsilon T_2 \uparrow \gamma T_2 + \gamma \leftarrow \nu$

$\delta_5 \equiv a\mathfrak{G}a\varepsilon T_2 x \gamma T_1 f\mathfrak{G}\gamma\delta_6$

$\delta_6 \equiv b\mathfrak{G}print.$

This program is run with input (3) in figure A22, and my first intuitive approach is seen to be correct. At line 4, f is bound to the address of the updated λ-closure $\lambda_4\varepsilon_2$, which represents the body of the procedure and the environment in which it is to be evaluated. Furthermore, when f is applied at line 12 the identifier x is local to the procedure body, and is initialised to the value 9. To underline the point that x is local to the procedure, I run the trivial program

```
1begin 2integer i;
      3procedure p(i); value i; integer i;  4i := i+1;
       i := 0;
      5p(i);
      6print(i)
end
```

in figure A23, where one sees that 0 is printed, in accordance with the ALGOL 60 semantics that an assignation to a parameter called by value does not alter the actual parameter supplied in the call. In order to run the corresponding program in which i is called by name, i.e.

```
1begin 2integer i;
      3procedure p(i); integer i; 4i := i+1;
       i := 0;
      5p(6i);
      7print(i)
end
```

(which prints 1), I must somehow arrange that the i in the procedure body is bound not to a new location, but to the outer i. This suggests somehow "freezing" the i passed as a parameter, and "unfreezing" it inside the procedure. I can do this by turning the i in the call into a λ-marker (which explains the extra subscript in this case), and code (7) as

$\delta_1 \equiv \Omega T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda i.\lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec} p.i0 \leftarrow \nu \delta_5$

$\lambda_4 \equiv \lambda i.T_0 i\epsilon\gamma T_0 i\epsilon\gamma \epsilon\, 1T_2 + \gamma \leftarrow \nu$

$\delta_5 \equiv \lambda_6 T_1 p\epsilon\gamma\delta_7$

$\lambda_6 \equiv \lambda\pi.i$

$\delta_7 \equiv i\epsilon print.$

Here, I replace the i in the call p(i) by $\lambda\pi.i$, and the i in the procedure body i := i+1 by $T_0 i\epsilon\gamma$. I run this in figure A24. As one sees, 1 is printed and all is well.

The intuitive approach for parameterless procedures is the correct approach, as the following example shows. The program

```
begin integer y;
      procedure g; y := read+read;
      g;
      print(y)
end
```

corresponds fairly clearly to

$$(\lambda y.(\lambda g.g();\ \text{print}(y))(\lambda\pi.y := \text{read+read}))\Omega,$$

and hence to

$\delta_1 \equiv \Omega T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda y.\lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec} g.T_0 g \varepsilon \gamma \delta_5$

$\lambda_4 \equiv \lambda\pi.y$ read read $T_2 + \gamma \leftarrow \nu$

$\delta_5 \equiv y \varepsilon$ print.

I run this program in figure A25 with input (2,3).

I can now give the rules for translating (non-type) procedures

(i) Procedure declarations (of procedure f, say)

a) if f has parameters $x_1, x_2, \ldots, x_n$, then put

$f = \lambda(x_1, x_2, \ldots, x_n)\ldots.$

b) if f is parameterless, then put $f = \lambda\pi\ldots.$

The λ-body is the translated body of the procedure, with each occurrence of x_1 replaced by $T_0 x_i \varepsilon \gamma$ if x_i is called by name.

(ii) Procedure calls (of procedure f, say)

a) if f has actual parameters $a_1, a_2, \ldots, a_n$, then translate $f(a_1, a_2, \ldots, a_n)$ as $b_1 b_2 \ldots b_n T_n f \varepsilon \gamma$, where b_1 is $a_1 \varepsilon$ if a_1 is called by value, and $\lambda\pi.a_1$ if a_1 is called by name

(with obvious alterations if a_1 is an expression).

b) if f is parameterless, translate f as $T_0 f \xi \gamma$.

In my subscripting scheme, as I have already noted, I must now also subscript each parameter called by name in a procedure call. I now run

```
begin boolean x;
      procedure p(a); boolean a; if read then p(a) else q(a);
      procedure q(a); boolean a; if read then begin p(a);
                                                a := ~a end;
      x := read;
      p(x);
      print(x);
end,
```

or, partially translated,

```
1begin2boolean x=Ω;
      3procedure (p,q) = (4λa.if read then 5p(6a) else 7q(8a),
                        - 9λa.if read then begin 10p(11a);
                                                 12a := ~a end else13);
      x := read;
     14p(15x);
     16print(x)
end,
```

in figure A26, with input (true,false,true,false,false). When I translate p(a) and q(a), because a is a parameter called by name, I replace a by $T_0 a \xi \gamma$. But then, since a is the actual parameter of a procedure which calls its parameter by name, I further replace it

by $\lambda\pi.T_0 a \mathcal{C} \gamma$. This example will appear in a much more streamlined and efficient form in section 5.1..

I now deal with type procedures. A type procedure is exactly the same as a non-type procedure, with the exception that one may assign a value of the required type to the procedure identifier. Now, I obviously cannot translate

```
begin integer x;
      integer procedure p(x); value x; integer x;
         p := x+1;                                        (8)
      x := 0;
      print(p(x)+p(x))
end
```

as $(\lambda x.(\lambda p.x := 0;\ \text{print}(p(x)+p(x)))(\lambda x.p := x+1))\Omega$,

since if I did, the assignation of x+1 to p during the first call would destroy the value of p, and I would then be unable to apply p at the second call. I accordingly set up a variable local to the procedure body, p#, and assign to that variable. Thus I translate (8) as

$$(\lambda x.(\lambda p.x := 0;\ \text{print}(p(x)+p(x)))(\lambda x.(\lambda p\#.p\# := x+1; p\#)\Omega))\Omega.$$

As an illustration, I run

```
begin integer z;
      integer procedure x;
      begin z := z+1; x := z end;
      z := 0;
      print(x=x)
end,
```

or

```
1begin2integer z=Ω;
      3integer procedure x=4λπ.(5integer x#=Ω; z := z+1;
                                                6x# := z; 7x#);
       z := 0;
      8print(x=x)
end,
```

on the I.S.C.E.M.O. machine in figure A27. Here, I have to use the property that when Control is exhausted, the machine copies down the head of the Stack to yield the value delivered by the procedure x. This is at lines 17 and 28.

4.4.

I now give a series of 7 examples of varying complexity (in figures A28-A34), to show the various aspects of the running of programs, and the translation of ALGOL 60 programs into control syntax form. Each program is given first, followed by its partial translation.

example 1.

```
1:begin procedure p(x); label x; 1:goto x;
        p(1);
      1:
end
1begin2label 1=δ3;
      3begin4label 1=δ9;
            procedure p=5λx.(6label 1=δ7; 7goto x);
            p(81)
          1:9
       end
end.
```

example 2.

```
begin boolean a,b;
      procedure p(x,y); value y; boolean x,y;
         x := y := x⊃y;
      a := b := false;
      p(a,b);
      print((a,b))
end
```

```
1begin2boolean (a,b)=(Ω,Ω);
      3procedure p = 4λ(x,y).x := y := x⊃y;
       a := b := false;
      5p(6a,b);
      7print(a); print(b)
end
```

example 3.

```
begin integer procedure p(x,y); value x,y; integer x,y;
         p := if y=1 then x else p(x,y-1)↑x;
      print(p(read,read))
end
```

```
1begin2procedure p = 3λ(x,y).(4integer p#=Ω;
                        p# := if y=1 then 5x else 6p(x,y-1)↑x; 7p#);
       print(p(read,read))
end
with input (2,2).
```

example 4.

```
begin integer procedure p;
      begin integer a;
            a := read;
            p := if a=0 then p else a
      end;
      print(p)
end
```

```
1begin 2procedure p = 3λπ.(4integer (p#,a)=(Ω,Ω); a := read;
                          5p# := if a=0 then 6p else 7a; 8p#);
      print(p)
end
```

with input (0,5).

example 5.

```
begin integer k;
      integer procedure p(a); value a; integer a;
      begin k := 5; p := a end;
      k := 2;
      print(p(k))
end
```

```
1begin 2integer k=Ω;
      3procedure p = 4λa.(5integer p#=Ω; k := 5; 6p# := a; 7p#);
       k := 2;
      8print(p(k))
end
```

example 6.

As example 5, but without "value a;":

$_8$print(p(k)) becomes $_8$print(p($_9$k)).

example 7.

```
begin integer a,b;
      procedure f(i,x); value i; integer i; label x;
      begin
          m:b := b+1;
            i := i+1;
            if b=4 then begin a := i; goto 1 end;
            if b=3 then goto x;
            if b=2 then f(i,x) else f(i,m)
      end
      a := b := 0;
      f(a,1);
    1:print(a)
end
```

```
1begin 2integer (a,b) = (Ω,Ω);
      3label 1=δ22;
       procedure f = 4λ(i,x).(5label m=δ6;
          m:6b := b+1;
            7i := i+1;
            8if b=4 then begin 9a := i; 10goto 1 end else11;
           12if b=3 then 13goto x else14;
           15if b=2 then 16f(i,17x) else 18f(i,19m);
       a := b := 0;
     20f(a,21 1);
   1:22print(a)
end
```

In each of these 7 examples, the program is simple enough for the result of running it to be predicted beforehand, by running the program in the head. Example 8 is a program which is, I hope, too difficult for the result of running it to be predicted beforehand. In it, a procedure calls itself recursively, with another procedure as parameter, and very soon one is lost, unless one has a mechanical means of keeping track of the environments.

example 8.

```
begin integer n;
      procedure p(x,c); value c; procedure x; integer c;
      begin begin procedure r(t); label t;
                  begin n := n+c;
                        x(t);
                        goto t
                  end;
                  if c>n then x(j) else p(r,c+1);
            j:end;
            goto 1
      end;
      procedure q(t); label t;
         n := n+1;
      n := 2;
      p(q,2);
    1:print(n)
end
```

```
1begin 2integer n=Ω;
    3label 1=δ20;
    procedure (p,q)=(4λ(x,c).(5begin 6label j=δ15;
                                procedure r=7λt.(n := n+c;
                                                 8x(9t);
                                                 10goto t);
                                if c>n then 11x(12j) else 13p(14r,c+1);
                              j:15end; 16goto 1),
                    17λt.(n := n+1));
    n := 2;
  18p(19q,2);
 1:20print(n)
end
```

4.5.

To complete my discussion of ALGOL 60, I have still to consider arrays, for-loops, switches and own variables.

4.5.1.

The last major feature of ALGOL 60 still to be accounted for is the ability to collect values into arrays. I render

array a[m:n];

as let a = formarray(((m,n))) in ....

Here, formarray is a primitive machine function which places an array of the appropriate size and dimension, with each element initialised to Ω, on the Stack. To cope with multidimensional arrays, the argument of

formarray is a list of argument pairs. If any upper bound is lower than the corresponding lower bound, then I take formarray as being undefined. The subscripting function is represented by a "σ" on Control. If a tuple of subscripts is on the top of the Stack, and the address of the array lies below them, then a σ at the head of Control causes the address of the required element to replace them on the head of the Stack. If the actual array lies below the subscript tuple, then a σ at the head of Control causes the required element itself to replace them on the Stack. I illustrate by running the program

```
begin integer n;
      n := read;
      begin real array a[1:n];
            integer 1;
            real max;
            max := i := 0;  comment not legal ALGOL 60;
          1:if i=n then goto m;
            i := i+1;
            a[i] := read;
            if a[i]>max then max := a[i];
            goto 1;
          m:print(max)
      end
end,
```

or,

```
1begin2integer n=Ω;
      n := read;
    3begin4real array a=formarray(((1,n)));
          integer i=Ω;
          real max=Ω;
          5label (l,m)=(δ6,δ15);
          max := i := 0;
       l:6if i=n then7goto m else 8;
          9i := i+1;
          10a[i] := read;
          11if a[i]>max then12max := a[i] else13;
          14goto l;
       m:15print(max)
    end
end
```

on the I.S.C.E.M.O. machine with input (3,7.0,4.0,9.0) in figure A36. At line 6, I have the parameters of formarray, tupled once by T_2 to form the bound-pair argument list, and again by T_1 since the array is one-dimensional and formarray operates upon a list with the same number of elements as the array has dimensions. The array appears on the Stack as the list (Ω,Ω,Ω). The two-dimensional array formed by formarray(((1,2),(1,3))) (which corresponds to array[1:2,1:3]) would appear as $((\Omega,\Omega,\Omega),(\Omega,\Omega,\Omega))$. At line 21 the address of a (l_2) is put onto the Stack, ready for subscripting. At line 22, a σ is at the head of Control, and the subscripts, tupled into a list, lie on the Stack above l_2. The result of the σ is shown at line 23, where

the address ($l_{2,1}$) of the first element of the array has been formed. This address is now used like any other; I assign to it at line 24, and take the contents of it at line 34. I now take liberties with ALGOL 60 and add a call by reference facility, to run the following programs:

```
begin procedure p(i,r); integer i; real r;
      begin i := 2;
            a[1] := 2;
            print(r)
      end;
      p(i,a[i])
end
```

where r is called by value, reference and name, in an environment where i is an integer equal to 1, and a is a [1:2] array initialised to (1,3). These three programs are run in figures A37a, A37b and A37c respectively.

4.5.2.

There are only three constructions of ALGOL 60 left to be considered: the switch, for loop and own variables.

The switch is nothing more than an initialised array of labels. I have only to place the appropriate δ-closures onto the Stack and then tuple them into a list, in readiness to be bound to a switch variable by a λ_{rec}.

A for-loop can invariably be expressed in terms of simpler constructs in ALGOL 60, and so no new mechanisms need be added to the machine.

The for-loop

for v := AE do S; is equivalent to

```
v := AE; S;.
```

The for-loop

for v := AE1 step AE2 until AE3 do S; is equivalent to

```
      v := AE1;
again:vtemp := AE2;
      if sign(vtemp)×(v-AE3)>0 then goto finish;
      S;
      v := v+vtemp;
      goto again;
finish:....
```

It should be noted that AE2 is evaluated only once in each pass through the loop, since it may be a procedure with side effects (only Randell and Russel, in R1, seem to get this right).

The for-loop

for v := AE while B do S; is equivalent to

```
again:v := AE;
      if B then S else goto finish;
      goto again;
finish:....
```

Finally, the for-loop

for v := FLE,FL do S; is equivalent to

for v := FLE do S; for v := FL do S;

The treatment of own variables is as follows. Firstly, for any own variable in a program, all occurrences of that own variable are altered to some identifier which occurs nowhere else in the program. Then, all the declarations of own variables are moved to the head of the program. Thus the program

```
begin
      procedure r;
      begin own integer a;
            ...
            begin own real a;
                  ...
                  a := a+1;
                  ...
            end;
            ...
            a := a+2;
            ...
      end;
      own boolean a;
      ...
      a := false;
      ...
end
```

would become:

```
begin own boolean a†;
      own integer a††;
      own real a†††;
      procedure r;
      begin
            ...
            begin
                  ...
                  a††† := a†††+1;
                  ...
            end;
            ...
            a†† := a††+2;
            ...
      end;
      ...
      a† := false;
      ...
end.
```

This is only one of the several conflicting views of what the declaration of own variables should mean, and the matter usually provokes hot discussion. I do not intend to pursue the matter further.

4.6.

I must now give the precise definition of the I.S.C.E.M.O. machine. The outstanding difficulty of such a specification (since I am ignoring data-types) is the specification of the Memory, and the transformations associated with it. The obvious method of specifying addresses is to have the Memory a list or group, and each address one of the list or group selectors 'nth' or 's(n)'. This necessitates having a secret counter kept in the memory, so that the machine can "remember" the next address to be allocated. A big problem is how to represent arrays. The straight forward way is to have an array as a list, or group whose elements are selected by consecutive selectors. Two-dimensional arrays would then be 1-D arrays whose elements are 1-D arrays, and so on to higher dimensions.

If I use only lists to contain the Memory, then the μ operator will not be needed (assignment to an address being by taking off the first and last part of the memory list, and then inserting the new value in the middle, discarding the old value), but some nasty programming results (but see section 5.5.3.). On the other hand, if groups are used, then the μ operator will be needed, but simpler programming results.

When the machine comes to the subscripting functions, it has to be able to tell whether an address or the array itself lies below the subscript list, so consequently it must be able to recognise an address when it sees one. I therefore keep an address as a composition of selectors, and also represent the Memory as a group. Thus the address $1_{i_1,i_2,\ldots,i_r}$ is held as the function $s(i_r)\cdot s(i_{r-1})\cdot\ldots\cdot s(i_1)$.

The total contents of the I, S, C, E, M and O components of the machine at any time will be called the <u>configuration</u> of the machine: the

total amount of information necessary to continue the computation uniquely. The contents of the S, C, E and M components will be called the state of the machine; the total amount of information necessary to continue the computation for any given input.

I can now give the definitions of a configuration, transform and initialise.

Definition of a configuration:

```
configuration =
       <input = identifier-list,
        state = <stack = <ε-markervaddressvstack-item>-list,
                  control = <ε-markervcontrol-item>-list,
                  environment = <top:<identifier,primitive-object>-list,
                                 static-chain:environment,
                                 dynamic-chain:environment>vPE,
                  memory = <count:nni,store:<stack-itemvstore>-group>>,
        output = identifier-list>.
address = primitive-object.
stack-item = primitive-objectv
             λ-closure = <λ-marker-part:λ-marker,
                          ε-marker-part:ε-marker>V
             δ-closure = <δ-marker-part:δ-marker,
                          ε-marker-part:ε-marker>V
             stack-item-list.
```

control-item = identifier∨

λ-marker∨

δ-marker∨

Δ-marker∨

applier∨

assigner∨

subscripter∨

choice∨

tupler = <tuplicity:nni>∨

formarray∨

contents-of∨

jump∨

voider∨

read∨

print.

ε-marker = environment.

λ-marker = <recursiveness:truth-value,

bound-variable-list:identifier-list,

body:control-item-list>.

δ-marker = <body:control-item-list>.

Δ-marker = <body:δ-marker>.

From this definition, I have the following predicates, selectors and constructors, with abbreviations I shall use in brackets: is-input, is-state, is-stack, is-control, is-environment (isenv), is-memory, is-output, is-address(isaddr), is-stack-item, is-control-item, is-top, is-identifier(isid), is-primitive-object(isprimob), is-PE, is-λ-closure, is-λ-marker(isλexp), is-δ-closure, is-δ-marker(isδexp), is-Δ-marker(isΔexp), is-applier(isγ), is-assigner(is←), is-subscripter(isσ), is-choice(isif), is-tupler,

is-formarray, is-contents-of(isC), is-jump(isJ), is-voider(isv), is-read and is-print; s-input, s-state, s-stack, s-control, s-environment, s-memory, s-output, s-top, s-identifier(id), s-primitive-object(primob), s-static-chain(sch), s-dynamic-chain(dch), s-count, s-store, s-λ-marker-part(λpart), s-ε-marker-part(εpart), s-δ-marker-part(δpart), s-tuplicity, s-recursiveness(rec), s-bound-variable-list(bvlist) and s-body; c-configuration(c-f), c-state(c-s), c-environment(c-env), c-memory, c-store, c-top, c-λ-closure and c-δ-closure.

I can now give:

```
transform(configuration) =
let I = input(configuration)
and S = stack·state(configuration)
and C = control·state(configuration)
and E = environment·state(configuration)
and M = memory·state(configuration)
and O = output(configuration)
 in
(isid(X) → cf(I,cs(lookup(X,E):S,Y,E,M),O)
          where rec lookup(I,E) = (isPE(E) → val(PE)(I),
                    isnull·top(E) → lookup(I,sch(E)),
                    id·h·top(E)=I → primob·h·top(E),
                         lookup(c-env(t·top(E),sch(E),dch(E)))),
isλexp(X) → cf(I,cs(c-λclosure(X,E):S,Y,E,M),O),
isδexp(X) → cf(I,cs(S,append(body(X))(Y),E,M),O),
isΔexp(X) → cf(I,cs(c-δclosure(body(X),E):S,Y,E,M),O),
is←(X) → cf(I,cs(hS:t²S,Y,E,c-memory(count(M),μ(hS,htS,store(M)))),O),
isσ(X) → cf(I,cs(subscript(hS,htS):t²S,Y,E,M),O)
```

```
        where subscript(L,V) = (isaddr(V) → affix(L)·V,affix(L)(V))
              where rec affix(L) = (isnull(tL) → s(hL),
                                                 affix(tL)·s(hL)),
isif(X) → cf(I,cs((hS=true → htS,hS=false → ht²S):t³S,Y,E,M),O),
istupler(X) → cf(I,cs(listⁿ(S):tⁿS,Y,E,M),O)
              where n = tuplicity(X),
isformarray(X) → cf(I,cs(arraygroup(hS):tS,Y,E,M),O)
      where rec (arraygroup(L),arraylist(m,n,L)) =
                ((isnull(L) → val(PE)(Ω), arraylist(1st·hL,2nd·hL,tL)),
                 (m=n → μ(arraygroup(L),s(m),c-group),
                         μ(arraygroup(L),s(m),arraylist(m+1,n,L)))),
isℰ(X) → cf(I,cs((hS)(store(M)):tS,Y,E,M),O),
isJ(X) → cf(I,(isδclosure(hS) → cs(voidto(E')(tS),
                 append(body·δpart·hS)(voidto(E')(Y)),E',M)),O)
                 where rec voidto(E) → λL.(hL=E → L,voidto(E)(tL))
                 and E' = εpart·hS,
isν(X) → cf(I,cs(tS,Y,E,M),O),
isenv(X) → cf(I,cs((htS=X → hS,hS=X → htS):t²S,Y,dch(E),M),O),
isγ(X) → (isprimob(hS) → cf(I,cs(apply(hS,htS):t²S,Y,E,M),O),
    isλclosure(hS) → cf(I,cs(E':t²S,append(body·λpart·hS)(E':Y),E',
        c-memory(count(M)+R,assign(count(M),update(htS),store(M)))),O)
    where rec update(L) = (~rec·λpart·hS → L,
              isnull(L) → nullist
             (isλclosure(hL) → c-λclosure(λpart·hL,E'),
              isδclosure(hL) → c-δclosure(δpart·hL,E'),hL):update(tL))
    and rec assign(A,V,M) = (isnull(V) → M,
                                         μ(hV,s(A),assign(A+1,tV,M)))
    where E' = c-env(unzip(bvlist·λpart·hS,formaddrs(count(M),
                                 count(M)+R)),εpart·hS,E)
```

```
                where rec formaddrs(R,T) = (R=T → nullist,
                                            s(R):formaddrs(R+1,T))
        where R = length·bvlist·λpart·hS),
isread(X) → cf(tI,cs(val(PE)(hI):S,Y,E,M),O),
isprint(X) → cf(I,cs(tS,Y,E,M),represent(PE)(hS):O))
where X = hC
and Y = tC.
```

Here, 'represent(PE)' is the inverse function to 'val(PE)'; it takes a primitive object and forms an identifier which represents it. Difficult points are the definitions of subscript, assign, E' and the associated change in the memory. I form a tuple with formaddrs, and then use unzip to open it out again, rather than try to form the addresses one at a time, solely for convenience.

I also have the initialising function:

```
initialise(I,X) = cf(acceptinput(I),cs(PE:(),makecontrol(X):PE:(),
                                       PE,c-memory(1,c-store)),()).
```

The functions 'acceptinput' and 'makecontrol' are not further formalised.

As with the S.C.E. machine, this definition allows considerable scope for "sabotage" by the clever programmer. For example, my definitions for subscripting allow me much more than ALGOL 60 allows; I can subscript an array more than once. That is, if I have array al[1:n,1:m,1:p] then my machine will allow me to write al[i][j][k], al[i][j,k] or al[i,j] k as well as al[i,j,k].

Chapter five

Extensions

5.1.

One of the major objections to the I.S.C.E.M.O. machine as presented in the previous section, is the inefficiency of several of the constructions used. It is my dual purpose in this chapter to streamline the I.S.C.E.M.O. machine, and to show some generalizations of ALGOL 60 which take advantage of the full range of possibilities of the I.S.C.E.M.O. machine.

The first obvious target for optimisation is the conditional statement. At the moment, I use an 'if' to select one of two λ-closures, then a 'γ' to apply the selected λ-closure to the null argument-list. This then creates a null environment layer, and I then continue the computation of the success arm of the conditional, after which I exit from the null environment layer. Clearly, the null argument-list and the extra γ are redundant. Also, for ALGOL 60, the λ-markers freezing the two arms of the conditional can only be placed on the Stack in the same environment as the one in which the *if* is applied, and thus the ε-marker parts of the closure will always point to the current environment. Hence, the null environment layer is also redundant. I could avoid the null environment layer by freezing the arms of a conditional statement as Δ-markers, and allowing a δ-closure to be unfrozen by a γ. However, Δ-markers, introduced solely to facilitate the use of J, are themselves good candidates for removal. Hence, from now on, I do not bother with the null argument list or the γ, but allow an if both to select the required λ-closure and then to "apply" it.

"Apply" here is in quotes to indicate that the λ-closure is not applied to an argument, nor is a new environment entered. I re-run the computation of λx.(x>0 → x×x,x-1)5, first shown in figure A9, in figure A38. The expression in control syntax is now

$$5T_2\lambda_1\gamma$$

$$\lambda_1 \equiv \lambda x.\lambda_3\lambda_2 x\epsilon 0T_2{>}\gamma if$$

$$\lambda_2 \equiv \lambda\pi.x\epsilon x\epsilon T_2{}^{\times}\gamma$$

$$\lambda_3 \equiv \lambda\pi.x\epsilon 1T_2{}^{-}\gamma.$$

At line 5, an if is at the head of Control. The second element on the Stack is chosen, and all three top elements removed. The body of the chosen λ-closure is then put into Control, and the computation continues normally. It might be thought that I should generalise this use of if, and allow the two λ-closures to have ε-markers different from the current environment. However, if I do this, then I should also allow both λ-closures to require an argument list, and I then lose efficiency, and also have the problem of each λ-closure requiring a different number of arguments. I therefore reject such attempts to generalise the use of if; besides, one can always obtain such generality if desired by the ordinary use of if. While I am considering conditionals, I make the observation that the begin and end in 'then begin ... end else' are redundant, the then and else being perfectly good brackets in themselves. Likewise, 'if boolean b; b := read; b=x>y then' is easily handled on my machine. In order to allow any number of statements after an else, as after a then, I must enclose the end of the conditional with some symbol, fi say. My conditionals now appear as either

if condition then statements1 else statements2 fi;

or if condition then statements1 fi;

where 'condition', 'statements1' and 'statements2' can be any sequence of statements (but 'condition' must end with a boolean expression).

As I have already indicated, Δ-markers are redundant so I now allow a J to act on a λ-closure just as it did on a δ-closure, provided that the λ-closure has a null bound-variable list (I reject the possibility of allowing J to act on a closure requiring an argument list: my use of J is sufficient). Since such a λ-closure is indistinguishable from a procedure with no parameters, a jump now appears as acting on a parameterless procedure. Indeed, there is a considerable similarity between parameterless, typeless procedures and labels; so much so that the goto symbol is actually redundant: an ALGOL 60 processor could ignore it completely. One can, in fact, eliminate all labels and goto statements from an ALGOL 60 program by transforming them into procedure declarations and procedure calls respectively ('van Wijngaarden's device', W4). Since the λ-calculus allows a λ-closure to be delivered out of the environment in which it was created (which is not allowed in ALGOL 60), in such a case one cannot remove everything above the appropriate ε-marker from the Stack and Control, because those ε-markers will already have been deleted. In this case, I make J add the ε-markers of the λ-closure to the heads of the Stack and Control, before appending the body of the λ-closure to the Control.

The next step I take is to remove the double occurrence of "λ" at the beginning of each block. At the moment, I use an ordinary λ-marker to declare the non-recursive variables and a λ_{rec}-marker to introduce the recursive ones. Unfortunately, it is not possible to use λ_{rec} everywhere,

as this would lead to programs running incorrectly, for example:

```
begin integer i;
      procedure p(i); integer i;
         i := i+1;
      i := read;
      print(p(i))
end,
```

as run in figure A39. However, I can use λ_{rec} to introduce the non-recursive variables in the blockhead, since no non-recursive variable can be initialised to a λ- or δ-closure. I shall however eliminate λ_{rec} later on.

I now note some extensions of ALGOL 60 which the comprehensive machinery of the I.S.C.E.M.O. machine allows me. For example, the λ-expression

```
x := y+(λr.r := (read+z)/2; r)Ω
```

corresponds to the illegal ALGOL 60

```
x := y+begin real r; r := (read+z)/2; r end;
```

In fact, I can now allow a block anywhere that a result is required. The last occurrence of 'r', to deliver its value, is actually unnecessary in this case, since 'r := (read+z)/2' will deliver the required value. I can also write expressions such as

```
if b>0 then a else b fi := 0;
```

and I shall allow `xx := addr x;`

which assigns the address of x to xx. This enables me to program list-processing algorithms. An example of a list-processing program

is given later on.

One final step with ALGOL 60 is to clean up my inefficient storage allocation, and remove some of the large number of contents-of operators which are required at present. At the moment, I allocate storage for procedures and labels, even though I cannot assign to them. From now on, I return to my original S.C.E. mechanism for applying a λ-closure, and use an explicit storage raising function '$' to create addresses. This gives me greatly improved storage handling facilities, and a much more efficient parameter-passing mechanism. Now, all parameters are called by value; call by reference is an address called by value, and call by name is a procedure called by value. This is not however the call by value of ALGOL 60: one now cannot assign to a parameter called by value, since what is passed is a value and not an address, nor is any address created. I have already hinted at call by reference in the program of figure A37b, but my new method is more efficient. I run the program of figure A26 again in figure A40, but now written in ALGOL 68 and on my new machine.

```
begin bool x;
      proc p = (ref bool a)void:if read then p(a) else q(a) fi;
      proc q = (ref bool a)void:if read then p(a); a := ~a fi;
      x := read;
      p(x);
      print(x)
end,
```

and partially translated:

```
1begin2ref bool x = loc bool;
      proc p = (3ref bool a)void:if read then 4p(a) else5q(a) fi;
      proc q = (6ref bool a)void:if read then 7p(a); 8a := ~a
                                 else 9skip fi;
      x := read;
    10p(x);
    11print(x)
end.
```

The brevity both of the compiled code and of the program run as compared with that of figure A26 is remarkable, especially the passing of the parameters.

5.2.

As indicated by the previous example, the alterations presented in this chapter so far have been motivated by a consideration of the possibilities of ALGOL 68. I now consider how I am to deal with programs containing constructions of the form:

```
begin int x;
      x := read;
      real r;
      ....
```

Certainly, this is equivalent to

```
begin int x;
      real r;
      x := read;
      ...,
```

but I cannot perform a similar transformation to the following piece of text:

```
begin int n;
      read(n);
      [1:n]real a1;
      ....
```

Also, as the machine stands at present, storage locations to be bound to a variable in a λ-marker must be allocated before the λ-marker is applied. If I try to implement an efficient storage scheme, where variables disappear on exit from the block in which they were created, this is intolerable. For both of these reasons, I separate the function of environment entry from that of binding variables to their arguments. From now on, I use λ in the usual way, but modify the δ-markers so that they have a bound-variable list. When a δ-marker reaches the head of Control, the identifiers of the bound variable list are coupled with the arguments of an argument list ready on top of the Stack, and these identifiers are added to the current environment. The δ-body is added to Control in the usual way. I must check that the identifiers added in this way are not already present, otherwise chaos will ensue. I do not now need λ_{rec} for recursive variables, since I shall enter an environment, the λ-marker which denotes the body of a procedure will be turned into a λ-closure with the current environment as its ε-marker, and when a δ-marker reaches the head of Control, the procedure identifier will be added to the current environment, coupled with its thus "updated" λ-closure.

The I.S.C.E.M.O. machine, with all the alterations to it so far considered, can handle very sophisticated programs indeed. For example, the following program can be handled with ease, after one trivial modification:

```
begin struct node = (ref node left,string val,ref node right);
     node tree := (node := (nil,"bob",nil),"jim",node := (nil,
                                                  "sam",nil));
     proc postwalk = (ref node root)void:if root :≠: nil then
                                   postwalk(left of root);          (9)
                                   print(val of root);
                                   postwalk(right of root) fi;
     postwalk(tree)
end.
```

My modified δ-markers mean that an address may be created on the Stack before being bound to a variable. If I now assign to that address, I will lose it forever, since the result of an assignation is the value assigned. Consequently, I alter assignments so that the result of an assignation is the address assigned to. Program (9) partially translated is

```
1begin
     2mode node = struct(ref node left,string val,ref node right);
     3ref node tree = heap node := (heap node := (nil,"bob",
                  nil),"jim",heap node := (nil,"sam",nil));
      proc postwalk = (4ref node root)void:if root :≠: nil then
                                   5postwalk(left of root);
                                   6print(val of root);
                                   7postwalk(right of root)
                                    else 8skip fi;
      postwalk(tree)
end,
```

and hence in control syntax is

$\delta_1 \equiv \delta\pi.T_0\lambda_2\gamma$

$\lambda_2 \equiv \lambda\pi.\$\$\Omega$"bob"$\Omega T_3 \leftarrow$"jim"$\$\Omega$"sam"$\Omega T_3 \leftarrow T_3 \leftarrow \lambda_4 T_2 \delta_3$

$\delta_3 \equiv \delta$(tree,postwalk).tree T_1 postwalkγ

$\lambda_4 \equiv \lambda$root.$\lambda_8\lambda_5$root$\Omega T_2 \neq \gamma$if

$\lambda_5 \equiv \lambda\pi$.root1$T_1 \sigma \mathcal{C} T_1$ postwalk$\gamma\delta_6$

$\delta_6 \equiv \delta\pi$.root2$T_1 \sigma \varepsilon$print$\delta_7$

$\delta_7 \equiv \delta\pi$.root3$T_1 \sigma \mathcal{C} T_1$postwalk$\gamma$

$\lambda_8 \equiv \lambda\pi$.

and is run in figure A41.

I can even run programs such as the following, which is not legal ALGOL 68:

```
begin op ∘ = (proc(real)real f,g)proc(real)real:
                                    proc(real x)real:f(g(x));
      proc(real)real sincos = sin∘cos;
      print(sincos(0))
end
```

In this program, the routine which is the result of ∘ is local to the body of ∘, and thus the identity declaration sincos = sin∘cos is illegal. However, on the I.S.C.E.M.O. machine, it runs perfectly well,

```
1begin2
      3op ∘ =(4proc(real)real f,g)proc(real)real:
                                    5proc(real x)real:f(g(x));
      6proc(real)real sincos = sin∘cos;
       print(sincos(0))
end,
```

or, in control syntax,

$\delta_1 \equiv \delta\pi.T_0\lambda_2\gamma$

$\lambda_2 \equiv \lambda\pi.\lambda_4 T_1 \delta_3$

$\delta_3 \equiv \delta\circ.\sin\ \cos\ T_2 \circ \gamma T_1 \delta_6$

$\lambda_4 \equiv \lambda(f,g).\lambda_5$

$\lambda_5 \equiv \lambda x.xT_1 g\gamma T_1 f\gamma$

$\delta_6 \equiv \delta\text{sincos}.OT_1\text{sincos}\gamma\text{print}.$

as seen in figure A42.

Two alterations to the machine I have not dealt with, but must mention, are the following. As already noted in chapter three, for ALGOL (either 60 or 68) programs, the static tree of a computation is always a subtree of the dynamic tree for that computation. This allows one to deallocate an environment after exit from it, since one will never be able to reenter it. Hence, I could if I wanted merge the Stack and Environment components of the machine into one "superstack". Also, in ALGOL 60 (but not ALGOL 68), constructions like the following are forbidden:

x := x+if y>1 then z+2 else goto alarm fi;.

And, on exit from blocks and procedures (except for exit from a type-procedure), the Stack is always empty. These two facts make it the case that a J at the head of Control will never cause anything to be removed from the Stack. Hence, the environment markers in the Stack can be omitted. Since these possibilities are not open to me when implementing the full λ-calculus (which, with my new mechanisms, I can do just as efficiently on the I.S.C.E.M.O. machine as on the S.C.E. machine), I have left these modifications to one side.

5.3.

I must now give the new definitions for a configuration, transform and initialise.

Definition of a configuration:

configuration =

<input = identifier-list,
state = <stack = <ε-markervstack-item>-list,
control = <ε-markervcontrol-item>-list,
environment = <top:<identifier,primitive-object>-list,
static-chain:environment,
dynamic-chain:environment>vPE,
memory = <count:nni,store:<stack-itemvstore>-group>>,
output = identifier-list>.

stack-item = primitive-objectv
address = primitive-objectv
λ-closure = <λ-marker-part:λ-marker,
ε-marker-part:ε-marker>v
stack-item-list.

control-item = identifierv
λ-markerv
δ-markerv
applierv
assignerv
subscripterv
choicev
tupler = <tuplicity:nni>v
generatorv

```
                formarray∨
                contents-of∨
                jump∨
                voider∨
                read∨
                print.
ε-marker = environment.
λ-marker = <bound-variable-list:identifier-list,
            body:control-item-list>.
δ-marker = <bound-variable-list:identifier-list,
            body:control-item-list>.
```

I have the same predicates, selectors and constructors as before, with the exception of is-δ-closure, is-Δ-marker, s-recursiveness, s-δ-marker-part and c-δ-closure, and the addition of is-generator(is$). Since there is now only one type of closure, I abbreviate is-λ-closure to isclosure and c-λ-closure to c-closure. I allow an address to be a stack-item so that I may form arrays of addresses, and also for ease of definition of the environment and memory components.

I shall need the functions member(x,E), which yields true if x is an identifier in the environment layer E, and false otherwise; and subenv(E,L) which tests if the environment E is on the list L:

```
member(x,E) = (isnull(E) → false,x=id·hE∨member(x,tE)).
subenv(E,L) = (isnull(L) → false,
                        E=hL → true,subenv(E,tL)).
```

Here, the "=" in "E=hL" is the predicate of equality between environments.

I can now give:

```
transform(configuration) =
let I = input(configuration)
and S = stack·state(configuration)
and C = control·state(configuration)
and E = environment·state(configuration)
and M = memory·state(configuration)
and O = output(configuration)
 in
(isid(X) → cf(I,cs(lookup(X,E):S,Y,E,M),O)
          where rec lookup(I,E) = (isPE(E) → val(PE)(I),
                    isnull·top(E) → lookup(I,sch(E)),
                    id·h·top(E)=I → primob·h·top(E),
                         lookup(c-env(t·top(E),sch(E),dch(E)))),
isλexp(X) → cf(I,cs(c-closure(X,E):S,Y,E,M),O),
isδexp(X) → cf(I,cs(tS,append(body(X))(Y),
       c-env(update(unzip(bvlist(X),hS),top(E)),sch(E),dch(E)),M),O)
        where rec update(L,E) = (isnull(L) → E,
                  ~member(id·hL,E) → update(tL,postfix(hL)(E)),
isε(X) → cf(I,cs(tS,Y,E,c-memory(count(M),μ(hS,htS,store(M)))),O),
isσ(X) → cf(I,cs(subscript(hS,htS):t²S,Y,E,M),O)
        where subscript(L,V) = (isaddr(V) → affix(L)·V,affix(L)(V))
              where rec affix(L) = (isnull(tL) → s(hL),
                                    affix(tL)·s(hL)),
isif(X) → cf(I,cs(t³S,append((εpart·htS=E∧εpart·ht²S=E∧
            isnull·bvlist·λpart·λtS∧isnull·bvlist·λpart·ht²S →
     body.(λpart((hS=true → htS,hS=false → ht²S))))(Y),E,M),O),
istupler(X) → cf(I,cs(listⁿ(S):tⁿS,Y,E,M),O)
```

```
                where n = tuplicity(X),
isS(X) → cf(I,cs(s(count(M)):S,Y,E,c-memory(count(M)+1,store(M))),O),
isformarray(X) → cf(I,cs(arraygroup(hS):tS,Y,E,M),O)
      where rec (arraygroup(L),arraylist(m,n,L)) =
                ((isnull(L) → val(PE)(Ω),arraylist(1st·hL,2nd·hL,tL)),
                 (m=n → μ(arraygroup(L),s(m),c-group),
                        μ(arraygroup(L),s(m),arraylist(m+1,n,L)))),
isC(X) → cf(I,cs((hS)(store(M)):tS,Y,E,M),O),
isJ(X) → cf(I,(isnull·bvlist·λpart·hS → cs(voidto(E')(tS),
            append(body·λpart·hS)(voidto(E')(Y)),E',M)),O)
            where rec voidto(E) = λL.(subenv(E,L) → voidto(E)(tL),E:L)
            and E' = εpart·hS,
isν(X) → cf(I,cs(tS,Y,E,M),O),
isenv(X) → cf(I,cs((htS=X → hS,hS=X → htS):t²S,Y,dch(E),M),O),
isγ(X) → (isprimob(hS) → cf(I,cs(apply(hS,htS):t²S,Y,E,M),O),
    isclosure(hS) → cf(I,cs(E':t²S,append(body·λpart·hS)(E':Y),
                                                    E',M),O)
    where E' = c-env(unzip(bvlist·λpart·hS,htS),εpart·hS,E)),
isread(X) → cf(tI,cs(val(PE)(hI):S,Y,E,M),O),
isprint(X) → cf(I,cs(tS,Y,E,M),represent(PE)(hS):O))
where x = hC
and Y = tC.
```

Initialise is as before:

```
initialise(I,X) = cf(acceptinput(I),cs(PE:(),makecontrol(X):PE:(),
                                        PE,c-memory(1,c-store)),()).
```

5.4.

I now give an extension to the I.S.C.E.M.O. machine, to enable it to handle the case clause:

case integral-expression in S_1, S_2, ..., S_n out S esac

where 'integral-expression' is any sequence of statements ending with one that yields an integral result, and S, S_1, S_2, ..., S_n are any statements whatever (and they may yield a value). This is rendered in control syntax as

$\lambda_0\lambda_1\lambda_2\cdots\lambda_n T_n$ integral-expression case

where $\lambda_0 \equiv \lambda\pi.S$, $\lambda_i \equiv \lambda\pi.S_i$, for $1 \leq i \leq n$,

and case has the following effect. When a case is at the head of Control, if the head of the Stack has the value i, then the ith. λ-marker from the second element on the Stack is chosen and "applied". If i is out of range (i.e. less than one, or more than the number of markers in the list), then the third element on the Stack is chosen and "applied".

This renders if redundant, since the conditional expression

if condition then S_1 else S_2 fi

is equivalent to

case abs condition in S_1 out S_2 esac.

Here, abs applied to a truth-value yields 1 for true, and 0 for false. Thus, if 'condition' is true, the one and only element in the "success" list (S_1) is chosen; and if it is false, then the "default" option

chooses S_2. However, in practice one would keep both the case and if control items. As with if, I eschew any attempts at generalisations.

I therefore make the following alterations to the definitions of a configuration and transform. In the definition of a configuration, I must add the line "...vcasev...". This gives me the predicate iscase. In the definition of transform, I must add after the section beginning with "isif(X) →" the section:

```
"iscase(X) → cf(I,cs(t³S,append((hS≠0∨hS>length(htS)→
                (εpart·ht²S=E∧isnull·bvlist·λpart·ht²S → body·λpart·ht²S),
                check(htS,E) → body·λpart·elem(hS)(htS)))(Y),E,M),O)
                where rec check(L,E) = (isnull(L) → true,
                        εpart·hL=E∧isnull·bvlist·λpart·hL∧check(tL,E)),".
```

5.5.

In this sub-section I shall outline those directions for future research which I should like to see carried through.

5.5.1.

One important area for future investigation is the notation and machinery needed to accommodate parallel processing, or collateral evaluation. Parallel processing is the evaluation of more than one expression at the same time. On the blackboard machine, this could be accomplished by allowing the Control and Stack lists to become forked at an ε-marker. The elaboration of the several arms of the Control could then proceed in parallel. This allows blocks and procedures to

be elaborated collaterally. Each branch of the Control (a "task") has access to the identifiers and values in the common stem of the Environment. Within each task evaluation is performed serially. Incidentally, coroutines are tasks with no common Control stem, that is, the Control forks at the PE marker. This idea can be extended further by allowing the Control and Stack to fork at any point, to allow the collateral elaboration of expressions such as formulas and assignments. Then, an expression such as x+y would be elaborated by having x and y placed onto the heads of a forked Stack, and the + would operate on the two heads, yielding one result (thus "unforking" the Stack). An example of such an extended version of the I.S.C.E.M.O. machine elaborating the program

```
par(x[i] := i := i+1,
    x[i] := i := i+1); print(i,x)                    (10)
```

in an environment where i is 1, and x is a ref[1:3]int initialised to (1,1,1), is shown in figure A43. Only one of the many possible elaborations is shown. The translation of (10) into control syntax would be:

$$\delta_1 \equiv \delta\pi.\left\{\begin{matrix}\delta_2\\ \delta_3\end{matrix}\right\}\delta_4$$

$$\delta_2 \equiv \delta\pi.\left\{\begin{matrix} x\ i\epsilon T_1\sigma \\ \left\{\begin{matrix} i \\ i\epsilon 1T_2+\end{matrix}\right\}\leftarrow\epsilon\end{matrix}\right\}\leftarrow\nu$$

$$\delta_3 \equiv \delta\pi.\left\{\begin{matrix} x\ i\epsilon T_1\sigma \\ \left\{\begin{matrix} i \\ i\epsilon 1T_2+\end{matrix}\right\}\leftarrow\epsilon\end{matrix}\right\}\leftarrow\nu$$

$$\delta_4 \equiv \delta\pi.i\epsilon\text{print } x1T_1\sigma\epsilon\text{print } x2T_1\sigma\epsilon\text{print } x3T_1\sigma\epsilon\text{print}$$

However, formalising this presents problems, both in the definitions of the data structures used, and in the function transform. The data structures must cater for the Control lists to split into two or more parallel lists at any point, and to rejoin again later on. This could be done by having the elements of Control a control-item-group. For transform, a method must be devised so that upon reaching a fork in the Control list, the Stack is forked and elaboration of all arms of Control proceeds collaterally. During such a time, transform must allow each task unimpeded access to the shared part of the Environment, with some method of resolving simultaneous accesses. Furthermore, some method must be devised such that when two or more control paths rejoin, elaboration of each arm stops upon reaching the junction, and no further progress is made until all arms have been elaborated up to the junction. Further, such a method must allow for all control paths to be terminated upon one of the control paths being prematurely terminated, e.g. by a goto. Also, provision must be made for arms being elaborated collaterally to signal to each other, possibly by means of semaphores[D1]. It may be more convenient to suppose an instruction-sequencing mechanism as in VDL[L5]. There, one of the set of all possible next instructions to be elaborated is chosen at random. If this should prove to be so, then a defining machine and programming language on the VDL pattern may be more appropriate.

5.5.2.

A second important area for future research is that of modes, or the tagging of data items as belonging to one of an infinite number of different data types. Certainly, the defining machine has no problems here: each mode can be assigned a numeral, and a numerical tag associated with each data item. The primitive operations can then be made to be undefined if the modes do not match. However, a coherent system is needed; it is not even certain what primitive modes and mode defining facilities are "best", but I believe the ALGOL 68 notion of MODEs[W5] to be a step in the right direction. Difficulties here are that values of one mode often need to be coerced to some other mode. This can be achieved by having the apply function, assigner, subscripter, etc. coerce their arguments to the required mode, or by inserting specific coercion operators into the expressions. This second course is not open if a dynamic mode system is used, that is, if the modes of operands are only checked at run-time, and are not required to be consistent at compile-time. While every value has a definite mode, a variable may be required to receive values of more than one mode. Thus a defining machine must be prepared to keep a list of the possible values that a variable may receive (although this information can be built into the mode, see W5), and a tag giving the mode of the value currently held. It may be desirable to have a variable able to receive values of any one of an infinite number of modes, for example, variables of mode general, or of mode printtype (which could contain any value capable of being printed). Also, circular (recursive) modes [W5,M6] are certainly required in a practical system. Thus it appears that a mode has much the same form as a constructed object (see chapter 2), and can be represented as a graph. Standish, in S1, considers modes in great detail.

5.5.3.

In Chapter 5, for efficiency I separated the action of environment entry from that of binding variables to their values. Since the λ-calculus is the most natural way of looking at procedure definition and activation, can the λ-calculus be modified to mirror my ad hoc invention rigorously? Or, should we take the unmodified I.S.C.E.M.O. machine as a basis for more advanced machines and not worry about a theoretical interpretation? For example, languages which are defined such that it is always the most recent value associated with a variable which is retrieved, can be implemented simply on the I.S.C.E.M.O. machine by eliminating the static chain. This seems to have no theoretical counterpart, because the environment component of the machine remembers substitutions already performed. Thus accessing the "wrong" value would cause in a theoretical model the alteration of β-reductions already made.

A difficult problem, mentioned earlier (section 2.2.), is the simulation of μ in the λ-calculus. Since the λ-calculus is a universal computing agency, it must be possible to do this. The problem is intimately linked with that of representing the data structures defined by SDs within the λ-calculus. This must be possible with a carefully-enough designed coding of graphs. As discussed in section 4.6., it is not necessary to use μ in my definition of transform there. This is because the data structures involved do not contain loops, despite the apparent loops formed during the updating of λ-closures by λ_{rec}. The last portion of any such "loop" is in fact an address, which is held as a function, and not as a pointer into the store. It is not possible, however, to avoid the use of μ in my definition of the transform of section 5.3.. This is because a constructor constructs a new node, and

links it to the arguments of the constructor (which are shared). Thus in effecting the updating of an environment by a δ-marker, the state contains the updated environment, but an ε-marker pointing to that environment in any λ-closure added during the updating will still point to the old environment node, replaced by the constructor, and thus only to those parts of the environment which were present before the updating. This occurs whether the defining machine copies or shares arguments.

5.6.

I shall end on a point of fundamental significance. Namely, it is impossible to use my defining machine to define itself. That is, one cannot write a program which defines the defining machine. The reason for this is simple: the defining machine itself is performing the interpretation. One cannot use an undefined machine to implement a definition; the defining machine must "know" how it is to perform in order to accomplish the interpretation of the definition. For example, for my definition of transform, I took the defining machine as sharing arguments. However, this gives no information as to the argument passing policy of a machine interpreting the defining machine. Depending on how the program representing the defining machine is written, such an interpreter could either copy or share arguments, or perform a mixture of both. Hence, the defining machine cannot specify exactly how it treats arguments, since this depends on how arguments are treated by its interpreter. Writing a program to implement a defining machine which copies arguments is useless if its interpreter makes only one copy of every constructed object. Expressed in another manner, for any program which purports to define the defining machine, I can write an

interpreter which implements the program correctly, but in a manner different to the one specified by the program. It is for this reason that an algorithmic or formal language cannot define a machine; there must be some machine already in existence to interpret the definition of such a language machine. There is an infinite regression of machines here; clearly (since computers do actually work), this infinite regression must be cut short at some point. It is cut short by a human being, who can supply the necessary intelligence to get the process started. This is clear; to define a machine a semantic definition is required, a syntactic description (a program) is useless unless a semantic description of the program's interpreter is given. All the syntax in the world is useless without an intelligent being to interpret the symbols used. Looked at in this manner, Gödel's theorem on the incompleteness of First Order Number Theory is obvious. No axiomatisation (syntactic description) of an inherently meaningful system can possibly hope to be complete. Certainly, a program meant to define the defining machine can be written, and a statement can be made that the interpreter of the defining machine works in the same manner as the defining program. This however requires some covering statement in English to accompany the program, and is an act of faith which cannot be formalised. There is no possible method of preventing an interested reader from misunderstanding the program, and hence misunderstanding the action of the interpreter, and for these misunderstandings to be consistent. There is no method of guaranteeing that two people will not read the same definition, and understand different things by it. This is well known; one can never be absolutely certain that what you understand something to mean and what another person understands that same thing to mean are in fact the same. One can only

use words to communicate, and all words, the definitions of which one may query, are defined in terms of other words. For this reason a dictionary is useless since it defines words in terms of other words; if you know no words to start with, you can never know. There must be an end to this infinite regression somewhere, and this end is in the intelligence of human beings, in a few basic words the definitions of which are known, and in terms of which all other meanings can be given. Likewise, mathematics assumes the definition of a set, it can never be defined in terms of simpler constructs; only equally undefined synonyms such as "collection", "ensemble", "group", etc., can be given. Thus, at the back of the mind of every intelligent being there must be a basic machine (interpreter) or "primitive environment", which is "built" into the genetic code and supplies the ultimate semantic definitions of all syntactic constructions encountered during the lifetime of the being. This also is well known; one cannot teach in a vacuum. All teaching is performed by introducing new concepts, definitions and ideas to the pupil in terms of what is already known and understood by the pupil. That is, in terms ultimately understandable by the primitive environment. Hence, all teaching merely tells the pupil something he already knows, although he may not be aware that he knows it. For example, it is possible to use the VDL interpreter[L5] to define itself superficially, but this requires the interpreter to be already programmed with the action of μ and the non-deterministic selection rule. Thus any "definition" of VDL is in terms of VDL itself and hence undefined. So, any human beings who wish to learn VDL must already have an appreciation of the action of μ (or an equivalent action) before they set out to learn the language.

APPENDIX A

figures A1 - A43

figure A1

$10\ \lambda_1\ \gamma$

$\lambda_1 \equiv \lambda a.\ a\ a\ a\ T_2\ \times\ \gamma\ T_2\ +\ \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT
1	ϵ_0	$10\ \lambda_1\ \gamma\ \epsilon_0$	0:PE
2	10	λ_1	
3	$\lambda_1 \epsilon_0$	γ	
4	ϵ_1	$a\ a\ a\ T_2\ \times\ \gamma\ T_2\ +\ \gamma\ \epsilon_1$	$1 \to 0: a = 10$
5	10	a	
6	10	a	
7	10	T_2	
8	(10 10)	$\times$	
9	$\times$	γ	
10	100	T_2	
11	(10 100)	$+$	
12	$+$	γ	
13	110	ϵ_1	
14	110	ϵ_0	
15	110		

figure A2

$\lambda_2\ \lambda_1\ \gamma$

$\lambda_1 \equiv \lambda f.\ 7\ f\ \gamma$

$\lambda_2 \equiv \lambda x.\ 2\ x\ T_2\ \times\ \gamma\ 1\ T_2\ +\ \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT
1	ϵ_0	$\lambda_2\ \lambda_1\ \gamma\ \epsilon_0$	0:PE
2	$\lambda_2 \epsilon_0$	λ_1	
3	$\lambda_1 \epsilon_0$	γ	
4	ϵ_1	$7\ f\ \gamma\ \epsilon_1$	$1 \to 0: f = \lambda_2 \epsilon_0$
5	7	f	
6	$\lambda_2 \epsilon_0$	γ	
7	ϵ_2	$2\ x\ T_2\ \times\ \gamma\ 1\ T_2\ +\ \gamma\ \epsilon_2$	$2 \to 0: x = 7$
8	2	x	
9	7	T_2	
10	(2 7)	$\times$	
11	$\times$	γ	
12	14	1	
13	1	T_2	
14	(14 1)	$+$	
15	$+$	γ	
16	15	ϵ_2	
17	15	ϵ_1	
18	15	ϵ_0	
19	15		

figure A3

$\lambda_3 \lambda_1 \gamma$

$\lambda_1 \equiv \lambda g.\, 2\, g\, \gamma\, \lambda_2\, \gamma$

$\lambda_2 \equiv \lambda f.\, 3\, f\, \gamma\, 5\, 4\, g\, \gamma\, \gamma\, T_2\, x\, \gamma$

$\lambda_3 \equiv \lambda x.\, \lambda_4$

$\lambda_4 \equiv \lambda y.\, x\, y\, T_2 + \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT
1	ϵ_0	$\lambda_3\ \lambda_1\ \gamma\ \epsilon_0$	0: PE
2	$\lambda_3 \epsilon_0$	λ_1	
3	$\lambda_1 \epsilon_0$	γ	
4	ϵ_1	$2\ g\ \gamma\ \lambda_2\ \gamma\ \epsilon_1$	1 → 0: $g = \lambda_3 \epsilon_0$
5	2	g	
6	$\lambda_3 \epsilon_0$	γ	
7	ϵ_2	$\lambda_4\ \epsilon_2$	2 → 0: $x = 2$
8	$\lambda_4 \epsilon_2$	ϵ_2	
9	$\lambda_4 \epsilon_2$	λ_2	
10	$\lambda_2 \epsilon_1$	γ	
11	ϵ_3	$3\ f\ \gamma\ 5\ 4\ g\ \gamma\ \gamma\ T_2\ x\ \gamma\ \epsilon_3$	3 → 1: $f = \lambda_4 \epsilon_2$
12	3	f	
13	$\lambda_4 \epsilon_2$	γ	
14	ϵ_4	$x\ y\ T_2\ +\ \gamma\ \epsilon_4$	4 → 2: $y = 3$
15	2	y	
16	3	T_2	
17	(2, 3)	$+$	
18	+	γ	
19	5	ϵ_4	

(contd...

figure A3 continued.

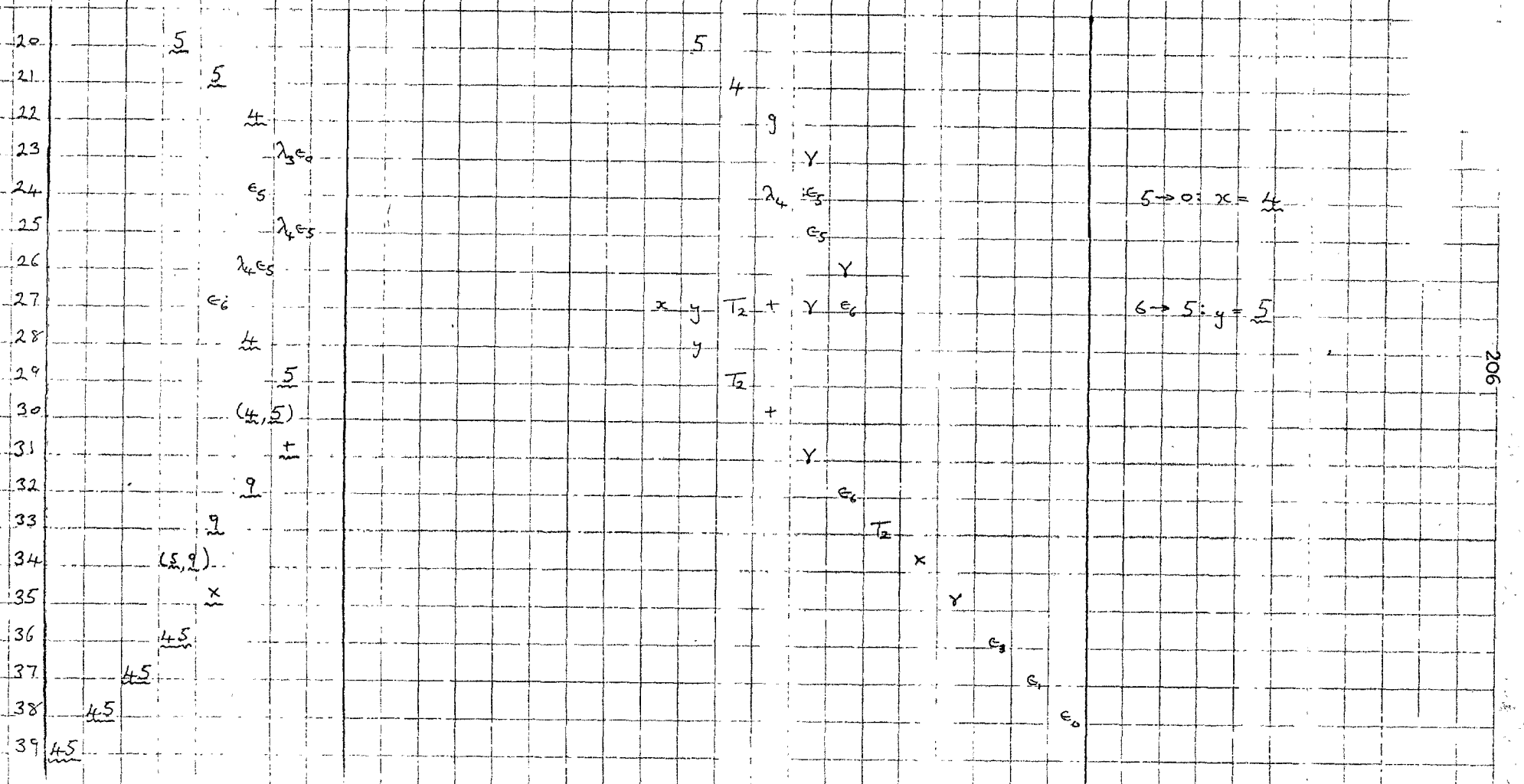


figure A4

$0\ \mathrm{suc}\ \lambda_2\ \lambda_1\ \gamma\ \gamma\ \gamma$

$\lambda_1 \equiv \lambda h.\ h\ h\ \gamma$

$\lambda_2 \equiv \lambda f.\ \lambda_3$

$\lambda_3 \equiv \lambda x.\ x\ f\ \gamma\ f\ \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT
1	ϵ_0	$0\ \mathrm{suc}\ \lambda_2\ \lambda_1\ \gamma\ \gamma\ \gamma\ \epsilon_0$	$0 : PE$
2	0	suc	
3	suc	λ_2	
4	$\lambda_2 \epsilon_0$	λ_1	
5	$\lambda_1 \epsilon_0$	γ	
6	ϵ_1	$h\ h\ \gamma\ \epsilon_1$	$1 \to 0 \cdot\ h = \lambda_2 \epsilon_0$
7	$\lambda_2 \epsilon_0$	h	
8	$\lambda_2 \epsilon_0$	γ	
9	ϵ_2	$\lambda_3\ \epsilon_2$	$2 \to 0 : f = \lambda_2 \epsilon_0$
10	$\lambda_3 \epsilon_2$	ϵ_2	
11	$\lambda_3 \epsilon_2$	ϵ_1	
12	$\lambda_3 \epsilon_2$	γ	
13	ϵ_3	$x\ f\ \gamma\ f\ \gamma\ \epsilon_3$	$3 \to 2 : x = \mathrm{suc}$
14	suc	f	
15	$\lambda_2 \epsilon_0$	γ	
16	ϵ_4	$\lambda_3\ \epsilon_4$	$4 \to 0 : f = \mathrm{suc}$
17	$\lambda_3 \epsilon_4$	ϵ_4	
18	$\lambda_3 \epsilon_4$	f	
19	$\lambda_2 \epsilon_0$	γ	
20	ϵ_5	$\lambda_3\ \epsilon_5$	$5 \to 0 : f = \lambda_3 \epsilon_4$
21	$\lambda_3 \epsilon_5$	ϵ_5	
22	$\lambda_3 \epsilon_5$	ϵ_3	
23	$\lambda_3 \epsilon_5$	γ	
24	ϵ_6	$x\ f\ \gamma\ f\ \gamma\ \epsilon_6$	$6 \to 5 : x = 0$
25	0	f	
26	$\lambda_3 \epsilon_4$	γ	
27	ϵ_7	$x\ f\ \gamma\ f\ \gamma\ \epsilon_7$	$7 \to 4 : x = 0$
28	0	f	
29	suc	γ	
30	1	f	
31	suc	γ	
32	2	ϵ_7	
33	1	f	
34	$\lambda_3 \epsilon_4$	γ	
35	ϵ_8	$x\ f\ \gamma\ f\ \gamma\ \epsilon_8$	$8 \to 4 : x = 2$
36	1	f	
37	suc	γ	
38	3	f	(cont'd...

figure A4 continued.

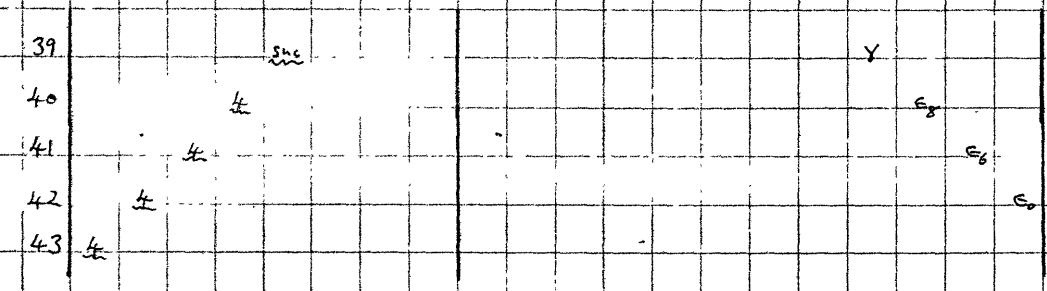


figure A5 $\lambda_5\ \lambda_1\ \gamma$

$\lambda_1 \equiv \lambda twice.\ \lambda_3\ \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda u.\ "bc"\ "a"\ u\ \gamma\ twice\ \gamma\ \gamma$

$\lambda_3 \equiv \lambda y.\ \lambda_4$

$\lambda_4 \equiv \lambda z.\ z\ rev\ \gamma\ y\ adj\ \gamma\ twice\ \gamma\ \gamma$

$\lambda_5 \equiv \lambda f.\ \lambda_6$

$\lambda_6 \equiv \lambda x.\ x\ f\ \gamma\ f\ \gamma$

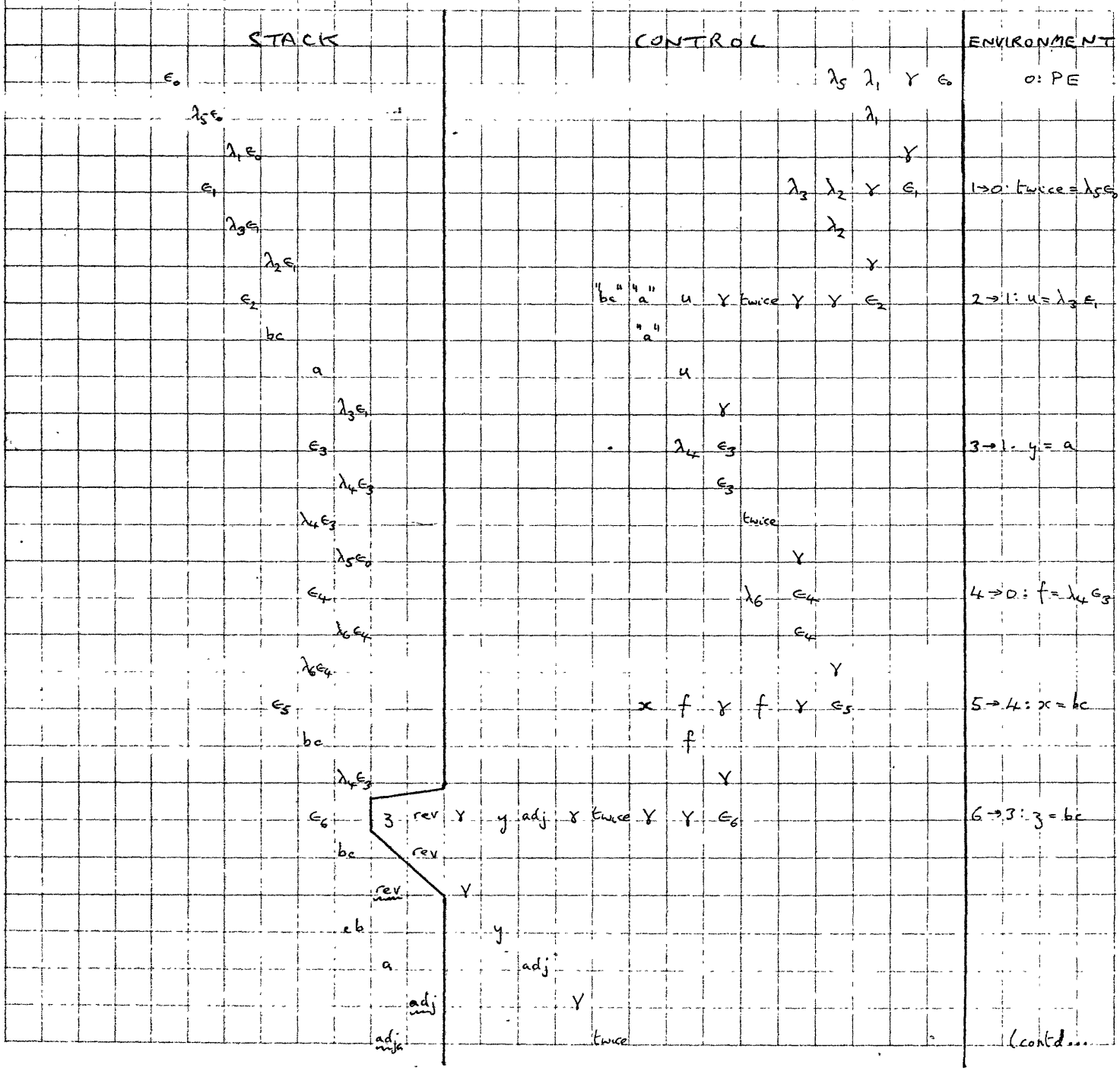


figure A5 continued.

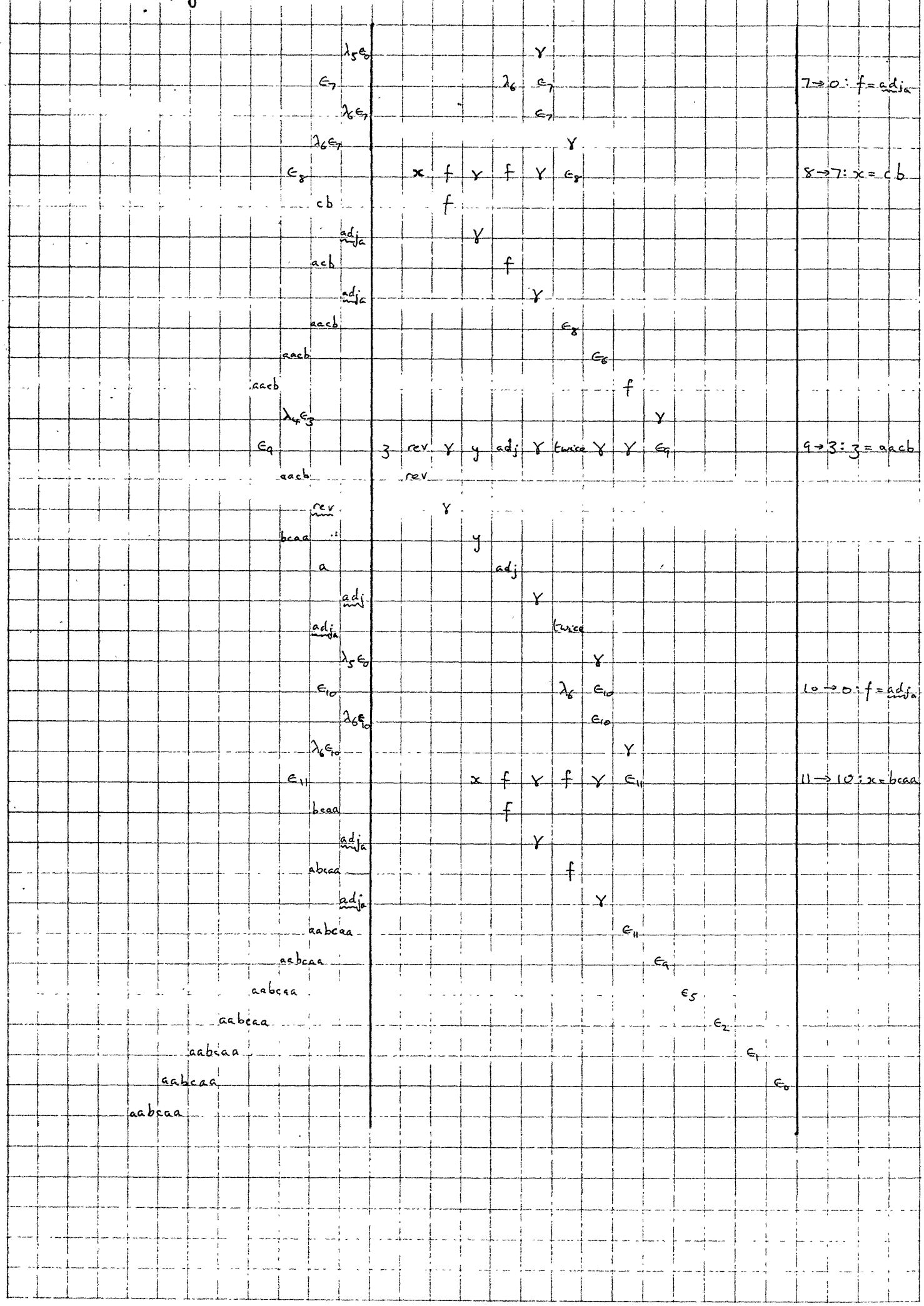


figure A6

$\lambda_2 \lambda_1 \gamma$

$\lambda_1 \equiv \lambda f . 7 2 T_2 - \gamma f \gamma f \gamma$

$\lambda_2 \equiv \lambda u . u u u T_2 \times \gamma T_2 + \gamma$

STACK	CONTROL	ENVIRONMENT
ϵ_0	$\lambda_2 \lambda_1 \gamma \epsilon_0$	0 : PE
$\lambda_2 \epsilon_0 \lambda_1 \epsilon_0$	γ	
ϵ_1	$7 2 T_2 - \gamma f \gamma f \gamma \epsilon_1$	$1 \to 0 : f = \lambda_2 \epsilon_0$
$(7,2) -$	γ	
$5 \lambda_2 \epsilon_0$	γ	
ϵ_2	$u u u T_2 \times \gamma T_2 + \gamma \epsilon_2$	$2 \to 0 : u = 5$
$5 (5,5) \times$	γ	
$(5,25) +$	γ	
30	ϵ_2	
$30 \lambda_2 \epsilon_0$	γ	
ϵ_3	$u u u T_2 \times \gamma T_2 + \gamma \epsilon_3$	$3 \to 0 : u = 30$
$30 (30,30) \times$	γ	
$(30,900) +$	γ	
930	ϵ_3	
930	ϵ_0	
930		

figure A7

$2 \lambda_1 \gamma$

$\lambda_1 \equiv \lambda a . \lambda_2$

$\lambda_2 \equiv \lambda x . x 3 \lambda_3 T_2 + \gamma$

$\lambda_3 \equiv \lambda y . y a T_2 \times \gamma$

STACK	CONTROL	ENVIRONMENT
ϵ_0	$2 \lambda_1 \gamma \epsilon_0$	0 : PE
$2 \lambda_1 \epsilon_0$	γ	
ϵ_1	$\lambda_2 \epsilon_1$	$1 \to 0 : a = 2$
$\lambda_2 \epsilon_1$	ϵ_1	
$\epsilon_0 \lambda_2 \epsilon_1$	ϵ_0	
$\lambda_2 \epsilon_1$		

figure A8

$F \lambda_1 \gamma$

$\lambda_1 \equiv \lambda f . \lambda_3 \lambda_2 \gamma$

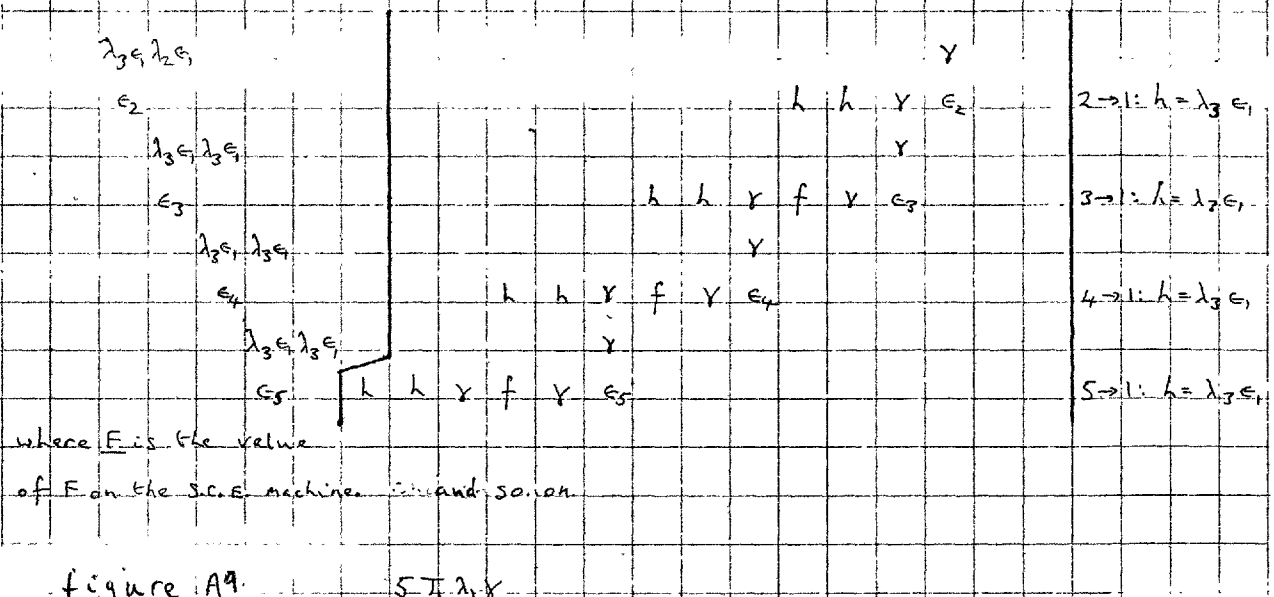
$\lambda_2 \equiv \lambda h . h h \gamma$

$\lambda_3 \equiv \lambda h . h h \gamma f \gamma$

STACK	CONTROL	ENVIRONMENT
ϵ_0	$F \lambda_1 \gamma \epsilon_0$	0 : PE
$F \lambda_1 \epsilon_0$	γ	
ϵ_1	$\lambda_3 \lambda_2 \gamma \epsilon_1$	$1 \to 0 : f = F$

(contd...

figure A8 continued.



where F is the value of F on the S.C.E. machine. and so on.

figure A9

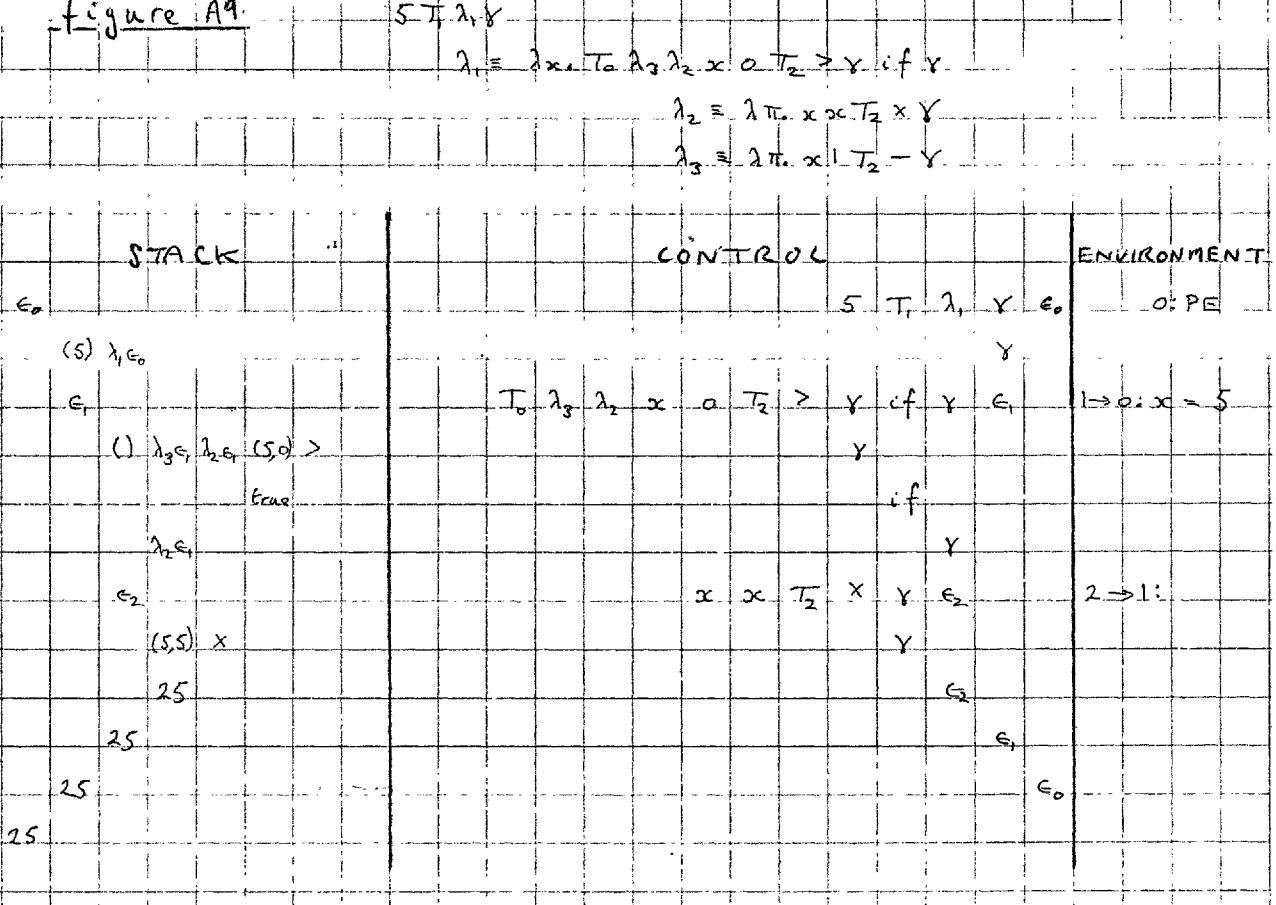


figure A10

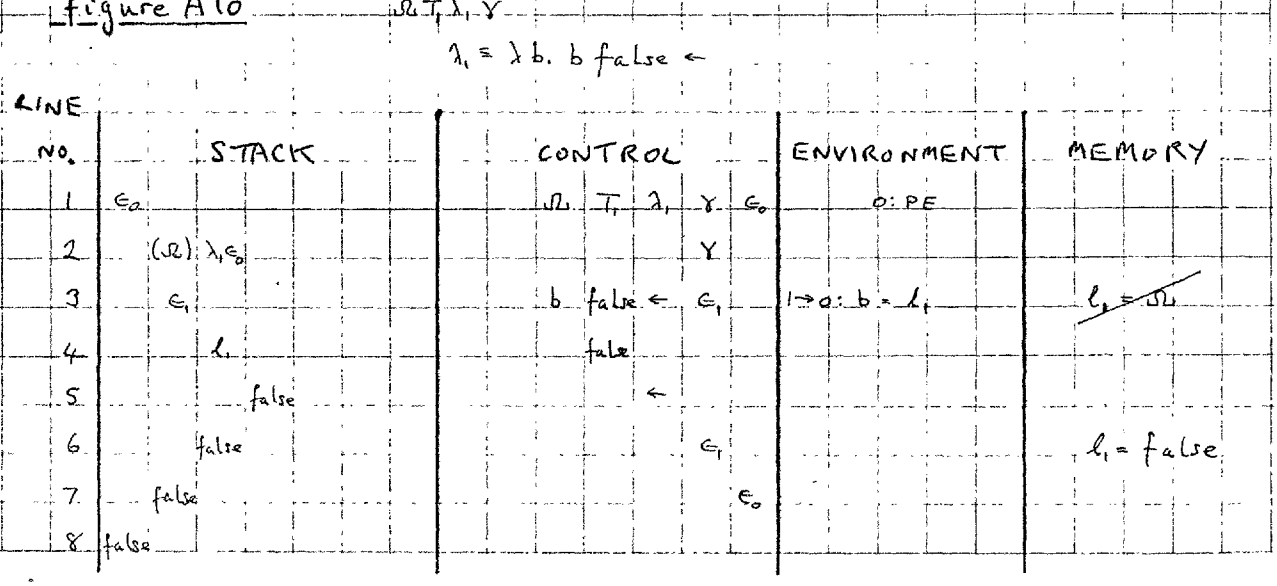


figure A-11

$\Omega_1 \Omega_2 T_2 \lambda_1 \gamma$

$\lambda_1 \equiv \lambda(x,y).\, x\, 0 \leftarrow v\, y\, x\, x\, 6\, |\, T_2 + \gamma \leftarrow \leftarrow$

LINE NO.	STACK	CONTROL	ENVIRONMENT	MEMORY
1	ϵ_0	$\Omega_1\ \Omega_2\ T_2\ \lambda_1\ \gamma\ \epsilon_0$	0: PE	
2	$(\Omega_1, \Omega_2)\ \lambda_1\ \epsilon_0$	γ		
3	ϵ_1	$x\ 0 \leftarrow v\ y\ x\ x\ :6\ |\ T_2 + \gamma \leftarrow \leftarrow \epsilon_1$	$1 \to 0$: $x = l_1$	~~$l_1 = \Omega_1$~~
4	l_1	0	$y = l_2$	~~$l_2 = \Omega_2$~~
5	0	$\leftarrow$		
6	0	v		~~$l_1 = 0$~~
7	ϵ_1	y		
8	l_2	x		
9	l_1	x		
10	l_1	6		
11	0	1		
12	$(0,1)\ +$	γ		
13	1	$\leftarrow$		
14	1	$\leftarrow$		$l_1 = 1$
15	1	ϵ_1		$l_2 = 1$
16	1	ϵ_0		
17	1			

figure A12

$\delta_1 \equiv \lambda(x,y).\, x\, 0 \leftarrow \nu\, \delta_2$

$\delta_2 \equiv y\, x\, x\, \epsilon_1\, 1\, T_2 + \gamma \leftarrow \leftarrow$

LINE NO.	STACK	CONTROL	ENVIRONMENT	MEMORY
1	ϵ_0	$\Omega_1\ \Omega_2\ T_2\ \lambda_1\ \gamma\ \epsilon_0$	0:PE	
2	$(\ell_1,\ell_2)\ \lambda_1\ \epsilon_0$	γ		
3	ϵ_1	$x\ 0 \leftarrow \nu\ \delta_2\ \epsilon_1$	$1 \to 0: x = \ell_1$	~~$\ell_1 = \Omega_1$~~
4	$\ell_1\ 0$	$\leftarrow$	$y = \ell_2$	~~$\ell_2 = \Omega_2$~~
5	0	ν		~~$\ell_1 = 0$~~
6	ϵ_1	δ_2		
7	ϵ_1	$y\ x\ x\ \epsilon_1\ 1\ T_2\ +\ \gamma \leftarrow \leftarrow$		
8	$\ell_2\ \ell_1\ \ell_1$	8		
9	0	1		
10	$(0,1)$ +	γ		
11	1	$\leftarrow$		
12	1	ϵ_1		$\ell_1 = 1, \ell_2 = 1$
13	1	ϵ_0		
14	1			

figure A13

$\delta_1 \equiv$ "result =" print δ_2

$\delta_2 \equiv$ read read $T_2 + \gamma$ print

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
534 486 +	ϵ_0	"result =" print δ_2 ϵ_0	0:PE		+
↑	result =	print			↑
	ϵ_0	δ_2			"result ="
	ϵ_0	read read T_2 + γ print			↑
↑	534	read			
	486	T_2			
↑	(534, 486) +	γ			
	1020	print			1020
	ϵ_0	ϵ_0			↑

figure A 14

$\delta_1 \equiv \Omega_1 \Omega_2 T_2 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda(a,b).\ b\ a\ \text{read} \leftarrow \leftarrow \upsilon\ \delta_3$

$\delta_3 \equiv \Omega_1 T_1 \lambda_4 \gamma\ \delta_6$

$\lambda_4 \equiv \lambda a.\ a\ \text{read} \leftarrow \upsilon\ \delta_5$

$\delta_5 \equiv b\ b\ \varepsilon\ a\ \varepsilon\ T_2 + \gamma \leftarrow \upsilon$

$\delta_6 \equiv b\ \varepsilon\ a\ \varepsilon\ T_2 + \gamma\ \text{print}$

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
3 5 ⊣	ϵ_0	$\Omega_1\ \Omega_2\ T_2\ \lambda_2\ \gamma\ \epsilon_0$	$0 : pE$		⊢
↑	$(\Omega_1, \Omega_2)\ \lambda_2\ \epsilon_0$	: γ			↑
	ϵ_1	$b\ a\ \text{read} \leftarrow \leftarrow \upsilon\ \delta_3\ \epsilon_1$	$1 \to 0 : a = l_1$	$l_1 = \Omega_1$	
	$l_2\ l_1$		$b = l_2$	$l_2 = \Omega_2$	
↑	3	read ←			
	3	←		$l_1 = 3$	
	3	υ		$l_2 = 3$	
	ϵ_1	δ_3			
	ϵ_1	$\Omega_1\ T_1\ \lambda_4\ \gamma\ \delta_6$			
	$(\Omega_1, \lambda_4)\ \epsilon_1$	γ			
	ϵ_2	$a\ \text{read} \leftarrow \upsilon\ \delta_5\ \epsilon_2$	$2 \to 1 : a = l_3$	$l_3 = \Omega_1$	
	l_3	read ←			
↑	5	υ		$l_3 = 5$	
	5	δ_5			
	ϵ_2				
	ϵ_2	$b\ b\ \varepsilon\ a\ \varepsilon\ T_2\ +\ \gamma \leftarrow \upsilon$			
	$l_2\ l_2$	ε			
	3 l_1	ε			
	5	T_2			
	(3,5) +	γ			
	8	←			

(cont'd...

figure A14 continued

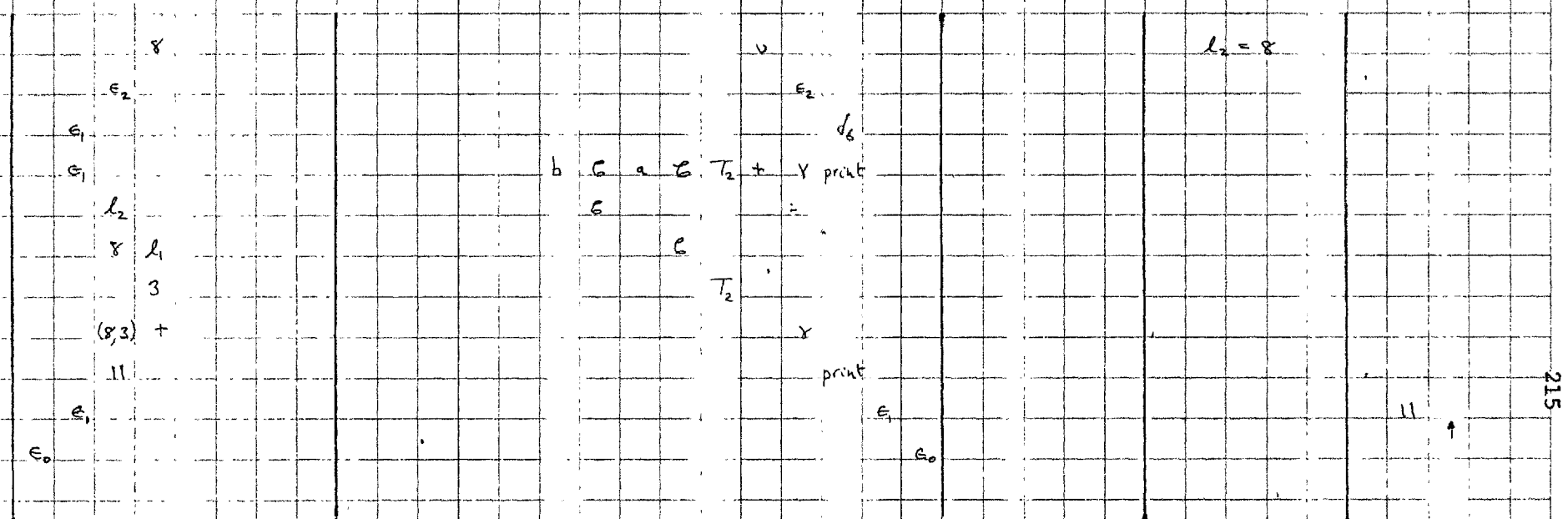


figure A15a

$\delta_1 \equiv T_0\ \lambda_3\ \lambda_2$ read if γ print

$\lambda_2 \equiv \lambda\pi.$ read

$\lambda_3 \equiv \lambda\pi.\ 0$

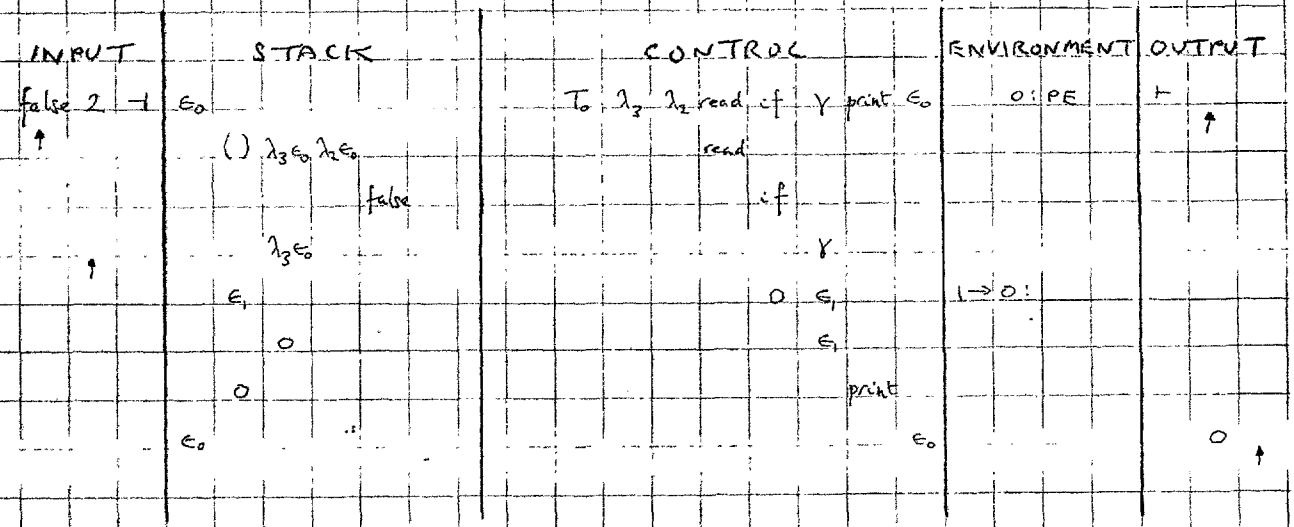


figure A15b

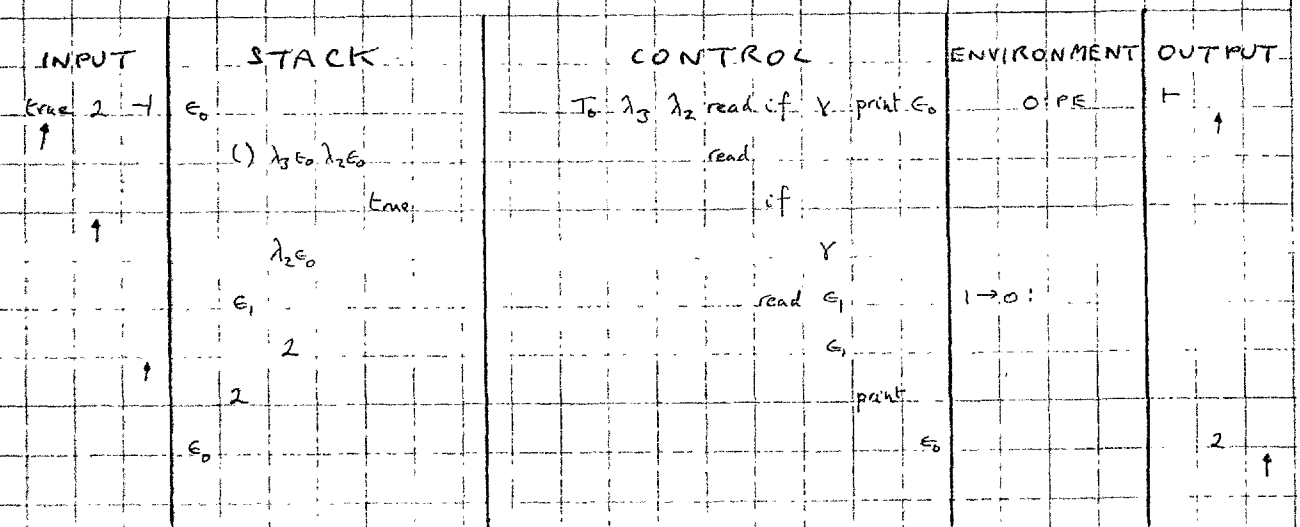


figure A16

$F\ T_1\ \lambda_1\ \gamma$

$\lambda_1 \equiv \lambda f.\ \lambda_3\ T_1\ \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda h.\ h\ \epsilon\ T_1\ h\ \epsilon\ \gamma$

$\lambda_3 \equiv \lambda k.\ k\ \epsilon\ T_1\ k\ \epsilon_\gamma\ T_1\ f\ \epsilon\ \gamma$

STACK	CONTROL	ENVIRONMENT	MEMORY
ϵ_0	$F\ T_1\ \lambda_1\ \gamma\ \epsilon_0$	0: PE	
$(\underline{F})\ \lambda_1 \epsilon_0$	γ		
ϵ_1	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_1$	1→0: $f = \ell_1$	$\ell_1 = \underline{F}$
$(\lambda_3 \epsilon_1)\ \lambda_2 \epsilon_1$	γ		
ϵ_2	$h\ \epsilon\ T_1\ h\ \epsilon\ \gamma\ \epsilon_2$	2→1: $h = \ell_2$	$\ell_2 = \lambda_3 \epsilon_1$
$(\lambda_3 \epsilon_1)\ \lambda_3 \epsilon_1$	γ		
ϵ_3	$h\ \epsilon\ T_1\ h\ \epsilon\ \gamma\ T_1\ f\ \epsilon\ \gamma\ \epsilon_3$	3→1: $h = \ell_3$	$\ell_3 = \lambda_3 \epsilon_1$
$(\lambda_3 \epsilon_1)\ \lambda_3 \epsilon_1$	γ		
ϵ_4	$h\ \epsilon\ T_1\ h\ \epsilon\ \gamma\ T_1\ f\ \epsilon\ \gamma\ \epsilon_4$	4→1: $h = \ell_4$	$\ell_4 = \lambda_3 \epsilon_1$
$(\lambda_3 \epsilon_1)\ \lambda_3 \epsilon_1$	γ		
ϵ_5	$h\ \epsilon\ T_1\ h\ \epsilon\ \gamma\ T_1\ f\ \epsilon\ \gamma\ \epsilon_5$	5→1: $h = \ell_5$	$\ell_5 = \lambda_3 \epsilon_1$

and so on.

where $\underline{F}$ is the value of F on the S.C.E. machine

figure A17

$2\ T_1\ \Omega\ T_1\ \lambda_1\ \gamma\ \gamma$

$\lambda_1 \equiv \lambda f.\ f\ \circ\ T_1\ f\ \mathcal{E}\ \gamma\ T_1\ \lambda_2 \leftarrow \upsilon\ f\ \mathcal{E}$

$\lambda_2 \equiv \lambda c.\ \lambda_3$

$\lambda_3 \equiv \lambda n.\ T_0\ \lambda_5\ \lambda_4\ n\ \mathcal{E}\ \circ\ T_2 = \gamma\ i\ f\ \gamma$

$\lambda_4 \equiv \lambda \pi.\ 1$

$\lambda_5 \equiv \lambda \pi.\ n\ \mathcal{E}\ 1\ T_2 - \gamma\ T_1\ f\ \mathcal{E}\ \gamma\ n\ \mathcal{E}\ T_2 \times \gamma\ c\ \mathcal{E}\ T_2 \times \gamma$

STACK	CONTROL	ENVIRONMENT	MEMORY
ϵ_0	$2\ T_1\ \Omega\ T_1\ \lambda_1\ \gamma\ \gamma\ \epsilon_0$	$0 : PE$	
$(2)\ (\Omega_1)\ \lambda_1\ \epsilon_0$	γ		
ϵ_1	$f\ \circ\ T_1\ f\ \mathcal{E}\ \gamma\ T_1\ \lambda_2 \leftarrow \upsilon\ f\ \mathcal{E}\ \epsilon_1$	$1 \rightarrow 0 : f = l_1$	$l_1 = \Omega_1$
$l_1\ (0)\ \Omega_1$	γ		

figure A18

$\delta_1 \equiv 1\ T_1\ \lambda_6\ T_1\ \lambda_2\ \gamma\ \gamma$

$\lambda_2 \equiv \lambda f.\ \lambda_4\ T_1\ \lambda_3\ \gamma$

$\lambda_3 \equiv \lambda h.\ h\ \mathcal{E}\ T_1\ h\ \mathcal{E}\ \gamma$

$\lambda_4 \equiv \lambda h.\ \lambda_5\ T_1\ f\ \mathcal{E}\ \gamma$

$\lambda_5 \equiv \lambda \pi.\ h\ \mathcal{E}\ T_1\ h\ \mathcal{E}\ \gamma$

$\lambda_6 \equiv \lambda f.\ \lambda_7$

$\lambda_7 \equiv \lambda n.\ T_0\ \lambda_9\ \lambda_8\ n\ \mathcal{E}\ \circ\ T_2 = \gamma\ i\ f\ \gamma$

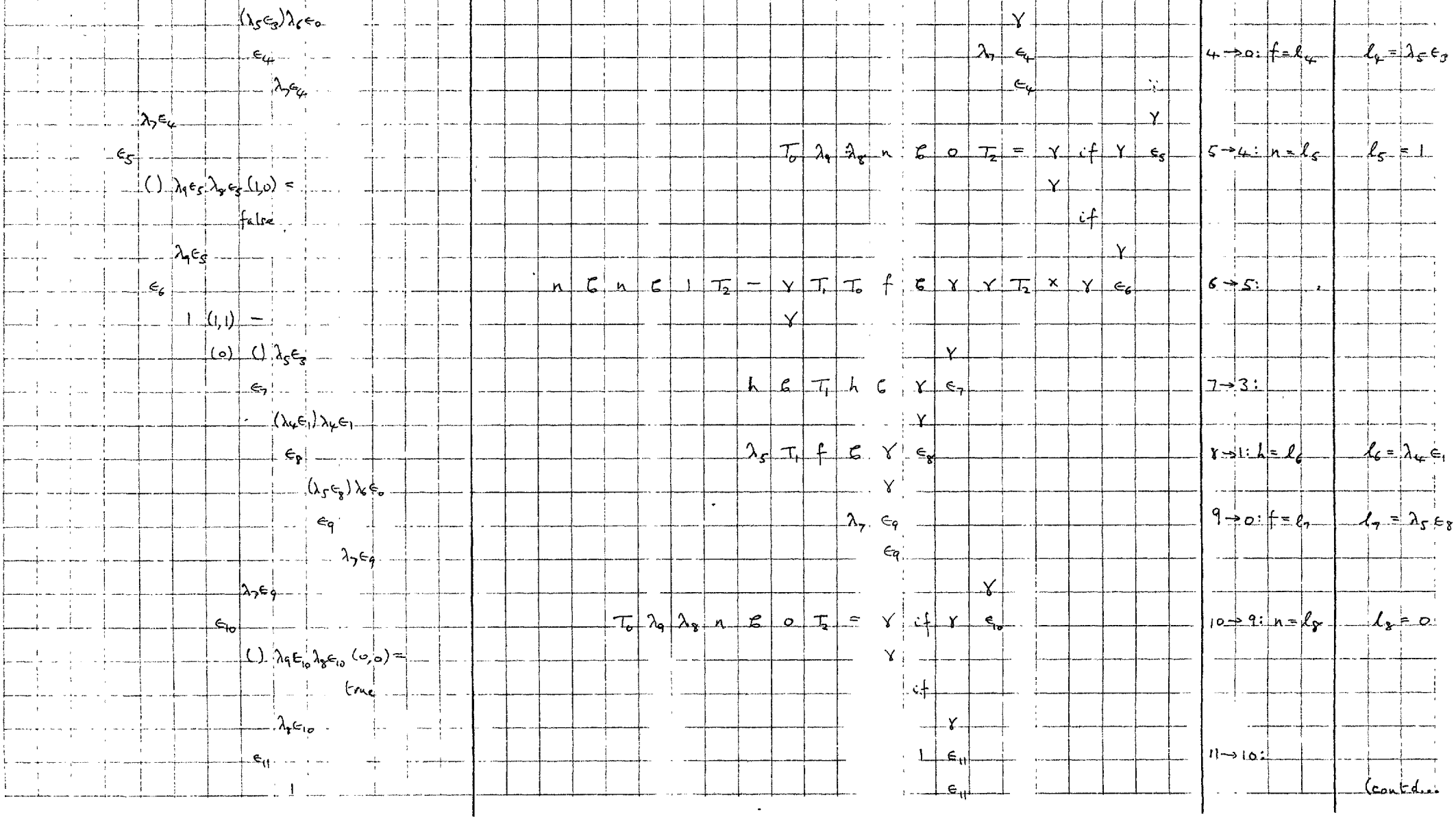
$\lambda_8 \equiv \lambda \pi.\ 1$

$\lambda_9 \equiv \lambda \pi.\ n\ \mathcal{E}\ n\ \mathcal{E}\ 1\ T_2 - \gamma\ T_1\ T_0\ f\ \mathcal{E}\ \gamma\ \gamma\ T_2 \times \gamma$

STACK	CONTROL	ENVIRONMENT	MEMORY
ϵ_0	$1\ T_1\ \lambda_6\ T_1\ \lambda_2\ \gamma\ \gamma\ \epsilon_0$	$0 : PE$	
$(1)\ (\lambda_6 \epsilon_0)\ \lambda_2 \epsilon_0$	γ		
ϵ_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ \epsilon_1$	$1 \rightarrow 0 : f = l_1$	$l_1 = \lambda_6 \epsilon_0$
$(\lambda_4 \epsilon_1)\ \lambda_3 \epsilon_1$	γ		
ϵ_2	$h\ \mathcal{E}\ T_1\ h\ \mathcal{E}\ \gamma\ \epsilon_2$	$2 \rightarrow 1 : h = l_2$	$l_2 = \lambda_4 \epsilon_1$
$(\lambda_4 \epsilon_1)\ \lambda_4 \epsilon_1$	γ		
ϵ_3	$\lambda_5\ T_1\ f\ \mathcal{E}\ \gamma\ \epsilon_3$	$3 \rightarrow 1 : h = l_3$	$l_3 = \lambda_4 \epsilon_1$

(cont'd...

figure A18 continued



(cont'd...

figure A18 continued (2)

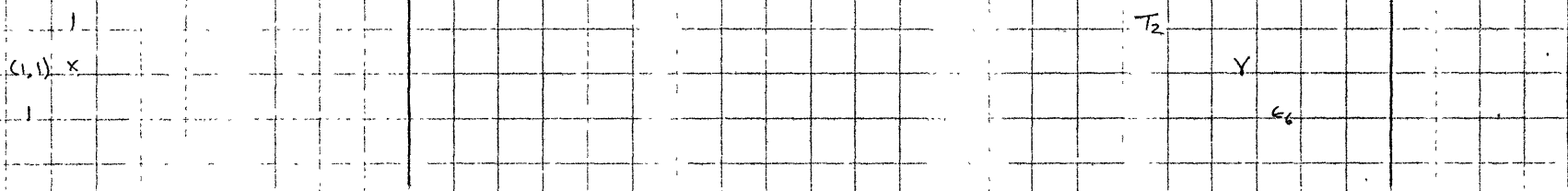


figure A19

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda f. 2 T_1 f \epsilon \gamma$

$\lambda_3 \equiv \lambda n. T_0 \lambda_5 \lambda_4 n \epsilon 0 T_2 = \gamma \text{ if } \gamma$

$\lambda_4 \equiv \lambda \pi. 1$

$\lambda_5 \equiv \lambda \pi. n \epsilon n \epsilon 1 T_2 - \gamma T_1 f \epsilon \gamma T_2 \times \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT	MEMORY
1	ϵ_0	$\lambda_3 T_1 \lambda_2 \gamma \epsilon_0$	0 : P.E.	no such =
2	$(\lambda_3 \epsilon_0) \lambda_2 \epsilon_0$	γ		
3	ϵ_1	$2 T_1 f \epsilon \gamma \epsilon_1$	$1 \to 0 : f = l_1$	$l_1 = \lambda_3 \epsilon_0$
4	(2) $\lambda_3 \epsilon_0$	γ		
5	ϵ_2	$T_0 \lambda_5 \lambda_4 n \epsilon 0 T_2 = \gamma \text{ if } \gamma \epsilon_2$	$2 \to 0 : n = l_2$	$l_2 = 2$
6	() $\lambda_5 \epsilon_2 \lambda_4 \epsilon_2 (2,0) =$	γ		
7	false	if		
8	$\lambda_5 \epsilon_2$	γ		
9	ϵ_3	$n \epsilon n \epsilon 1 T_2 - \gamma T_1 f \epsilon \gamma T_2 \times \gamma \epsilon_3$	$3 \to 2 :$	
10	2 (2,1) −	γ		
11	1	T_1		
12	(1)	f lookup f		

figure A20

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec} f\ 2\ T_1\ f\ 6\ \gamma$

$\lambda_3 \equiv \lambda n.\ T_0\ \lambda_5\ \lambda_4\ n\ 6\ 0\ T_2 = \gamma\ if\ \gamma$

$\lambda_4 \equiv \lambda \pi.1$

$\lambda_5 \equiv \lambda \pi.\ n\ 6n\ 6\ 1\ T_2 - \gamma\ T_1\ f\ 6\ \gamma\ T_2 \times \gamma$

STACK

ϵ_0

$(\lambda_3 \epsilon_0)\ \lambda_2\ \epsilon_0$

ϵ_1

(2) $\lambda_3\ \epsilon_1$

ϵ_2

(1) $\lambda_5 \epsilon_2\ \lambda_4 \epsilon_2\ (2,0) =$

false

$\lambda_5 \epsilon_2$

(*) ϵ_3

$2\ (2,1) -$

1

(1)

CONTROL

$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_0$

γ

$2\ T_1\ f\ 6\ \gamma\ \epsilon_1$

γ

$T_0\ \lambda_5\ \lambda_4\ n\ 6\ 0\ T_2 = \gamma\ if\ \gamma\ \epsilon_2$

γ

if

γ

$n\ 6\ n\ 6\ 1\ T_2 - \gamma\ T_1\ f\ 6\ \gamma\ T_2 \times \gamma\ \epsilon_3$

γ

T_1

f — lookup f

ENVIRONMENT

0: PE

$1 \to 0:\ f = l_1$

$2 \to 1:\ n = l_2$

$3 \to 2:$

MEMORY

update closure

$l_1 = \lambda_3\ \epsilon_1$

$l_2 = 2$

figure A 21

$\delta_1 \equiv \Omega_4 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda a. \Delta_{16} \Delta_4 T_2 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec}(k,l).\ a\ o \leftarrow \nu \delta_4$

$\delta_4 \equiv T_0 \lambda_6 \lambda_5 \text{read}\ \text{if}\ \gamma\ \delta_7$

$\lambda_5 \equiv \lambda\pi.\ k\ \mathcal{B}\ J$

$\lambda_6 \equiv \lambda\pi.$

$\delta_7 \equiv \Omega_1 T_1 \lambda_8 \gamma$

$\lambda_8 \equiv \lambda b. \Delta_{13} T_1 \lambda_9 \gamma$

$\lambda_9 \equiv \lambda_{rec} k.\ b\ \text{read} \leftarrow \nu\ \delta_{10}$

$\delta_{10} \equiv a\ o\ \mathcal{B}\ T_0 \lambda_{12} \lambda_{11}\ b\ \mathcal{B}\ \text{if}\ \gamma\ T_2 \leftarrow \gamma \leftarrow \nu\ \delta_{13}$

$\lambda_{11} \equiv \lambda\pi.$

$\lambda_{12} \equiv \lambda\pi.\ 2$

$\delta_{13} \equiv T_0 \lambda_{15} \lambda_{14}\ b\ \mathcal{B}\ \text{read}\ T_2 = \gamma\ \text{if}\ \gamma$

$\lambda_{14} \equiv \lambda\pi.\ k\ \mathcal{B}\ J$

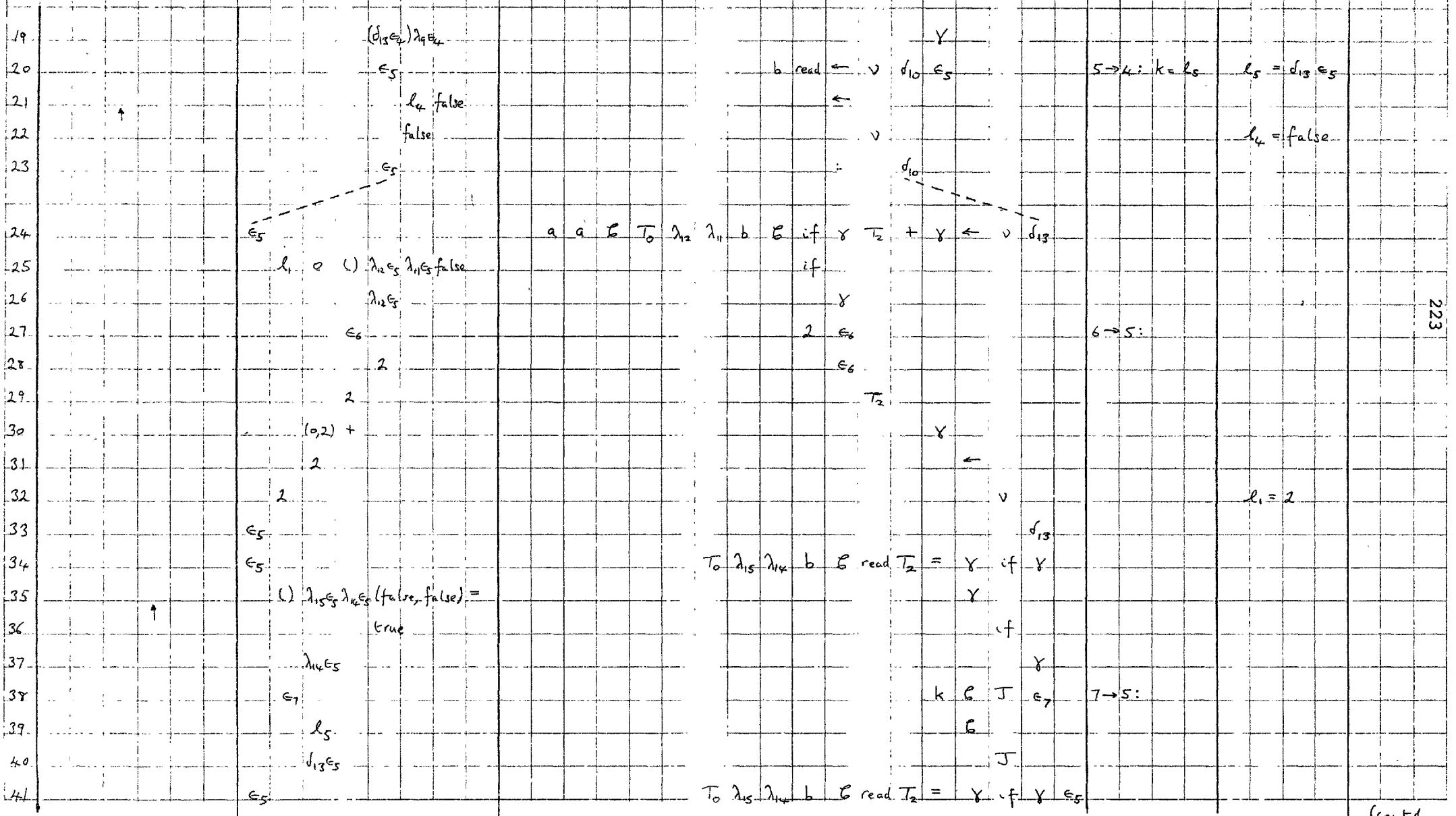
$\lambda_{15} \equiv \lambda\pi.\ l\ \mathcal{B}\ J$

$\delta_{16} \equiv a\ \mathcal{B}\ \text{print}$

LINE NO.	INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
1	false false false true true ⊣ ϵ_0 ↑		$\Omega_1\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	0: PE		⊢ ↑
2		$(\Omega_1)\ \lambda_2\ \epsilon_0$	γ			
3		ϵ_1	$\Delta_{16}\ \Delta_4\ T_2\ \lambda_3\ \gamma\ \epsilon_1$	1→0: $a = l_1$	~~$l_1 = \Omega_1$~~	
4		$\delta_{16}\epsilon_1$	Δ_4			
5		$\delta_4\epsilon_1$	T_2			
6		$(\delta_{16}\epsilon_1, \delta_4\epsilon_1)\ \lambda_3\ \epsilon_1$	γ			
7		ϵ_2	$a\ o \leftarrow \nu\ \delta_4\ \epsilon_2$	2→1: $k = l_2$	$l_2 = \delta_{16}\epsilon_2$	
8		$l_1\ o$	$\leftarrow$	$l = l_3$	$l_3 = \delta_4\epsilon_2$	
9		o	ν		~~$l_1 = o$~~	
10		ϵ_2	δ_4			
11		ϵ_2	$T_0\ \lambda_6\ \lambda_5\ \text{read}\ \text{if}\ \gamma\ \delta_7$			
12		$()\ \lambda_6\epsilon_2\ \lambda_5\epsilon_2\ \text{false}$	if			
13	↑	$\lambda_6\epsilon_2$	γ			
14		ϵ_3	ϵ_3	3→2:		
15		ϵ_2	δ_7			
16		ϵ_2	$\Omega_1\ T_1\ \lambda_8\ \gamma$			
17		$(\Omega_1)\ \lambda_8\epsilon_2$	γ			
18		ϵ_4	$\Delta_{13}\ T_1\ \lambda_9\ \gamma\ \epsilon_4$	4→2: $b = l_4$	~~$l_4 = \Omega_1$~~	

(contd..

figure A 21 continued



(contd..

figure A21 continued (2)

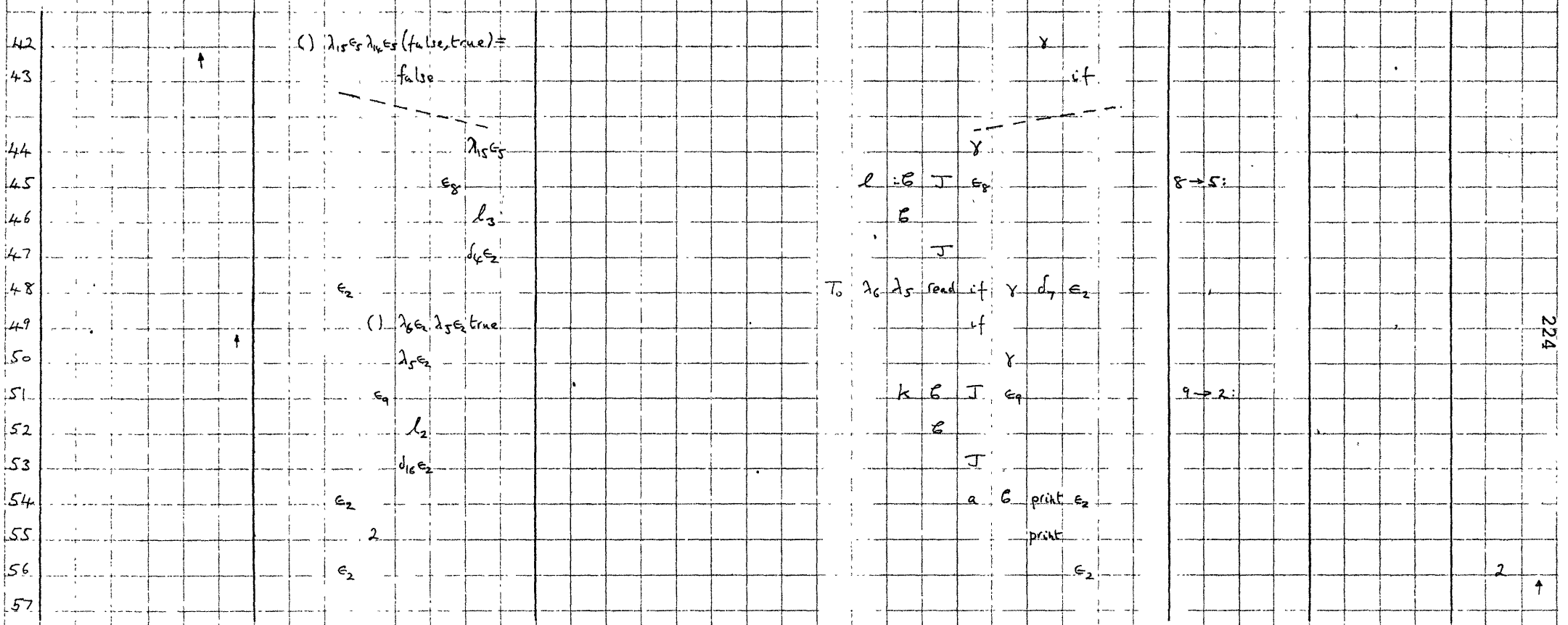


figure A22

$\delta_1 \equiv \Omega_1 \Omega_2 T_2 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda (a,b).\ \lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{read} f.\ a\ read \leftarrow \upsilon\ \delta_5$

$\lambda_4 \equiv \lambda x.\ b \times \mathbin{@} 2 \times \mathbin{@} T_2 \uparrow \gamma\ T_2 + \gamma \leftarrow \upsilon$

$\delta_5 \equiv a\ @\ a\ @\ T_2\ x\ \gamma\ T_1\ f\ @\ \gamma\ \delta_6$

$\delta_6 \equiv b\ @\ print$

LINE NO.	INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
1	3 ⊣	ϵ_0	$\Omega\ \Omega\ T_2\ \lambda_2\ \gamma\ \epsilon_0$	$0 : pe_\perp$		⊢
2	↑	$(l_2, l_2)\ \lambda_2\ \epsilon_0$	γ			
3		ϵ_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ \epsilon_1$	$1 \to 0 : a = l_1$	~~$l_1 = \delta_6$~~	
4		$(\lambda_4, \epsilon_1)\ \lambda_3\ \epsilon_1$	γ	$b = l_2$	~~$l_2 = \delta_6$~~	
5		ϵ_2	$a\ read \leftarrow \upsilon\ \delta_5\ \epsilon_2$	$2 \to 1 : f = l_3$	$l_3 = \lambda_4\ \epsilon_2$	
6	↑	$l_1\ 3$	$\leftarrow$			
7		3	υ		$l_1 = 3$	
8		ϵ_2	δ_5			
9		ϵ_2	$a\ @\ a\ @\ T_2\ x\ \gamma\ T_1\ f\ @\ \gamma\ \delta_6$			
10		$(3,3)\ x$	γ			
11		(9)	f			
12		$\lambda_4\ \epsilon_2$	γ			
13		ϵ_3	$b\ x\ @\ 2\ x\ @\ T_2\ \uparrow\ \gamma\ T_2\ +\ \gamma \leftarrow \upsilon\ \epsilon_3$	$3 \to 2 : x = l_4$	$l_4 = 9$	
14		$l_2\ 9\ (2,9)$ ↑	γ			
15		512	T_2			
16		$(9,512)\ +$	γ			
17		521	$\leftarrow$			
18		521	υ		$l_2 = 521$	
19		ϵ_3	ϵ_3			
20		ϵ_2	δ_6			
21		ϵ_2	$b\ @\ print$			

(cont'd..

figure A22 continued

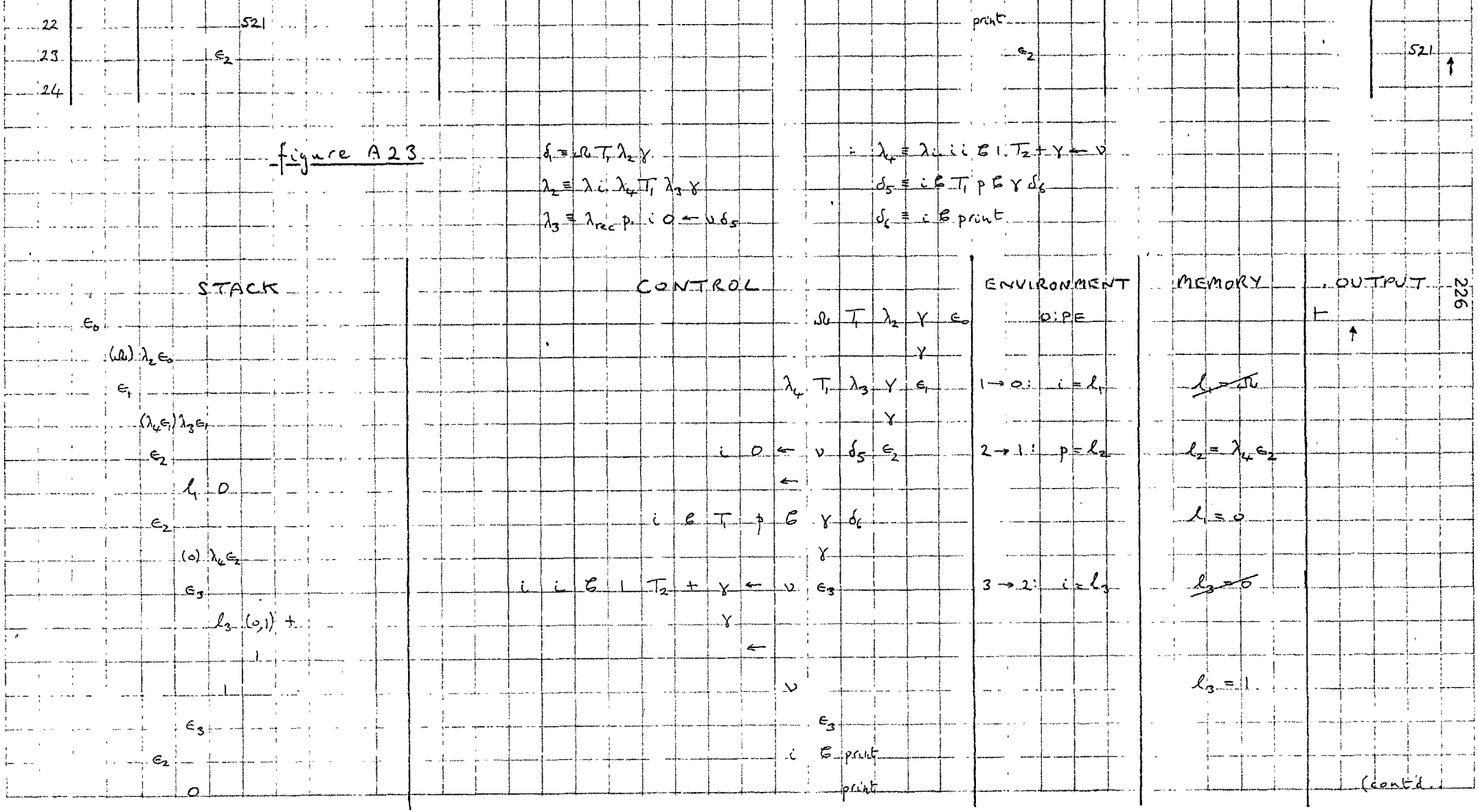


figure A23

$\delta_1 \equiv \omega \; T_1 \; \lambda_2 \; \gamma$

$\lambda_2 \equiv \lambda \; i \; \lambda_4 \; T_1 \; \lambda_3 \; \gamma$

$\lambda_3 \equiv \lambda_{rec} \; p \; . \; i \; 0 \leftarrow \nu \; \delta_5$

$\therefore \; \lambda_4 \equiv \lambda \; i \; i \; \mathcal{B} \; 1 \; T_2 + \gamma \leftarrow \nu$

$\delta_5 \equiv i \; \mathcal{B} \; T_1 \; p \; \mathcal{B} \; \gamma \; \delta_6$

$\delta_6 \equiv i \; \mathcal{B} \; print$

figure A23 continued

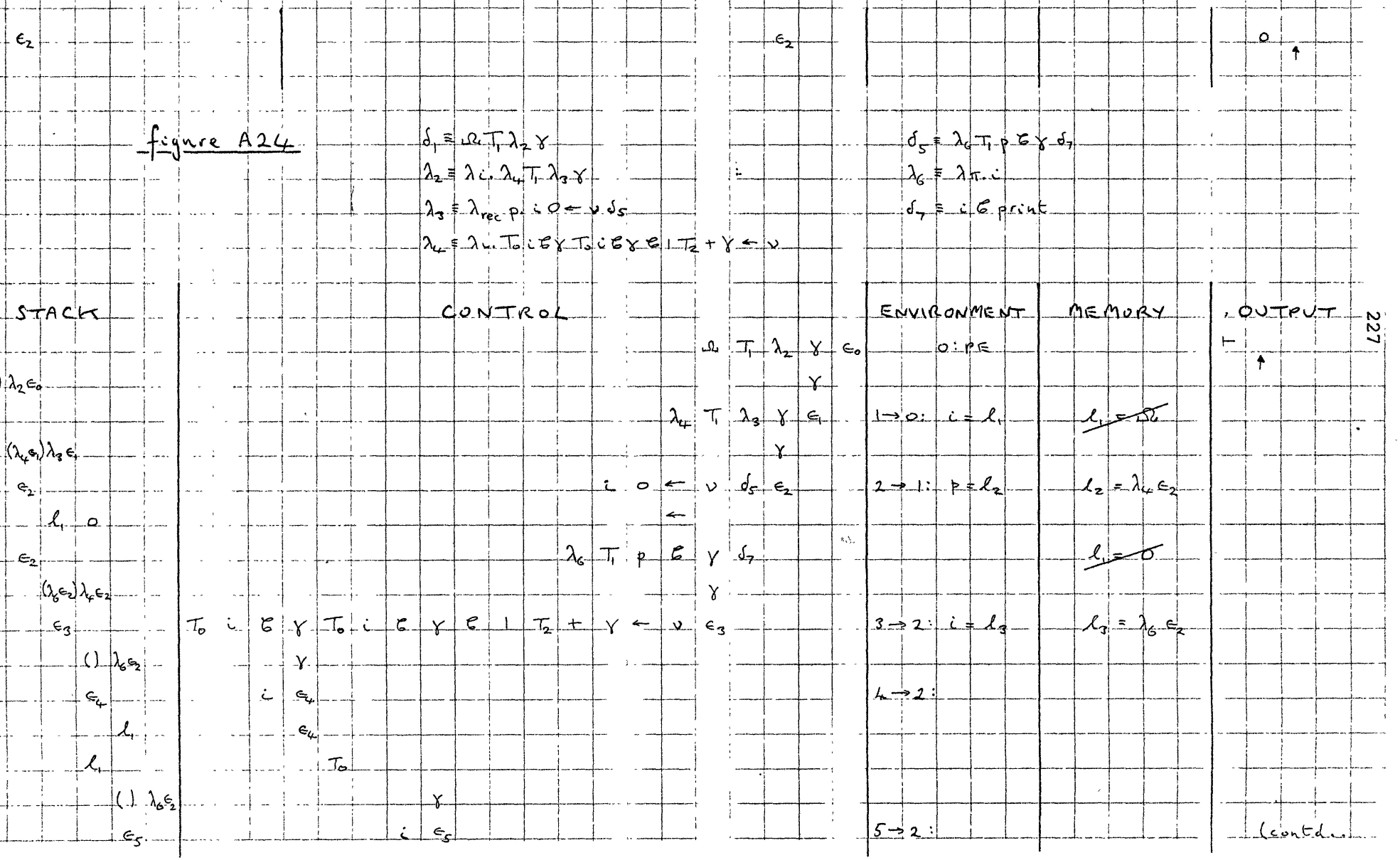


figure A24

figure A24 continued

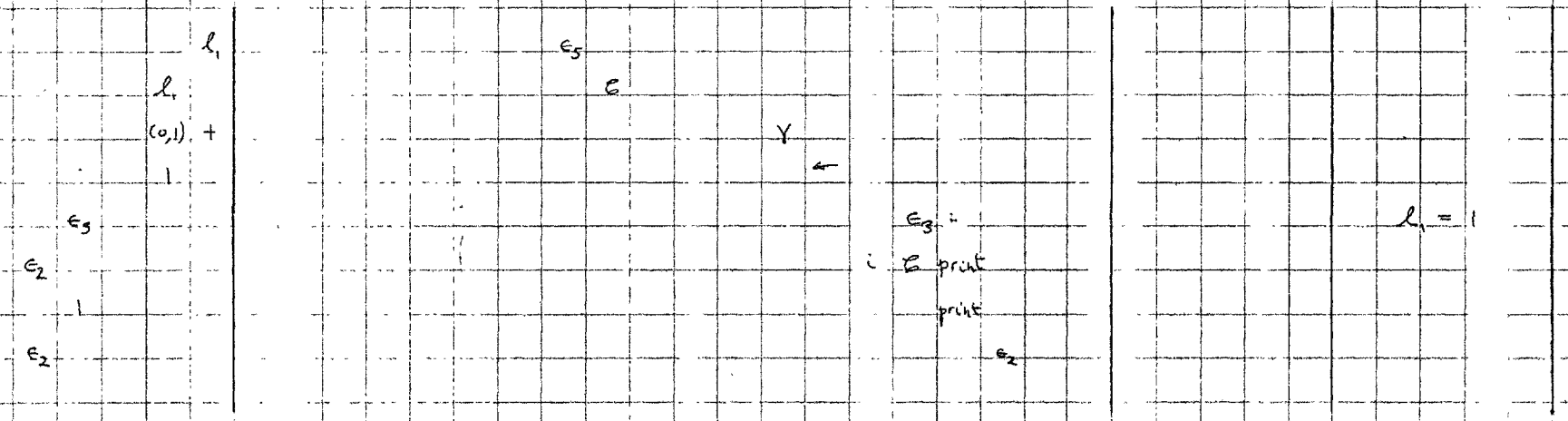


figure A25

$\delta_1 \equiv \lambda_k T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda y . \lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec} g . T_0 g 6 \gamma \delta_5$

$\lambda_4 \equiv \lambda \pi . y$ read read $T_2 + \gamma \leftarrow \nu$

$\delta_5 \equiv y$ 6 print

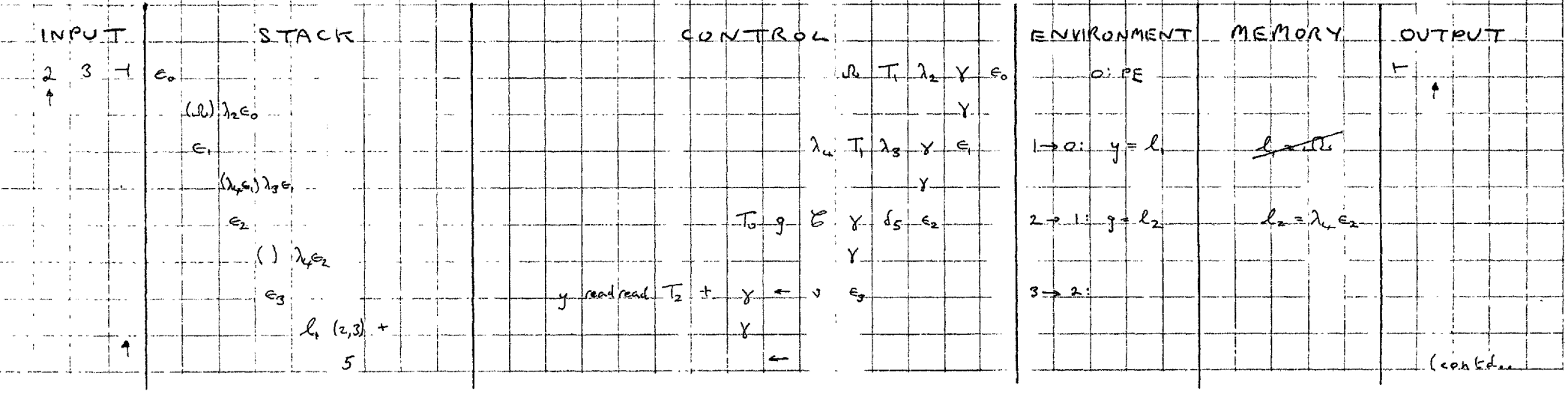


figure A25 continued

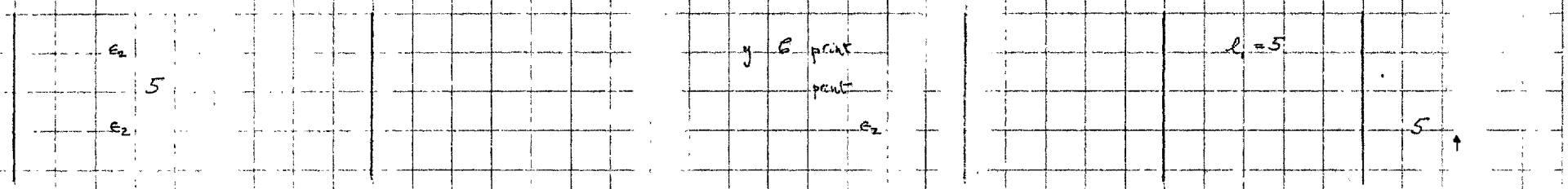


figure A26

$\delta_1 \equiv \Omega\ T_1\ \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda x.\ \lambda_4\ \lambda_9\ T_2\ \lambda_3\ \gamma$

$\lambda_3 \equiv \lambda_{rec}(p,q).\ x$ read $\leftarrow \nu\ \delta_{14}$

$\lambda_4 \equiv \lambda a.\ T_0\ \lambda_7\ \lambda_5$ read if γ

$\lambda_5 \equiv \lambda\pi.\ \lambda_6\ T_1\ p\ C\ \gamma$

$\lambda_6 \equiv \lambda\pi.\ T_0\ a\ C\ \gamma$

$\lambda_7 \equiv \lambda\pi.\ \lambda_8\ T_1\ q\ C\ \gamma$

$\lambda_8 \equiv \lambda\pi.\ T_0\ a\ C\ \gamma$

$\lambda_9 \equiv \lambda a.\ T_0\ \lambda_{13}\ \lambda_{10}$ read if γ

$\lambda_{10} \equiv \lambda\pi.\ \lambda_{11}\ T_1\ p\ C\ \gamma\ \delta_{12}$

$\lambda_{11} \equiv \lambda\pi.\ T_0\ a\ C\ \gamma$

$\delta_{12} \equiv T_0\ a\ C\ \gamma\ T_0\ a\ C\ \gamma\ C\ T_1 \sim \gamma \leftarrow \nu$

$\lambda_{13} \equiv \lambda\pi.$

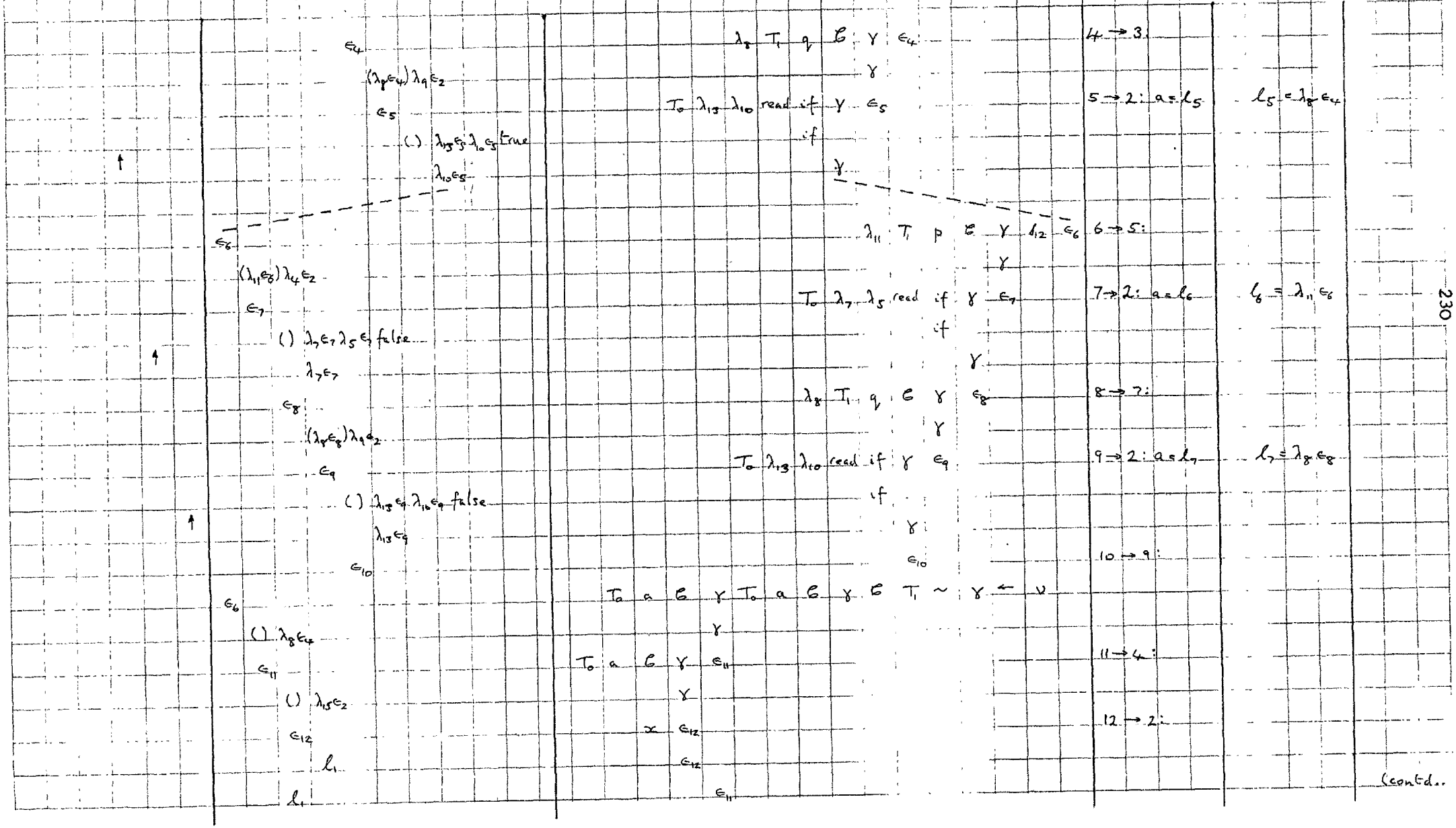
$\delta_{14} \equiv \lambda_{15}\ T_1\ p\ C\ \gamma\ \delta_{16}$

$\lambda_{15} \equiv \lambda\pi.\ x$

$\delta_{16} \equiv x\ C$ print

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
true false true false false ⊣	ϵ_0	$\Omega\ T_1\ \lambda_2\ \gamma$	ϵ_0	0: PE	⊢
	$(\Omega)\ \lambda_2 \epsilon_0$	γ			
	ϵ_1	$\lambda_4\ \lambda_9\ T_2\ \lambda_3\ \gamma$	ϵ_1	1→0: $x = l_1$	~~$l_1 = \Omega_1$~~
	$(\lambda_4 \epsilon_1, \lambda_9 \epsilon_1)\ \lambda_3 \epsilon_1$	γ			
	ϵ_2	x read $\leftarrow \nu\ \delta_{14}$	ϵ_2	2→1: $p = l_2$	$l_2 = \lambda_4 \epsilon_2$
	l_1 true	$\leftarrow$		$q = l_3$	$l_3 = \lambda_9 \epsilon_2$
	ϵ_2	$\lambda_{15}\ T_1\ p\ C\ \gamma\ \delta_{16}$			~~$l_1 = \text{true}$~~
	$(\lambda_{15} \epsilon_2)\ \lambda_4 \epsilon_2$	γ			
	ϵ_3	$T_0\ \lambda_7\ \lambda_5$ read if γ	ϵ_3	3→2: $a = l_4$	$l_4 = \lambda_5 \epsilon_2$
	() $\lambda_7 \epsilon_3\ \lambda_5 \epsilon_3$ false	if			
	$\lambda_7 \epsilon_3$	γ			(contd.

figure A26 continued



(contd..

figure A26 continued (2)

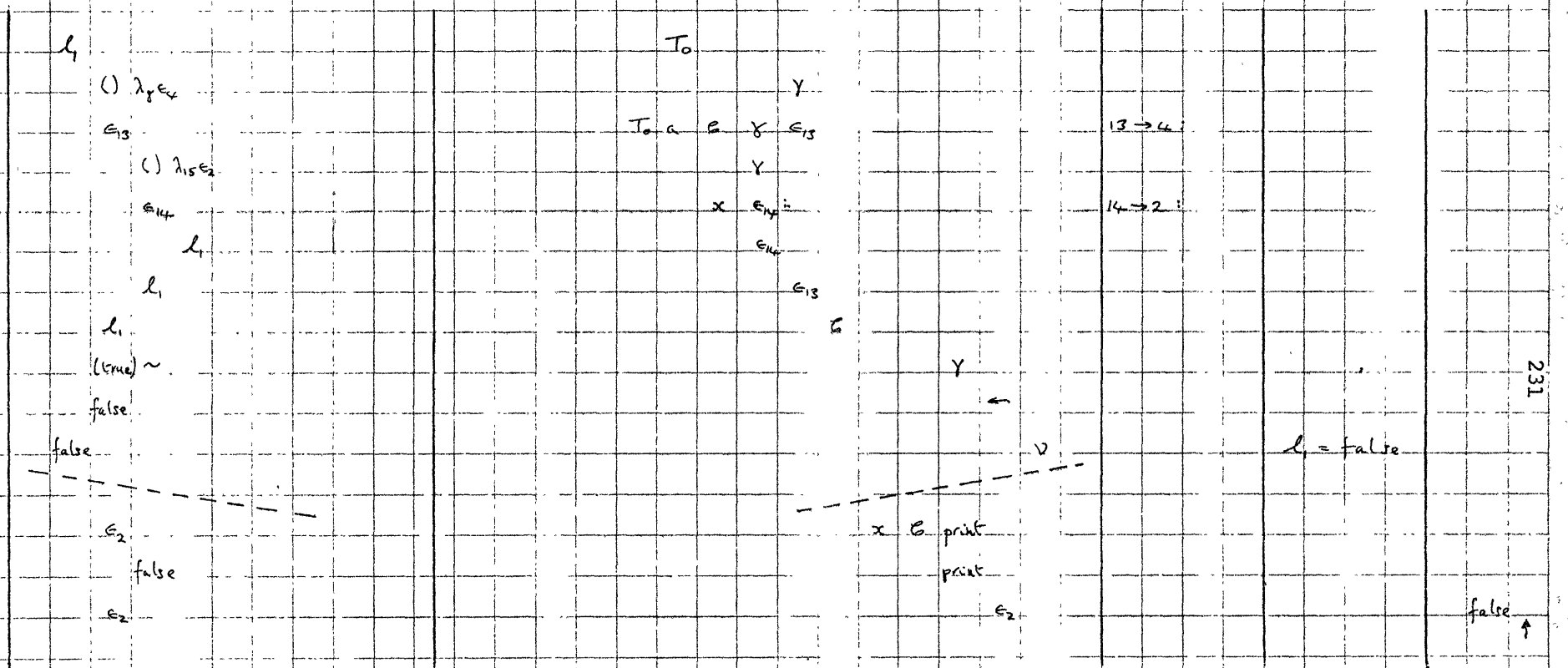


figure A 27

$\delta_1 \equiv \omega\ T_1\ \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda_3 \cdot \lambda_4\ T_1\ \lambda_3\ \gamma$

$\lambda_3 \equiv \lambda_{rec}\, x.\ 3\ 0 \leftarrow \nu\ \delta_8$

$\lambda_4 \equiv \lambda\pi . \omega\ T_1\ \lambda_5\ \gamma$

$\lambda_5 \equiv \lambda x . x \# 3\ 3\ 6\ 1\ T_2 + \gamma \leftarrow \nu\ \delta_6$

$\delta_6 \equiv x \# 3\ 6 \leftarrow \nu\ \delta_7$

$\delta_7 \equiv x \# 6$

$\delta_8 \equiv T_0\ x\ 6\ \gamma\ T_0\ x\ 6\ \gamma\ T_2 = \gamma$ print

LINE NO.	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
1	ϵ_0	$\omega\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	OPE		⊢
2	$(\omega)\ \lambda_2 \epsilon_0$	γ			↑
3	ϵ_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ \epsilon_1$	$1 \to \omega : 3 = l_1$	~~$l_1 = \omega$~~	
4	$(\lambda_4 \epsilon_1) \lambda_3 \epsilon_1$	γ			
5	ϵ_2	$3\ 0 \leftarrow \nu\ \delta_8\ \epsilon_2$	$2 \to 1 : x = l_2$	$l_2 = \lambda_4 \epsilon_2$	
6	$l_1\ 0$	$\leftarrow$			
7	ϵ_2	$T_0\ x\ 6\ \gamma\ T_0\ x\ 6\ \gamma\ T_2 = \gamma$ print		~~$l_1 = 0$~~	
8	$() \lambda_4 \epsilon_2$	γ			
9	ϵ_3	$\omega\ T_1\ \lambda_5\ \gamma\ \epsilon_3$	$3 \to 2 :$		
10	$() \lambda_5 \epsilon_3$	γ			
11	ϵ_4	$3\ 3\ 6\ 1\ T_2 + \gamma \leftarrow \nu\ \delta_6\ \epsilon_4$	$4 \to 3 : x \# l_3$	~~$l_3 = \omega$~~	
12	$l_1\ (\omega, 1) +$	γ			
13	1	$\leftarrow$			
14	ϵ_4	$x \# 3\ 6 \leftarrow \nu\ \delta_7$		~~$l_1 = T$~~	
15	$l_3\ 1$	$\leftarrow$			
16	ϵ_4	$x \# 6$		$l_3 = 1$	
17	1	ϵ_4			
18	1	T_0			
19	$() \lambda_4 \epsilon_2$	γ			
20	ϵ_5	$\omega\ T_1\ \lambda_5\ \gamma\ \epsilon_5$	$5 \to 2 :$		(contd.

figure A27 continued

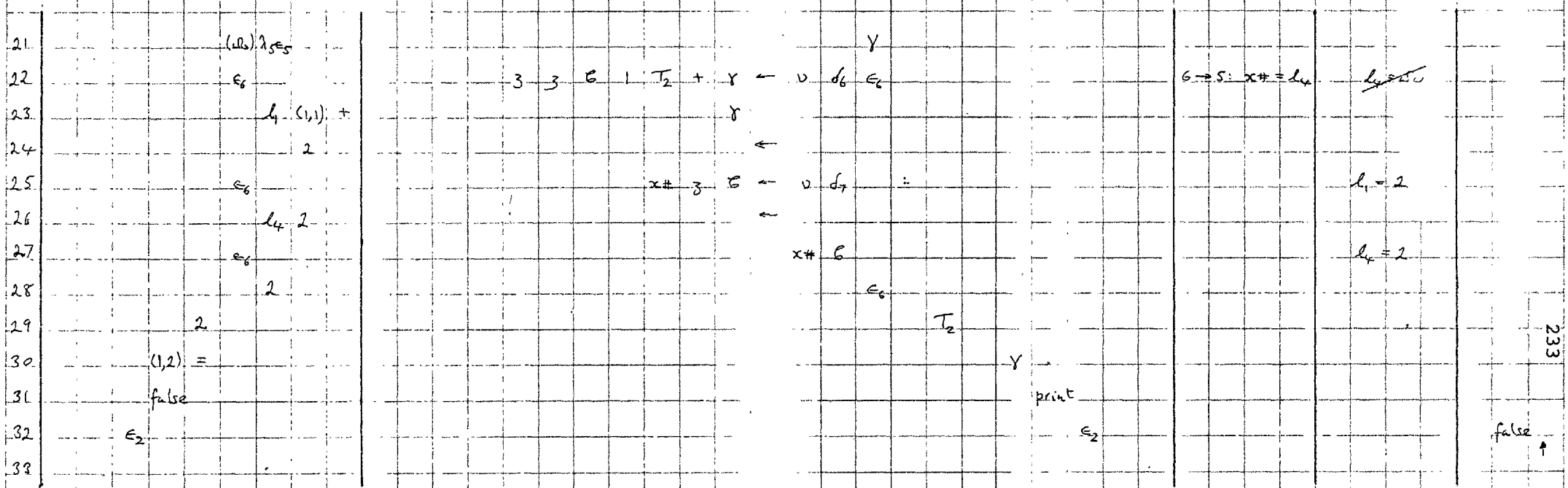


figure A28 (example 1)

$\delta_1 \equiv \Delta_3\, T_1\, \lambda_2\, \gamma$

$\lambda_2 \equiv \lambda_{rec}\, l.\, \delta_3$

$\delta_3 \equiv \Delta_9\, \lambda_5\, T_2\, \lambda_4\, \gamma$

$\lambda_4 \equiv \lambda_{rec}\, (l,p).\, \lambda_8\, T_1\, p\, \mathcal{E}\, \gamma$

$\lambda_5 \equiv \lambda x.\, \Delta_7\, T_1\, \lambda_6\, \gamma$

$\lambda_6 \equiv \lambda_{rec}\, l.\, \delta_7$

$\delta_7 \equiv T_0 \times \mathcal{E}\, \gamma\, \mathcal{E}\, J$

$\lambda_8 \equiv \lambda_{T_1}.\, l$

$\delta_9 \equiv$

STACK	CONTROL	ENVIRONMENT	MEMORY
ϵ_0	$\Delta_3\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	0: PE	
$(\delta_3 \epsilon_0)\ \lambda_2 \epsilon_0$	γ		
ϵ_1	$\delta_3\ \epsilon_1$	$1 \to 0$: $l = l_1$	$l_1 = \delta_3\ \epsilon_1$
ϵ_1	$\Delta_9\ \lambda_5\ T_2\ \lambda_4\ \gamma$		
$(\delta_9 \epsilon_1, \lambda_5 \epsilon_1)\ \lambda_4 \epsilon_1$	γ		
ϵ_2	$\lambda_8\ T_1\ p\ \mathcal{E}\ \gamma\ \epsilon_2$	$2 \to 1$: $l = l_2$	$l_2 = \delta_9\ \epsilon_2$
$(\lambda_8 \epsilon_2)\ \lambda_5 \epsilon_2$	γ	$p = l_3$	$l_3 = \lambda_5\ \epsilon_2$
ϵ_3	$\Delta_7\ T_1\ \lambda_6\ \gamma\ \epsilon_3$	$3 \to 2$: $x = l_4$	$l_4 = \lambda_8\ \epsilon_2$
$(\delta_7 \epsilon_3)\ \lambda_6 \epsilon_3$	γ		
ϵ_4	$\delta_1\ \epsilon_4$	$4 \to 3$: $l = l_5$	$l_5 = \delta_7\ \epsilon_4$
ϵ_4	$T_0 \times \mathcal{E}\ \gamma\ \mathcal{E}\ J$		
$(\ )\ \lambda_8 \epsilon_2$	γ		
ϵ_5	$l\ \epsilon_5$	$5 \to 2$:	
l_2	ϵ_5		
l_2	$\mathcal{E}$		
$\delta_9 \epsilon_2$	J		
ϵ_2	ϵ_2		
ϵ_1	ϵ_1		
ϵ_0	ϵ_0		

figure A29 (example 2)

$\delta_1 \equiv \omega_1 \omega_2 T_2 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda(a, b).\ \lambda_4 T_1 \lambda_3 \gamma$

$\lambda_3 \equiv \lambda_{rec} p.\ a\ b\ \text{false} \leftarrow \leftarrow \upsilon\ \delta_5$

$\lambda_4 \equiv \lambda(x, y).\ T_0\ x\ \mathcal{E}\ \gamma\ y\ T_0\ x\ \mathcal{E}\ \gamma\ \mathcal{E}\ y\ \mathcal{E}\ T_2 \supset \gamma \leftarrow \leftarrow \upsilon$

$\delta_5 \equiv \lambda_6\ b\ \mathcal{E}\ T_2\ p\ \mathcal{E}\ \gamma\ \delta_7$

$\lambda_6 \equiv \lambda \pi.\ a$

$\delta_7 \equiv a\ \mathcal{E}\ \text{print}\ b\ \mathcal{E}\ \text{print}$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_0	$\vdots\ \omega_1\ \omega_2\ T_2\ \lambda_2\ \gamma\ \epsilon_0$	OPE		⊢
$(\omega_1, \omega_2)\ \lambda_2\ \epsilon_0$	γ			
ϵ_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ \epsilon_1$	$1 \to 0$: $a = l_1$	~~$l_1 = \omega_1$~~	
$(\lambda_4, \epsilon_1)\ \lambda_3\ \epsilon_1$	γ	$b = l_2$	~~$l_2 = \omega_2$~~	
ϵ_2	$a\ b\ \text{false} \leftarrow \leftarrow \upsilon\ \delta_5\ \epsilon_2$	$2 \to 1$: $p = l_3$	$l_3 = \lambda_4, \epsilon_2$	
$l_1\ l_2\ \text{false}$	$\leftarrow$			
ϵ_2	$\lambda_6\ b\ \mathcal{E}\ T_2\ p\ \mathcal{E}\ \gamma\ \delta_7$		~~$l_1 = \text{false}$~~	
$(\lambda_6, \epsilon_2, \text{false})\ \lambda_4\ \epsilon_2$	γ		$l_2 = \text{false}$	
ϵ_3	$T_0\ x\ \mathcal{E}\ \gamma\ y\ T_0\ x\ \mathcal{E}\ \gamma\ \mathcal{E}\ y\ \mathcal{E}\ T_2 \supset \gamma \leftarrow \leftarrow \upsilon\ \epsilon_3$	$3 \to 2$: $x = l_4$	$l_4 = \lambda_6, \epsilon_2$	
$()\ \lambda_6, \epsilon_2$	γ	$y = l_5$	~~$l_5 = \text{false}$~~	
ϵ_4	$a\ \epsilon_4$	$4 \to 2$:		
l_1	ϵ_4			
l_1	y			
$l_5\ ()\ \lambda_6, \epsilon_2$	γ			
ϵ_5	$a\ \epsilon_5$	$5 \to 2$:		
l_1	ϵ_5			
l_1	$\mathcal{E}$			
$(\text{false}, \text{false}) \supset$	γ			
true	$\leftarrow$			
ϵ_2	$a\ \mathcal{E}\ \text{print}\ b\ \mathcal{E}\ \text{print}$		$l_5 = \text{true}$	(cont'd...)

figure A29 continued

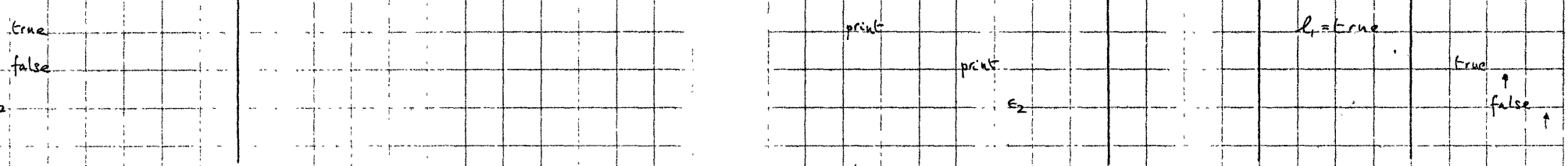


figure A30 (example 3)

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec} p.\, read\ read\ T_2\ p\ \mathcal{E}\ \gamma\ print$

$\lambda_3 \equiv \lambda(x,y).\, \Omega_0\ T_1\ \lambda_4\ \gamma$

$\lambda_4 \equiv \lambda p\#.\, p\#\ T_0\ \lambda_6\ \lambda_5\ y\ \mathcal{E}\ |\ T_2 = \gamma\ if\ \gamma \leftarrow v\ \delta_7$

$\lambda_5 \equiv \lambda \pi.\, x\ \mathcal{E}$

$\lambda_6 \equiv \lambda \pi.\, x\ \mathcal{E}\ y\ \mathcal{E}\ |\ T_2 = \gamma\ T_2\ p\ \mathcal{E}\ \gamma\ x\ \mathcal{E}\ T_2 \uparrow \gamma$

$\delta_7 \equiv p\#\ \mathcal{E}$

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
1 2 ⊣	ϵ_0	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_0$			⊢
	$(\lambda_3, \epsilon_0)\ \lambda_2\ \epsilon_0$	γ			
	ϵ_1	$read\ read\ T_2\ p\ \mathcal{E}\ \gamma\ print\ \epsilon_1$	1→0: $p = l_1$	$l_1 = \lambda_3\ \epsilon_1$	
	$(2,2)\ \lambda_3\ \epsilon_1$	γ			
	ϵ_2	$\Omega_0\ T_1\ \lambda_4\ \gamma\ \epsilon_2$	2→1: $x = l_2$	$l_2 = 2$	
	$(\Omega_0)\ \lambda_4\ \epsilon_2$	γ	$y = l_3$	$l_3 = 2$	
	ϵ_3	$p\#\ T_0\ \lambda_6\ \lambda_5\ y\ \mathcal{E}\ |\ T_2 = \gamma\ if\ \gamma \leftarrow v\ \delta_7\ \epsilon_3$	3→2: $p\# = l_4$	~~$l_4 = \Omega_0$~~	
	$l_4\ ()\ \lambda_6\ \epsilon_3\ (2,1) =$	γ			
	false	if			
	$\lambda_6\ \epsilon_3$	γ			
	⋮	⋮			
	ϵ_4	$x\ \mathcal{E}\ y\ \mathcal{E}\ |\ T_2 = \gamma\ T_2\ p\ \mathcal{E}\ \gamma\ x\ \mathcal{E}\ T_2 \uparrow \gamma\ \epsilon_4$	4→3:		
	$2\ (2,1) =$	γ			(contd.

figure A30 continued

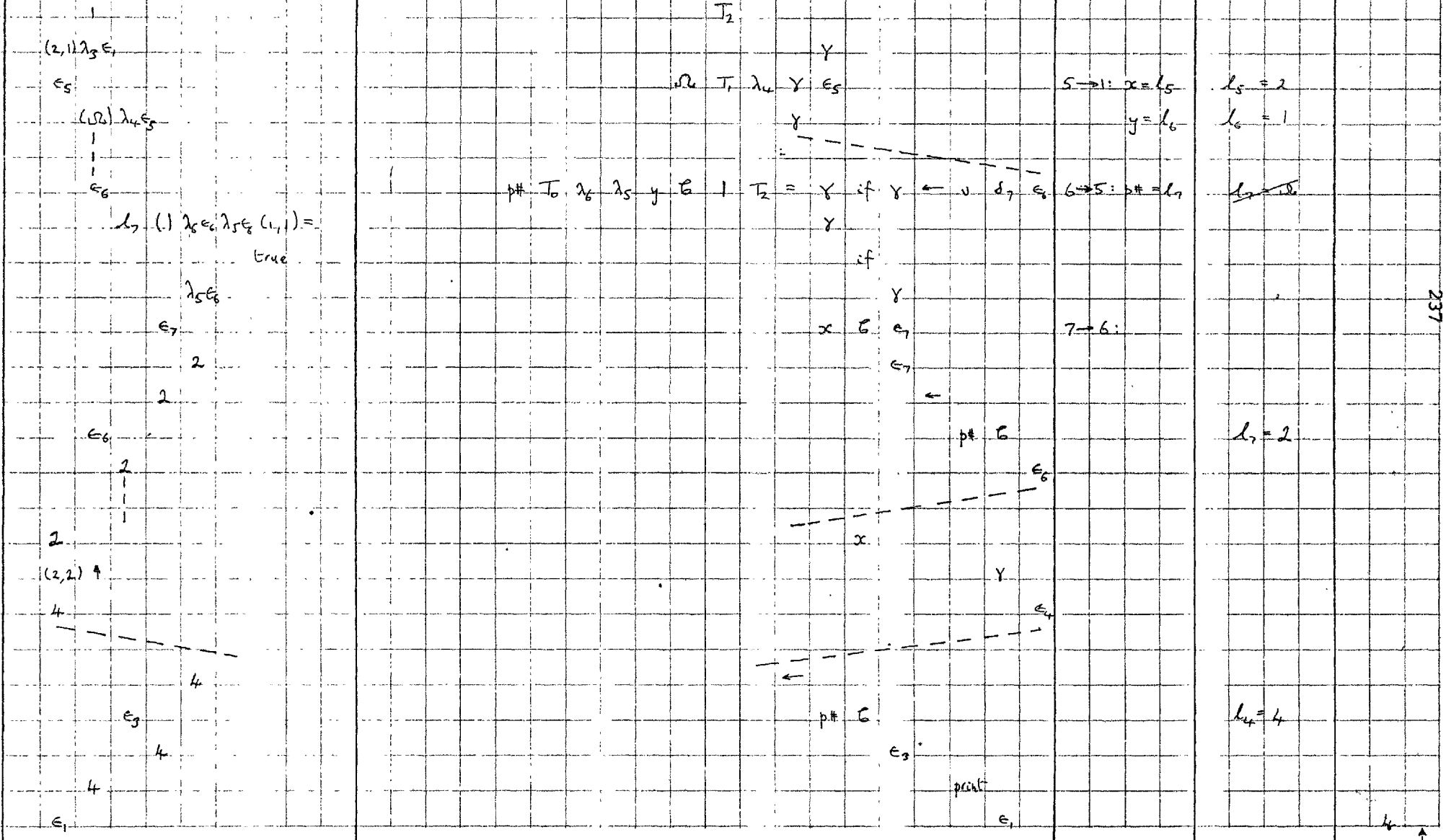


figure A31 (example 4)

$\delta_1 \equiv \lambda_3\, T_1\, \lambda_2\, \gamma$

$\lambda_2 \equiv \lambda_{rec}\, p.\, T_0\, p\, \beta\, \gamma\, print$

$\lambda_3 \equiv \lambda\pi.\, \Omega_1 \Omega_2\, T_2\, \lambda_4\, \gamma$

$\lambda_4 \equiv \lambda(p\#, a).\ a\ read \leftarrow \nu\ \delta_5$

$\delta_5 \equiv p\#\ T_0\ \lambda_7 \lambda_6\ a\ \beta\ 0\ T_2 = \gamma\ if\ \gamma \leftarrow \nu\ \delta_8$

$\lambda_6 \equiv \lambda\pi.\, T_0\, p\, \beta\, \gamma$

$\lambda_7 \equiv \lambda\pi.\, a\, \beta$

$\delta_8 \equiv p\#\ \beta$

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
0 5 ⊣	ϵ_0	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	0 PE		⊢
↑	$(\lambda_3 \epsilon_0)\ \lambda_2\ \epsilon_0$	γ			↑
	ϵ_1	$T_0\ p\ \beta\ \gamma\ print\ \epsilon_1$	$1 \to 0$: $p = l_1$	$l_1 = \lambda_3 \epsilon_1$	
	$()\ \lambda_3\ \epsilon_1$	γ			
	ϵ_2	$\Omega_1\ \Omega_2\ T_2\ \lambda_4\ \gamma\ \epsilon_2$	$2 \to 1$:		
	$(\Omega_1, \Omega_2)\ \lambda_4\ \epsilon_2$	γ			
	ϵ_3	$a\ read \leftarrow \nu\ \delta_5\ \epsilon_3$	$3 \to 2$: $p\# = l_2$	~~$l_2 = \Omega_1$~~	
	$l_3\ 0$	$\leftarrow$	$a = l_3$	~~$l_3 = \Omega_2$~~	
↑	ϵ_3	$p\#\ T_0\ \lambda_7\ \lambda_6\ a\ \beta\ 0\ T_2 = \gamma\ if\ \gamma \leftarrow \nu\ \delta_8$		$l_3 = 0$	
	$l_2\ ()\ \lambda_7 \epsilon_3\ \lambda_6 \epsilon_3\ (0,0) =$	γ			
	$true$	if			
	$\lambda_6 \epsilon_3$	γ			
	ϵ_4	$T_0\ p\ \beta\ \gamma\ \epsilon_4$	$4 \to 3$:		
	$()\ \lambda_3 \epsilon_1$	γ			
	ϵ_5	$\Omega_1\ \Omega_2\ T_2\ \lambda_4\ \gamma\ \epsilon_5$	$5 \to 1$:		
	$(\Omega_1, \Omega_2)\ \lambda_4\ \epsilon_5$	γ			
	ϵ_6	$a\ read \leftarrow \nu\ \delta_5\ \epsilon_6$	$6 \to 5$: $p\# = l_4$	~~$l_4 = \Omega_1$~~	
	$l_5\ 5$	$\leftarrow$	$a = l_5$	~~$l_5 = \Omega_2$~~	
↑	ϵ_6	$p\#\ T_0\ \lambda_7\ \lambda_6\ a\ \beta\ 0\ T_2 = \gamma\ if\ \gamma \leftarrow \nu\ \delta_8$		$l_5 = 5$	(cont'd..

figure A31 continued

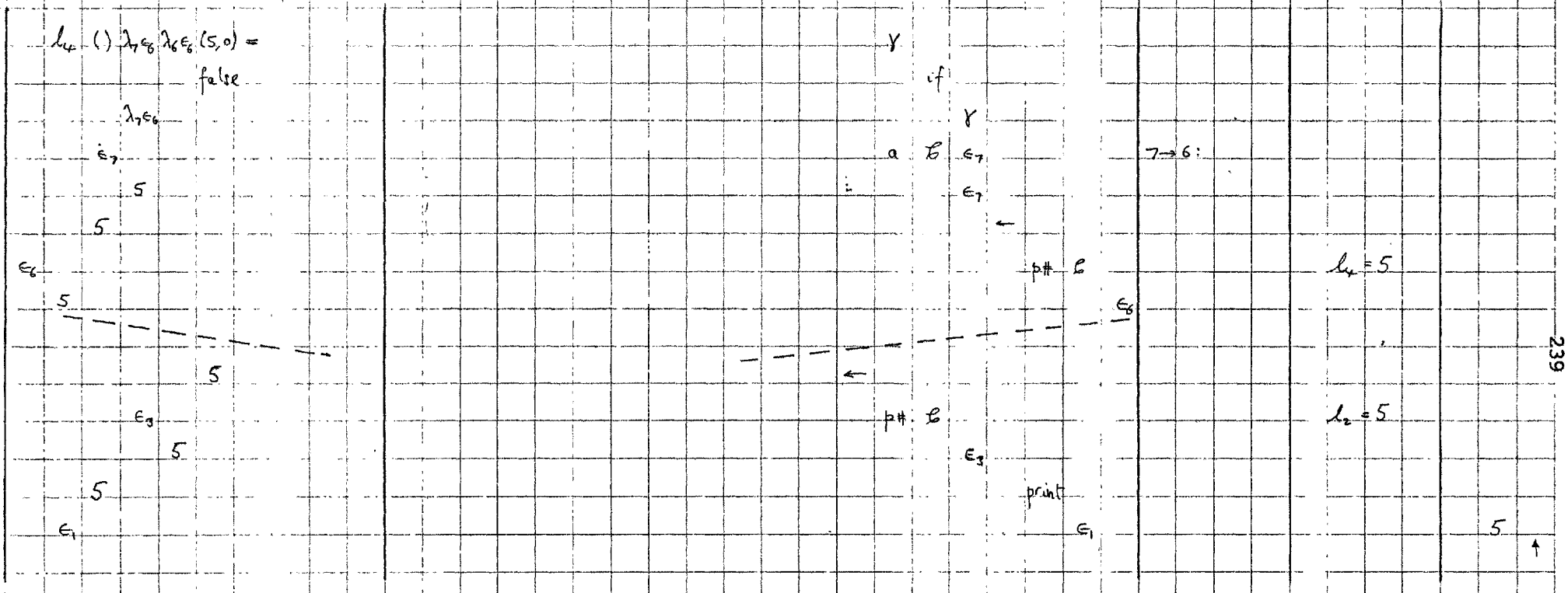


figure A32 (example 5)

$\delta_1 \equiv \Omega\ T_1 \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda k.\ \lambda_4\ T_1\ \lambda_3\ \gamma$

$\lambda_3 \equiv \lambda_{rec}\ p.\ k\ 2 \leftarrow \nu\ \delta_8$

$\lambda_4 \equiv \lambda a.\ \Omega\ T_1\ \lambda_5\ \gamma$

$\lambda_5 \equiv \lambda p\#.\ k\ 5 \leftarrow \nu\ \delta_6$

$\delta_6 \equiv p\#\ a\ e \leftarrow \nu\ \delta_7$

$\delta_7 \equiv p\#\ e$

$\delta_8 \equiv k\ e\ T_1\ p\ e\ \gamma\ print$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
e_0	$\Omega\ T_1\ \lambda_2\ \gamma\ e_0$			⊢
$(\Omega)\ \lambda_2 e_0$	γ			↑
e_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ e_1$	1→0: $k = l_1$	~~$l_1 = \Omega$~~	
$(\lambda_4 e_1)\ \lambda_3 e_1$	γ			
e_2	$k\ 2 \leftarrow \nu\ \delta_8\ e_2$	2→1: $p = l_2$	$l_2 = \lambda_4 e_2$	
$l_1\ 2$	$\leftarrow$			
e_2	$k\ e\ T_1\ p\ e\ \gamma$ print		~~$l_1 = 2$~~	
$(2)\ \lambda_4 e_2$	γ			
e_3	$\Omega\ T_1\ \lambda_5\ \gamma\ e_3$	3→2: $a = l_3$	$l_3 = 2$	
$(\Omega)\ \lambda_5 e_3$	γ			
e_4	$2\ 5 \leftarrow \nu\ \delta_6\ e_4$	4→3: $p\# = l_4$	~~$l_4 = \Omega$~~	
$l_1\ 5$	$\leftarrow$			
e_4	$p\#\ a\ e \leftarrow \nu\ \delta_7$		$l_1 = 5$	
$l_4\ 2$	$\leftarrow$			
e_4	$p\#\ e$		$l_4 = 2$	
2	e_4			
2	print			2 ↑
e_2	e_2			

figure A33 (example 6)

$\delta_1 \equiv \Omega\, T_1\, \lambda_2\, \gamma$

$\lambda_2 \equiv \lambda k.\ \lambda_4\, T_1\, \lambda_3\, \gamma$

$\lambda_3 \equiv \lambda_{recp}.\ k\,2 \leftarrow v\ \delta_8$

$\lambda_4 \equiv \lambda a.\ \Omega\, T_1\, \lambda_5\, \gamma$

$\lambda_5 \equiv \lambda p\#.\ k\,5 \leftarrow v\ \delta_6$

$\delta_6 \equiv p\#\ T_0\ a\ \varepsilon\ \gamma\ \varepsilon \leftarrow v\ \delta_7$

$\delta_7 \equiv p\#\ \varepsilon$

$\delta_8 \equiv \lambda_9\, T_1\, p\ \varepsilon\ \gamma\ \text{print}$

$\lambda_9 \equiv \lambda \pi.\ k$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_0	$\Omega\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	0 : PE		⊢ ↑
$(\Omega)\ \lambda_2 \epsilon_0$	γ			
ϵ_1	$\lambda_4\ T_1\ \lambda_3\ \gamma\ \epsilon_1$	$1 \to 0$: $k = l_1$	~~$l_1 = \Omega$~~	
$(\lambda_4 \epsilon_1)\ \lambda_3 \epsilon_1$	γ			
ϵ_2	$k\ 2 \leftarrow v\ \delta_8\ \epsilon_2$	$2 \to 1$: $p = l_2$	$l_2 = \lambda_4 \epsilon_2$	
$l_1\ 2$	$\leftarrow$			
ϵ_2	$\lambda_9\ T_1\ p\ \varepsilon\ \gamma\ \text{print}$		~~$l_1 = 1$~~	
$(\lambda_4 \epsilon_2)\ \lambda_4 \epsilon_2$	γ			
ϵ_3	$\Omega\ T_1\ \lambda_5\ \gamma\ \epsilon_3$	$3 \to 2$: $a = l_3$	$l_3 = \lambda_9 \epsilon_2$	
$(\Omega)\ \lambda_5 \epsilon_3$	γ			
ϵ_4	$k\ 5 \leftarrow v\ \delta_6\ \epsilon_4$	$4 \to 3$: $p\# = l_4$	~~$l_4 = \Omega$~~	
$l_1\ 5$	$\leftarrow$			
ϵ_4	$p\#\ T_0\ a\ \varepsilon\ \gamma\ \varepsilon \leftarrow v\ \delta_7$		$l_1 = 5$	
$l_4\ (\ )\ \lambda_9 \epsilon_2$	γ			
ϵ_5	$k\ \epsilon_5$	$5 \to 2$:		
l_1	ϵ_5			
l_1	ε			
5	$\leftarrow$			
ϵ_4	$p\#\ \varepsilon$		$l_4 = 5$	(cont'd..

figure A33 continued

5

5

ϵ_2

ϵ_4

print

ϵ_3

5

figure A34 (example 7)

$\delta_1 \equiv \Omega\,\Omega\,T_2\,\lambda_2\,\gamma$

$\lambda_2 \equiv \lambda(a,b).\,\Delta_{22}\,\lambda_4\,T_2\,\lambda_3\,\gamma$

$\lambda_3 \equiv \lambda_{rec}(l,f).\,a\,b\,o \leftarrow \leftarrow \nu\,\delta_{20}$

$\lambda_4 \equiv \lambda(i,x).\,\Delta_6\,T_1\,\lambda_5\,\gamma$

$\lambda_5 \equiv \lambda_{rec} m.\,b\,b\,\mathcal{C}1\,T_2 + \gamma \leftarrow \nu\,d_7$

$\delta_6 \equiv b\,b\,\mathcal{C}1\,T_2 + \gamma \leftarrow \nu\,\delta_7$

$\delta_7 \equiv i\,i\,\mathcal{C}1\,T_2 + \gamma \leftarrow \nu\,\delta_8$

$\delta_8 \equiv T_0\,\lambda_{11}\,\lambda_9\,b\,\mathcal{C}4\,T_2 = \gamma\,if\,\gamma\,\delta_{12}$

$\lambda_9 \equiv \lambda\pi.\,a\,i\,\mathcal{C} \leftarrow \nu\,\delta_{10}$

$\delta_{10} \equiv l\,\mathcal{C}\,J$

$\lambda_{11} \equiv \lambda\pi.$

$\delta_{12} \equiv T_0\,\lambda_{14}\,\lambda_{13}\,b\,\mathcal{C}3\,T_2 = \gamma\,if\,\gamma\,\delta_{15}$

$\lambda_{13} \equiv \lambda\pi\,T_0\,x\,\mathcal{C}\,\gamma\,\mathcal{C}J$

$\lambda_{14} \equiv \lambda\pi.$

$\delta_{15} \equiv T_0\,\lambda_{18}\,\lambda_{16}\,b\,\mathcal{C}2\,T_2 = \gamma\,if\,\gamma$

$\lambda_{16} \equiv \lambda\pi.\,i\,\mathcal{C}\,\lambda_{17}\,T_2\,f\,\mathcal{C}\,\gamma$

$\lambda_{17} \equiv \lambda\pi.\,T_0\,x\,\mathcal{C}\,\gamma$

$\lambda_{18} \equiv \lambda\pi.\,i\,\mathcal{C}\,\lambda_{19}\,T_2\,f\,\mathcal{C}\,\gamma$

$\lambda_{19} \equiv \lambda\pi.\,m$

$\delta_{20} \equiv a\,\mathcal{C}\,\lambda_{21}\,T_2\,f\,\mathcal{C}\,\gamma\,\delta_{22}$

$\lambda_{21} \equiv \lambda\pi.\,l$

$\delta_{22} \equiv a\,\mathcal{C}\,\text{print}$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_0	$\Omega\ \Omega\ T_2\ \lambda_2\ \gamma\ \epsilon_0$	OPE		⊢
$(\Omega,\Omega)\,\lambda_2\,\epsilon_0$	γ			
ϵ_1	$\Delta_{22}\ \lambda_4\ T_2\ \lambda_3\ \gamma\ \epsilon_1$	1→0: $a = l_1$	~~$l_1 = \Omega$~~	
$(\delta_{22}\,\epsilon_1, \lambda_4\,\epsilon_1)\,\lambda_3\,\epsilon_1$	γ	$b = l_2$	~~$l_2 = \Omega$~~	
ϵ_2	$a\ b\ o \leftarrow \leftarrow \nu\ \delta_{20}\ \epsilon_2$	2→1: $l = l_3$	$l_3 = \delta_{22}\,\epsilon_2$	
$l_1\ l_2\ o$	$\leftarrow$	$f = l_4$	$l_4 = \lambda_4\,\epsilon_2$	
ϵ_2	$a\ \mathcal{C}\ \lambda_{21}\ T_2\ f\ \mathcal{C}\ \gamma\ \delta_{22}$		~~$l_2 = o$~~	
$(o, \lambda_{21}, \epsilon_2)\,\lambda_4\,\epsilon_2$	γ		~~$l_1 = o$~~	
ϵ_3	$\Delta_6\ T_1\ \lambda_5\ \gamma\ \epsilon_3$	3→2: $i = l_5$	~~$l_5 = o$~~	(contd.

figure A34 continued.

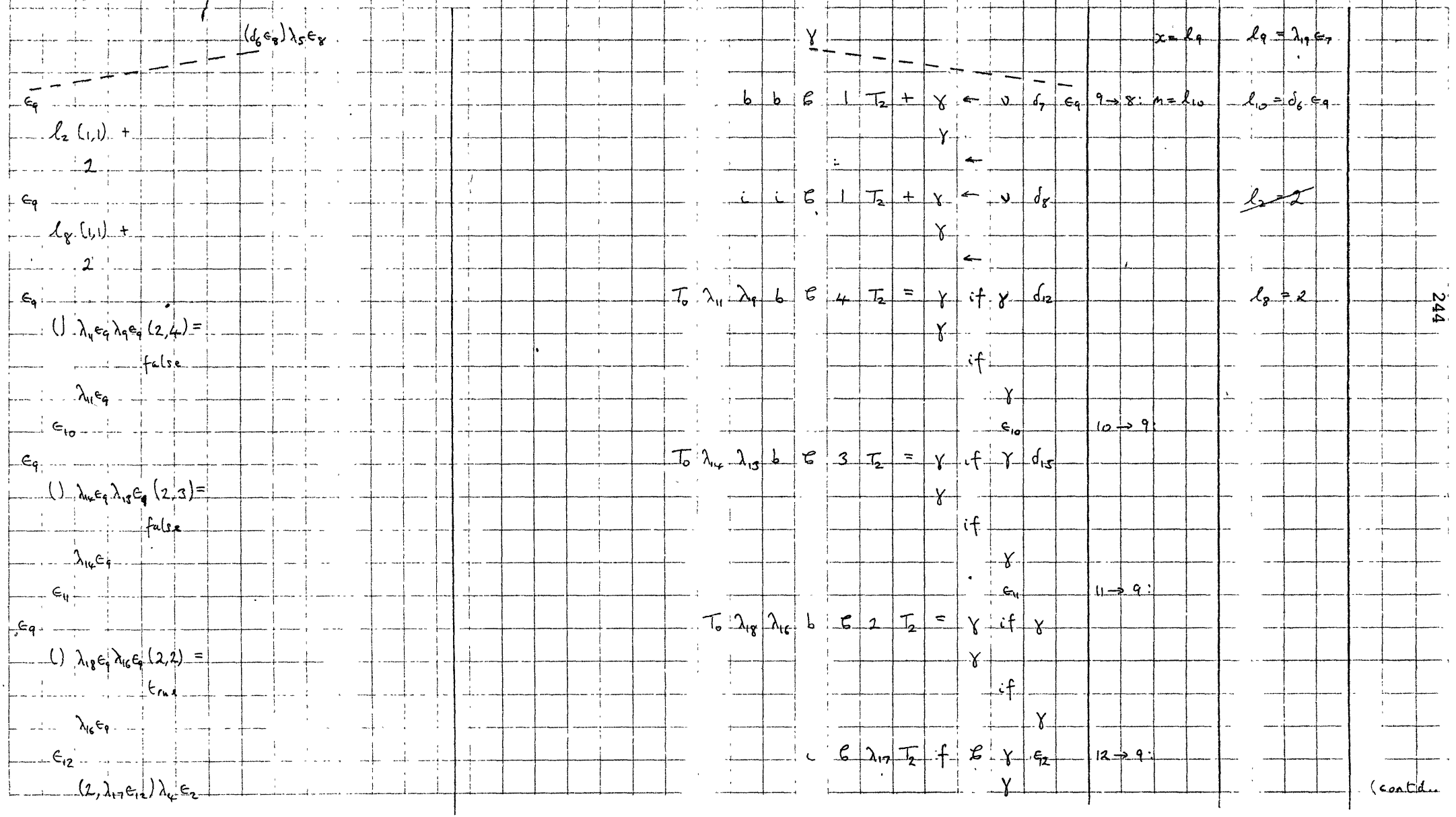
```
(δ6 ε3) λ5 ε3
ε4
l2 (0,1) +
1
ε4
l5 (0,1) +
1
ε4
() λ11 ε4 λ9 ε4 (1,4) =
false
λ11 ε4
ε5
ε4
() λ14 ε4 λ13 ε4 (1,3) =
false
λ14 ε4
ε6
ε4
() λ18 ε4 λ16 ε4 (1,2) =
false
λ18 ε4
ε7
(1, λ19 ε7) λ4 ε2
ε8
```

```
                                                                γ
b b 6 1 T2 + γ ← ν δ7 ε4
                γ
                ←
i i 6 1 T2 + γ ← ¬ν δ8
                γ
                ←
T0 λ11 λ9 b 6 4 T2 = γ if γ δ12
                γ
                if
                γ
                ε5
T0 λ14 λ13 b 6 3 T2 = γ if γ δ15
                γ
                if
                γ
                ε6
T0 λ18 λ16 b 6 2 T2 = γ if γ
                γ
                if
                γ
i 6 λ19 T2 f 6 γ ε7
                γ
Δ6 T1 λ5 γ ε8
```

```
                x = l6      l6 = λ21 ε2
4→3 : m = l7    l7 = δ6 ε4
                l2 = T
                l5 = T
5→4 :
6→4 :
7→4 :
8→2 : i = l8    l8 = T
```


(continued...

figure A34 continued (2)



(contid...

figure A34 continued (3)

```
ε13
(δ6 ε13) λ5 ε13
ε14
l2 (2,1) +
3
ε14
l11 (2,1) +
3
ε14
() λ11 ε14 λ9 ε14 (3,4) ÷
false
λ11 ε14
ε15
ε14
() λ14 ε14 λ13 ε14 (3,3) =
true
λ13 ε14
ε16
() λ17 ε12
ε17
() λ19 ε7
ε18
l7
l7
```

```
                                    Δ6 T1 λ5 γ ε13
                                    γ
b b ε 1 T2 + γ ← ν δ7 ε14
γ
:  ←
i i ε 1 T2 + γ ← ν δ8
γ
←
T0 λ11 λ9 b ε 4 T2 = γ if γ δ12
γ
if
γ
ε15
T0 λ14 λ13 b ε 3 T2 = γ if γ δ15
γ
if
γ
T0 x ε γ ε J ε16
γ
T0 x ε γ ε17
γ
m ε18
ε18
ε
```

```
13→2: i = l11      l11 = 1
      x = l12      l12 = λ17 ε12
14→13: m = l13     l13 = δ6 ε14
                   l2 = 3
                   l11 = 3
15→14:
16→14.
17→12:
18→7:
```


(cont'd...

figure A.34 continued (4)

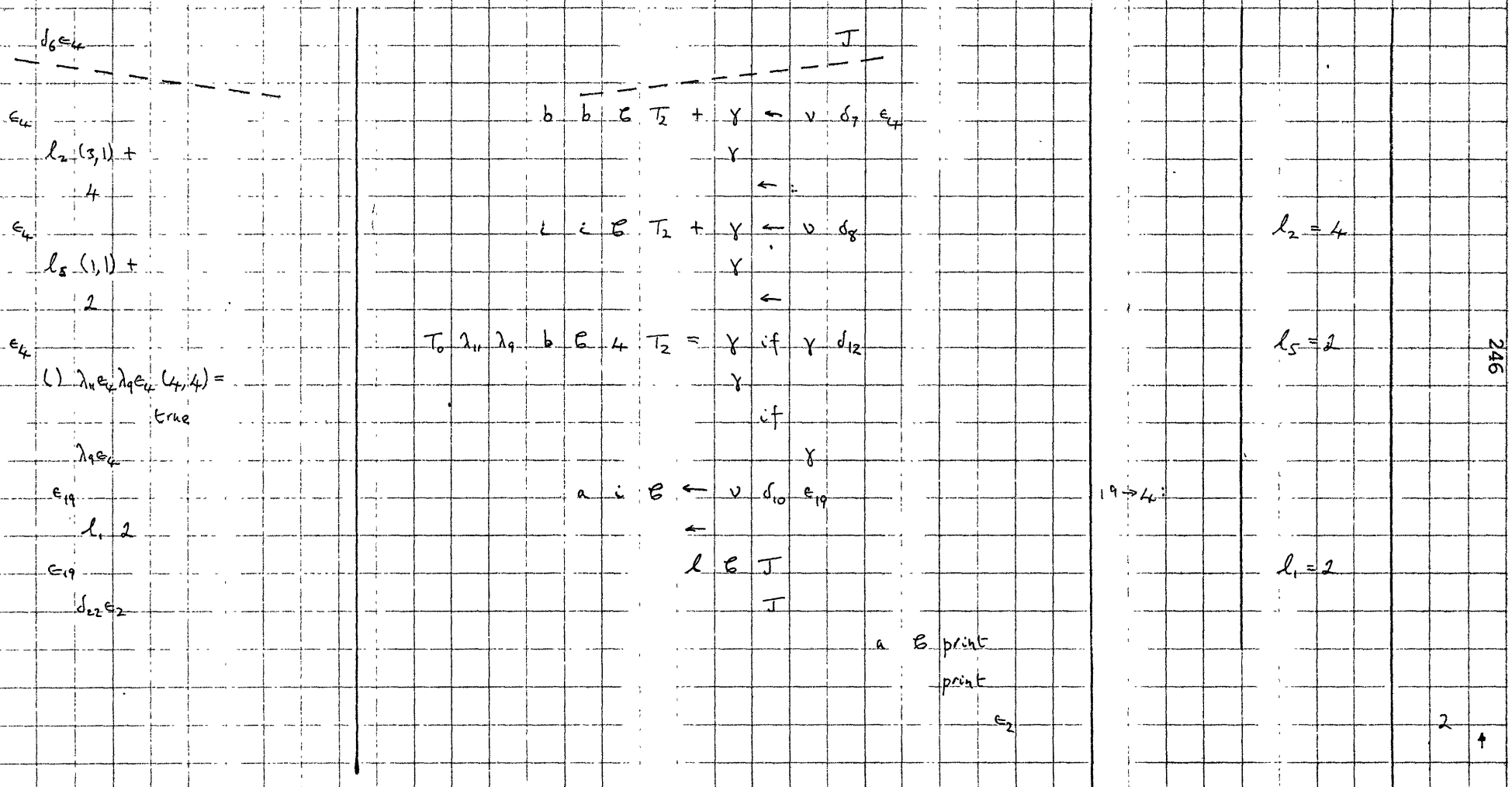


figure A.35 (example 8)

$\delta_1 \equiv \Omega\, T_1\, \lambda_2\, \gamma$

$\lambda_2 \equiv \lambda n.\, \Delta_{20}\, \lambda_4\, \lambda_{17}\, T_3\, \lambda_3\, \gamma$

$\lambda_3 \equiv \lambda_{rec}\, (l, p, q).\, n\, 2 \leftarrow \nu\, \delta_{18}$

$\lambda_4 \equiv \lambda (x, c).\, \delta_5\, \delta_{16}$

$\delta_5 \equiv \Delta_{15}\, \lambda_7\, T_2\, \lambda_6\, \gamma$

$\lambda_6 \equiv \lambda_{rec}\, (j, r).\, T_0\, \lambda_{13}\, \lambda_{11}\, c\, \mathcal{E}\, n\, \mathcal{E}\, T_2 > \gamma\, ;\, \gamma$

$\lambda_7 \equiv \lambda t.\, n\, n\, \mathcal{E}\, c\, \mathcal{E}\, T_2 + \gamma \leftarrow \nu.\, \delta_8$

$\delta_8 \equiv \lambda_9\, T_1\, T_0\, x\, \mathcal{E}\, \gamma\, \mathcal{E}\, \gamma\, \delta_{10}$

$\lambda_9 \equiv \lambda \pi.\, T_0 \vdash \mathcal{E}\, \gamma$

$\delta_{10} \equiv T_0 \vdash \mathcal{E}\, \gamma\, \mathcal{E}\, J$

$\lambda_{11} \equiv \lambda \pi.\, \lambda_{12}\, T_1\, T_0\, x\, \mathcal{E}\, \gamma\, \mathcal{E}\, \gamma$

$\lambda_{12} \equiv \lambda \pi.\, j$

$\lambda_{13} \equiv \lambda \pi.\, \lambda_{14}\, c\, \mathcal{E}\, 1\, T_2 + \gamma\, T_2\, p\, \mathcal{E}\, \gamma$

$\lambda_{14} \equiv \lambda \pi.\, c$

$\delta_{15} \equiv$

$\delta_{16} \equiv l\, \mathcal{E}\, J$

$\lambda_{17} \equiv \lambda t.\, n\, n\, \mathcal{E}\, 1\, T_2 + \gamma \leftarrow \nu$

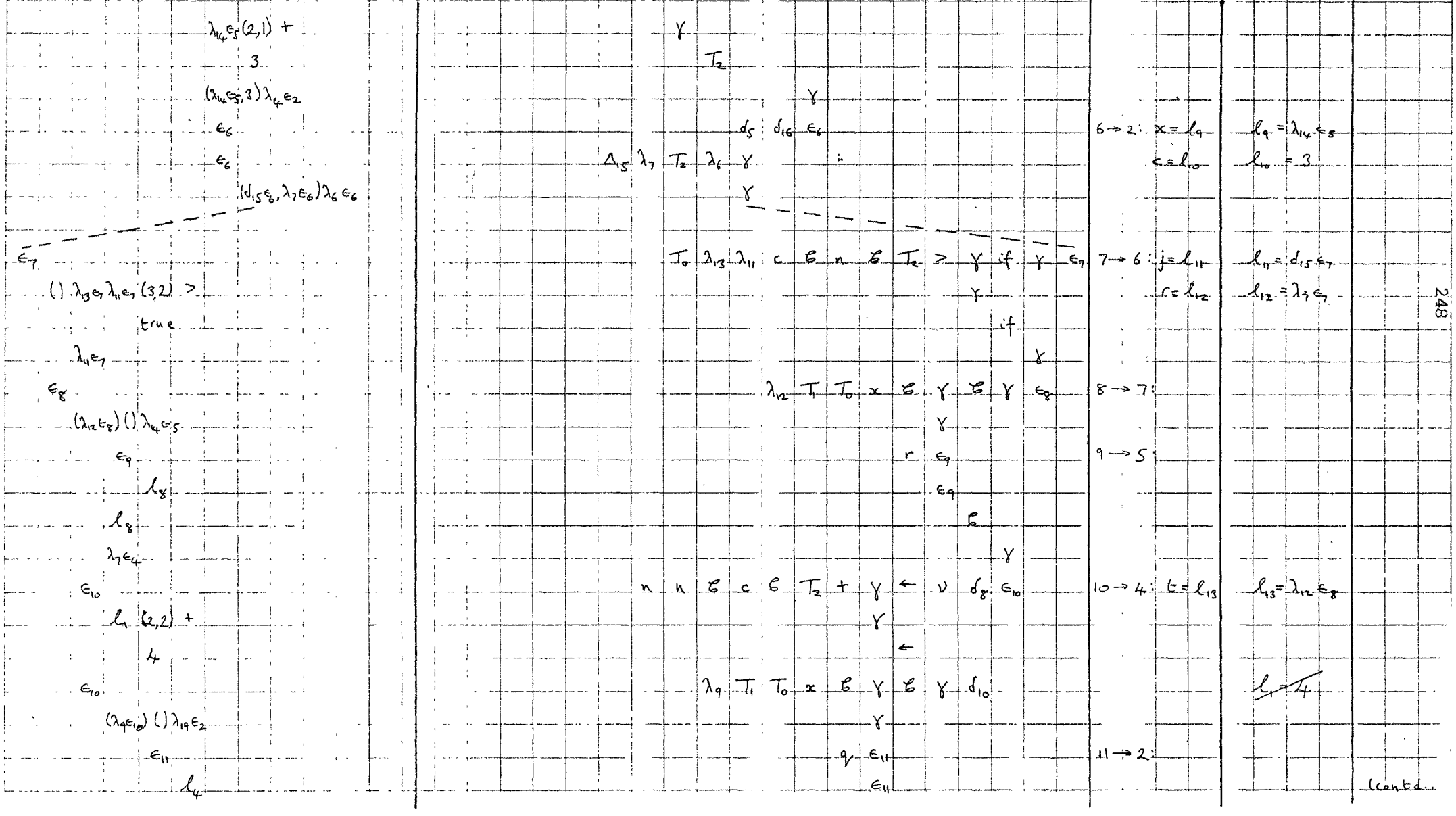
$\delta_{18} \equiv \lambda_{19}\, 2\, T_2\, p\, \mathcal{E}\, \gamma\, \delta_{20}$

$\lambda_{19} \equiv \lambda \pi.\, q$

$\delta_{20} \equiv n.\, \mathcal{E}\, print$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_0	$\Omega\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	$\sigma : p\epsilon$		$\vdash$
$(\Omega)\ \lambda_2\ \epsilon_0$	γ			
ϵ_1	$\Delta_{20}\ \lambda_4\ \lambda_{17}\ T_3\ \lambda_3\ \gamma\ \epsilon_1$	$1 \to 0 :\ n = l_1$	~~$l_1 = \Omega$~~	
$(\delta_{20}\epsilon_1, \lambda_4\epsilon_1, \lambda_{17}\epsilon_1)\ \lambda_3\ \epsilon_1$	γ			
ϵ_2	$n\ 2\ \leftarrow\ \nu\ \delta_{18}\ \epsilon_2$	$2 \to 1 :\ l = l_2$	$l_2 = \delta_{20}\epsilon_2$	
l_1	2	$p = l_3$	$l_3 = \lambda_4\epsilon_2$	
2	$\leftarrow$	$q = l_4$	$l_4 = \lambda_{17}\epsilon_2$	
ϵ_2	$\lambda_{19}\ 2\ T_2\ p\ \mathcal{E}\ \gamma\ \delta_{20}$		~~$l_1 = I$~~	
$(\lambda_{19}\epsilon_2, 2)\ \lambda_4\ \epsilon_2$	γ			
ϵ_3	$\delta_5\ \delta_{16}\ \epsilon_3$	$3 \to 2 :\ x = l_5$	$l_5 = \lambda_{19}\epsilon_2$	
ϵ_3	$\Delta_{15}\ \lambda_7\ T_2\ \lambda_6\ \gamma$	$c = l_6$	$l_6 = 2$	
$(\delta_{15}\epsilon_3, \lambda_7\epsilon_3)\ \lambda_6\ \epsilon_3$	γ			
ϵ_4	$T_0\ \lambda_{13}\ \lambda_{11}\ c\ \mathcal{E}\ n\ \mathcal{E}\ T_2 > \gamma\ \text{if}\ \gamma\ \epsilon_4$	$4 \to 3 :\ j = l_7$	$l_7 = \delta_{15}\epsilon_4$	
$()\ \lambda_{13}\epsilon_4\ \lambda_{11}\epsilon_4\ (2,2) > \gamma$		$r = l_8$	$l_8 = \lambda_7\epsilon_4$	
false	if			
$\lambda_{14}\epsilon_4$	γ			
ϵ_5	$\lambda_{14}\ c\ \mathcal{E}\ 1\ T_2 + \gamma\ T_2\ p\ \mathcal{E}\ \gamma\ \epsilon_5$	$5 \to 4 :$		(contd...)

figure A35 continued



(cont'd...

figure A35 continued (2)

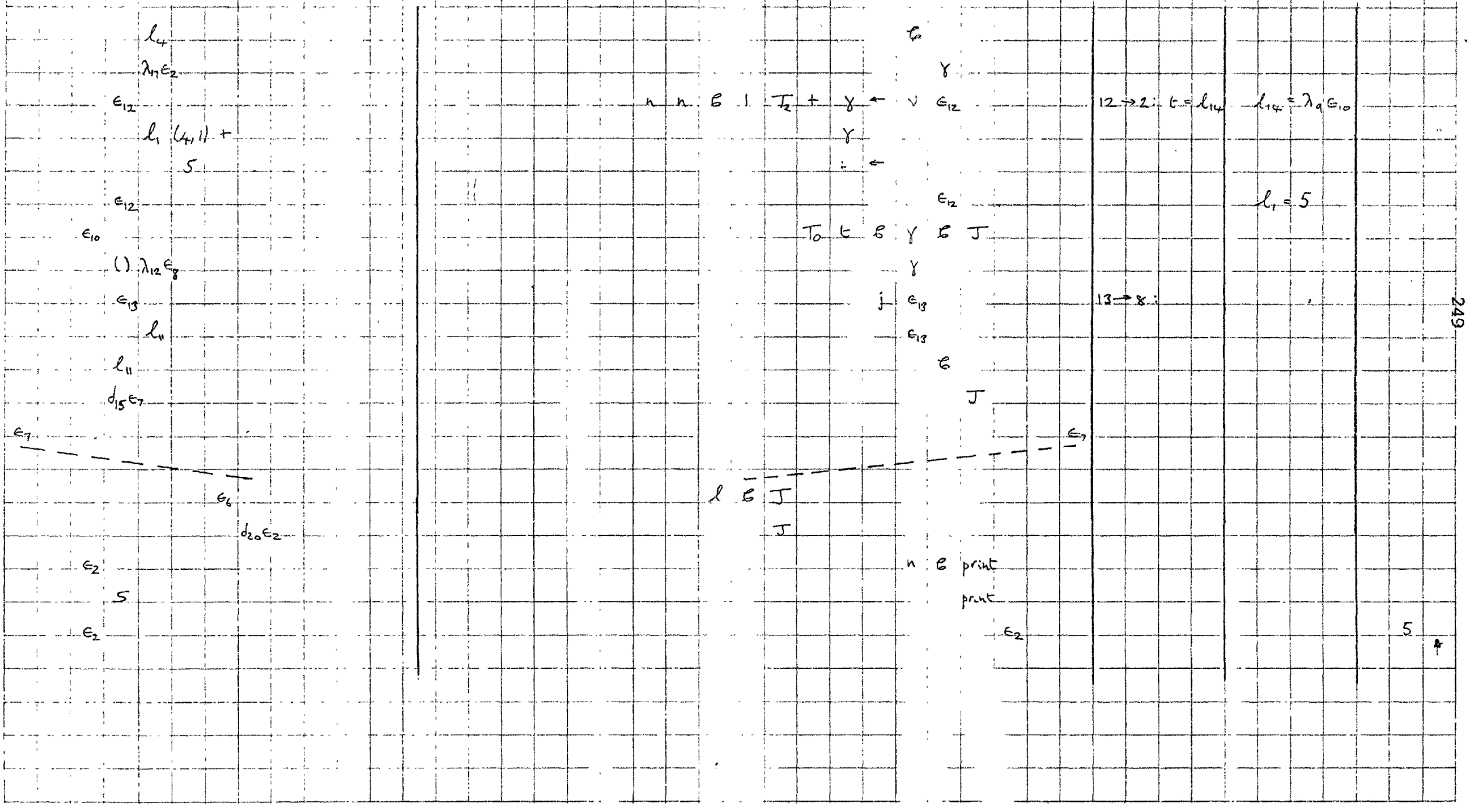


figure A.36

$\delta_1 \equiv \Omega\Omega T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda n.\, n\ \text{read} \leftarrow \nu\, \delta_3$

$\delta_3 \equiv 1\ n\ \mathcal{C}\ T_2\ T_1\ \text{formarray}\ \Omega\Omega T_3\ \lambda_4\ \gamma$

$\lambda_4 \equiv \lambda(a, i, max).\ \Delta_6\ \Delta_{15}\ T_2\ \lambda_5\ \gamma$

$\lambda_5 \equiv \lambda_{rec}(l, m).\ max\ i\ 0 \leftarrow \leftarrow \nu\, \delta_6$

$\delta_6 \equiv T_0\ \lambda_8\ \lambda_7\ i\ \mathcal{C}\ n\ \mathcal{C}\ T_2 = \gamma\ if\ \gamma\ \delta_9$

$\lambda_7 \equiv \lambda\pi.\ m\ \mathcal{C}\ \mathcal{J}$

$\lambda_8 \equiv \lambda\pi.$

$\delta_9 \equiv i\ i\ \mathcal{C}\ 1\ T_2 + \gamma \leftarrow \nu\, \delta_{10}$

$\delta_{10} \equiv a\ i\ \mathcal{C}\ T_1\ or\ read \leftarrow \nu\, \delta_{11}$

$\delta_{11} \equiv T_0\ \lambda_{13}\ \lambda_{12}\ a\ i\ \mathcal{C}\ T_1\ or\ \mathcal{C}\ max\ \mathcal{C}\ T_2 > \gamma\ if\ \gamma\ \delta_{14}$

$\lambda_{12} \equiv \lambda\pi.\ max\ a\ i\ \mathcal{C}\ T_1\ \nu\ \mathcal{C} \leftarrow \nu$

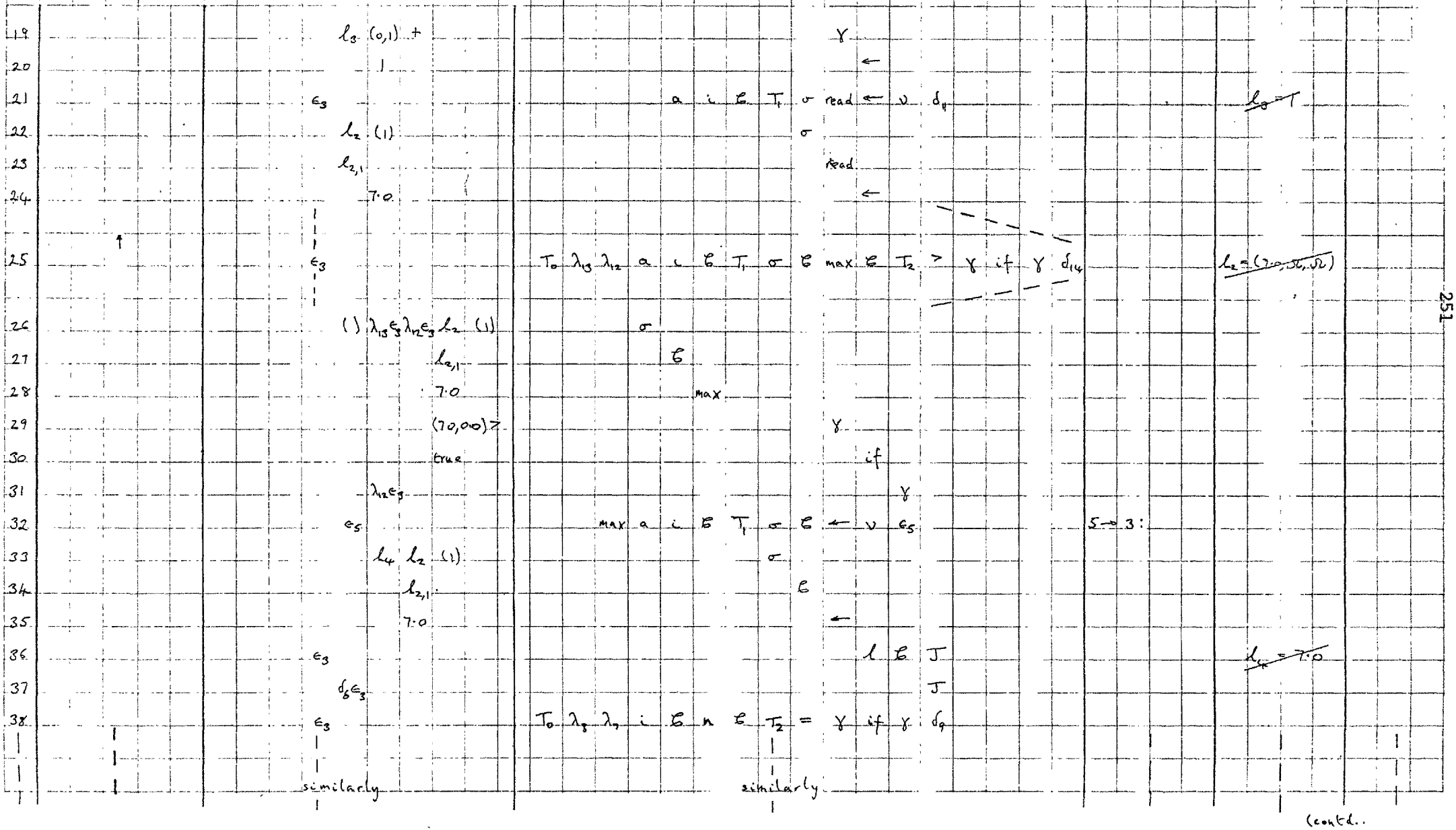
$\lambda_{13} \equiv \lambda\pi.$

$\delta_{14} \equiv l\ \mathcal{C}\ \mathcal{J}$

$\delta_{15} \equiv max\ \mathcal{C}\ print$

LINE NO.	INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
1	3 7 0 4 0 9 0 ⊣ ↑	ϵ_0	$\Omega\ T_1\ \lambda_2\ \gamma\ \epsilon_0$	0 : PE		⊢ ↑
2		$(\Omega)\ \lambda_2 \epsilon_0$	γ			
3		ϵ_1	n read $\leftarrow \nu\ \delta_3\ \epsilon_1$	$1 \to 0 : n = l_1$	~~$l_1 = \Omega$~~	
4	↑	l_1 3	$\leftarrow$			
5		ϵ_1	$1\ n\ \mathcal{C}\ T_2\ T_1$ formarray $\Omega\ \Omega\ T_3\ \lambda_4\ \gamma$		$l_1 = 3$	
6		$((1,3))$	formarray			
7		$(\Omega_1, \Omega_2, \Omega)$	Ω			
8		$((\Omega_1, \Omega_2, \Omega), \Omega_3, \Omega_4)\ \lambda_4 \epsilon_1$	γ			
9		ϵ_2	$\Delta_6\ \Delta_{15}\ T_2\ \lambda_5\ \gamma\ \epsilon_2$	$2 \to 1 : a = l_2$	~~$l_2 = (\Omega_1, \Omega_2, \Omega_3)$~~	
10		$\delta_6 \epsilon_2\ \delta_{15} a_2$	T_2	$i = l_3$	~~$l_3 = \Omega_4$~~	
		$(\delta_6 \epsilon_2, \delta_{15} \epsilon_2)\ \lambda_5 \epsilon_2$	γ	$max = l_4$	~~$l_4 = \Omega_5$~~	
11		ϵ_3	$max\ i\ 0 \leftarrow \leftarrow \nu\ \delta_6\ \epsilon_3$	$3 \to 2 : l = l_5$	$l_5 = \delta_6 \epsilon_3$	
12		$l_4\ l_3\ 0$	$\leftarrow$	$m = l_6$	$l_6 = \delta_{15} \epsilon_3$	
13		ϵ_3	$T_0\ \lambda_8\ \lambda_7\ i\ \mathcal{C}\ n\ \mathcal{C}\ T_2 = \gamma\ if\ \gamma\ \delta_9$		~~$l_3 = 0$~~	
14		$()\ \lambda_8 \epsilon_3\ \lambda_7 \epsilon_3\ (0,3) =$	γ		~~$l_4 = 0 : 0$~~	
15		false	if			
16		$\lambda_8 \epsilon_3$	γ			
17		ϵ_4	ϵ_4	$4 \to 3 :$		
18		ϵ_3	$i\ i\ \mathcal{C}\ 1\ T_2 + \gamma \leftarrow \nu\ \delta_{10}$			(cont'd...

figure A36 continued



(contd..

figure A36 continued (3)

```
?
                                    ϵ3
                                    () λ8ϵ3.λ7ϵ3.(3,3) =
                                                  true
                                          λ7ϵ3
                                    ϵ10
                                          δ15ϵ3
                                    ϵ3
                                    9.0
                                    ϵ3
```

```
T0 λ8 λ7 . ϵ n ϵ T2 = γ if γ δ9
                      γ
                        if
                      . γ
                    n ϵ J ϵ10
                        J
                        max ϵ print
                            print
                                  ϵ3
```

```
10→3:
```

```
l2=(7.0,4.0,9.0)
l3=3
l4=9.0
```

```
9.0
```

figure A37a

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec}\, p.\, \lambda_6 a\, i.\, \mathcal{C}\, T_1\, \sigma\, \mathcal{C}\, T_2\, p\, \mathcal{C}\, \gamma$

$\lambda_3 \equiv \lambda(i,r).\, T_0\, \mathcal{C}\, \gamma\, 2 \leftarrow v\, \delta_4$

$\delta_4 \equiv a\, l\, T_1\, \sigma\, 2 \leftarrow v\, \delta_5$

$\delta_5 \equiv r\, \mathcal{C}\, \text{print}$

$\lambda_6 \equiv \lambda \pi.\, i$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_1	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_1$	$1 \to 0$: $i = l_1$	~~$l_1 = 1$~~	⊢
$(\lambda_3 \epsilon_1)\ \lambda_2 \epsilon_1$	γ	$a = l_2$	~~$l_2 = (1,3)$~~	↑
ϵ_2	$\lambda_6\ a\ i\ \mathcal{C}\ T_1\ \sigma\ \mathcal{C}\ T_2\ p\ \mathcal{C}\ \gamma\ \epsilon_2$	$2 \to 1$: $p = l_3$	$l_3 = \lambda_3 \epsilon_2$	
$\lambda_6 \epsilon_2\ l_2\ (1)$	σ			
$l_{2,1}$	$\mathcal{C}$			
1	T_2			
$(\lambda_6 \epsilon_2, 1)\ \lambda_3 \epsilon_2$	γ			
ϵ_3	$T_0\ i\ \mathcal{C}\ \gamma\ 2 \leftarrow v\ \delta_4\ \epsilon_3$	$3 \to 2$: $i = l_4$	$l_4 = \lambda_6 \epsilon_2$	
$()\ \lambda_6 \epsilon_2$	γ	$r = l_5$	$l_5 = 1$	
ϵ_4	$i\ \epsilon_4$	$4 \to 2$		
l_1	ϵ_4			
$l_1\ 1$	$\leftarrow$			
ϵ_3	$a\ l\ T_1\ \sigma\ 2 \leftarrow v\ \delta_5$		$l_1 = 2$	
$l_2\ (1)$	σ			
$l_{2,1}\ 2$	$\leftarrow$			
ϵ_3	$r\ \mathcal{C}$ print		$l_2 = (2,3)$	
1	print			
ϵ_3	ϵ_3			

figure A376

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec}\, p.\, \lambda_6\, a\, i\, \varepsilon\, T_1\, \sigma\, T_2\, p\, \varepsilon\, \gamma$

$\lambda_3 \equiv \lambda(i,r).\, T_0\, i\, \varepsilon\, \gamma\, 2 \leftarrow v\, \delta_4$

$\delta_4 \equiv a\, 1\, T_1\, \sigma\, 2 \leftarrow v\, \delta_5$

$\delta_5 \equiv r\, \varepsilon\, \varepsilon$ print

$\lambda_6 \equiv \lambda \pi.\, \pi$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_1	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_1$	$1 \to 0$: $i = l_1$	~~$l_1 = 1$~~	
$(\lambda_3, \epsilon_1)\ \lambda_2\ \epsilon_1$	: γ	$a = l_2$	~~$l_2 = (1,3)$~~	
ϵ_2	$\lambda_6\ a\ i\ \varepsilon\ T_1\ \sigma\ T_2\ p\ \varepsilon\ \gamma\ \epsilon_2$	$2 \to 1$: $p = l_3$	$l_3 = \lambda_3 \epsilon_2$	
$\lambda_6 \epsilon_2\ l_2\ (1)$	σ			
$l_{2,1}$	T_2			
$(\lambda_6 \epsilon_2, l_{2,1})\ \lambda_3\ \epsilon_2$	γ			
ϵ_3	$T_0\ i\ \varepsilon\ \gamma\ 2 \leftarrow v\ \delta_4\ \epsilon_3$	$3 \to 2$: $i = l_4$	$l_4 = \lambda_6 \epsilon_2$	
$(.)\ \lambda_6 \epsilon_2$	γ	$r = l_5$	$l_5 = l_{2,1}$	
ϵ_4	$i\ \epsilon_4$	$4 \to 2$:		
l_1	ϵ_4			
$l_1\ 2$	$\leftarrow$			
ϵ_3	$a\ 1\ T_1\ \sigma\ 2 \leftarrow v\ \delta_5$		$l_1 = 2$	
$l_2\ (1)$	σ			
$l_{2,1}\ 2$	$\leftarrow$			
ϵ_3	$r\ \varepsilon\ \varepsilon$ print		$l_2 = (2,3)$	
l_5	ε			
$l_{2,1}$	ε			
2	print			2
ϵ_3	ϵ_3			

figure A37c

$\delta_1 \equiv \lambda_3 T_1 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda rec\, p.\ \lambda_6 \lambda_7 T_2\, p\, \mathcal{E}\gamma$

$\lambda_3 \equiv \lambda(i,r).\ T_0\, i\, \mathcal{E}\gamma\, 2 \leftarrow \nu\delta_4$

$\delta_4 \equiv a\ i\ T_1\ \sigma\ 2 \leftarrow \nu\delta_5$

$\delta_5 \equiv T_0\ r\ \mathcal{E}\gamma\ \mathcal{E} print$

$\lambda_6 \equiv \lambda\pi.\ i$

$\lambda_7 \equiv \lambda\pi\ a\ i\ \mathcal{E}\ T_1\ \sigma$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_1	$\lambda_3\ T_1\ \lambda_2\ \gamma\ \epsilon_1$	$1 \to 0$: $i = l_1$	~~$l_1 = 1$~~	
$(\lambda_3 \epsilon_1)\ \lambda_2 \epsilon_1$	$:\ \gamma$	$a = l_2$	~~$l_2 = (1,3)$~~	
ϵ_2	$\lambda_6\ \lambda_7\ T_2\ p\ \mathcal{E}\ \gamma\ \epsilon_2$	$2 \to 1$: $p = l_3$	$l_3 = \lambda_3 \epsilon_2$	
$(\lambda_6 \epsilon_2, \lambda_7 \epsilon_2)\ \lambda_3 \epsilon_2$	γ			
ϵ_3	$T_0\ i\ \mathcal{E}\ \gamma\ 2 \leftarrow \nu\ \delta_4\ \epsilon_3$	$3 \to 2$: $i = l_4$	$l_4 = \lambda_6 \epsilon_2$	
$()\ \lambda_6 \epsilon_2$	γ	$r = l_5$	$l_5 = \lambda_7 \epsilon_2$	
ϵ_4	$i\ \epsilon_4$	$4 \to 2$:		
l_1	ϵ_4			
$l_1\ 2$	$\leftarrow$			
ϵ_3	$a\ i\ T_1\ \sigma\ 2 \leftarrow \nu\ \delta_5$		$l_1 = 2$	
$l_2\ (1)$	σ			
$l_{2,1}\ 2$	$\leftarrow$			
ϵ_3	$T_0\ r\ \mathcal{E}\ \gamma\ \mathcal{E}$ print		$l_2 = (2,3)$	
$()\ \lambda_7 \epsilon_2$	γ			
ϵ_5	$a\ i\ \mathcal{E}\ T_1\ \sigma\ \epsilon_5$	$5 \to 2$:		
$l_2\ (2)$	σ			
$l_{2,2}$	ϵ_5			
$l_{2,2}$	$\mathcal{E}$			
3	print			
ϵ_3	ϵ_3			

figure A38

$S T_1 \lambda_1 \gamma$

$\lambda_1 \equiv \lambda x.\ \lambda_3 \lambda_2\ x\ \mathcal{E}0\ T_2 > \gamma$ if

$\lambda_2 \equiv \lambda \pi.\ x\ \mathcal{E} x\ \mathcal{E} T_2 \times \gamma$

$\lambda_3 \equiv \lambda \pi.\ x\ \mathcal{E} 1\ T_2 - \gamma$

LINE NO.	STACK	CONTROL	ENVIRONMENT	MEMORY
1	ϵ_0	$S\ T_1\ \lambda_1\ \gamma\ \epsilon_0$	$0 : PE$	
2	$(S)\ \lambda_1 \epsilon_0$	γ		
3	ϵ_1	$\lambda_3\ \lambda_2\ x\ \mathcal{E}\ 0\ T_2 > \gamma$ if ϵ_1	$1 \to 0 : x = l_1$	$l_1 = S$
4	$\lambda_3 \epsilon_1, \lambda_2 a_1\ (S,0) >$	γ		
5	true	if		
6	ϵ_1	$x\ \mathcal{E}\ x\ \mathcal{E}\ T_2 \times \gamma$ ϵ_1		
7	$(S,S)\ \times$	γ		
8	25	ϵ_1		
9	25			

figure A39

$\delta_1 \equiv \Omega \lambda_3 T_2 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec}\ (i,p).\ i\ read \leftarrow v.\ \delta_4$

$\lambda_3 \equiv \lambda_{rec}\ i.\ T_0\ i\ \mathcal{E}\gamma\ T_0\ i\ \mathcal{E}\gamma\ \mathcal{E}\ 1\ T_2 + \gamma \leftarrow v$

$\delta_4 \equiv \lambda_5 T_1\ p\ \mathcal{E}\gamma\ print$

$\lambda_5 \equiv \lambda_{rec}\ \pi.\ i$

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
3 ⊣	ϵ_0	$\Omega\ \lambda_3\ T_2\ \lambda_2\ \gamma\ \epsilon_0$	$0 : PE$		⊢
↑	$(\Omega, \lambda_3 \epsilon_0)\ \lambda_2 \epsilon_0$	γ			4
	ϵ_1	$i\ read \leftarrow v\ \delta_4\ \epsilon_1$	$1 \to 0 : i = l_1$	~~$l_1 = \Omega$~~	
	$l_1\ 3$	$\leftarrow$	$p = l_2$	$l_2 = \lambda_3 \epsilon_1$	
↑	ϵ_1	$\lambda_5\ T_1\ p\ \mathcal{E}\ \gamma\ print$		$l_1 = 3$	(contd..

figure A39 continued

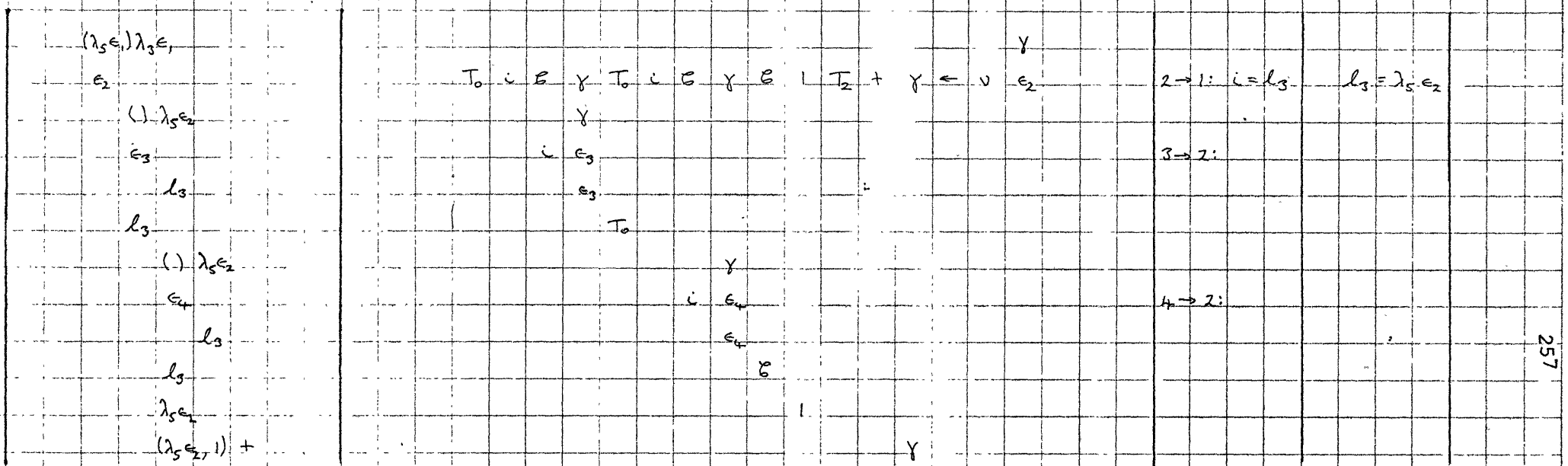


figure A40

$d_1 \equiv S\lambda_3\lambda_6 T_3 \lambda_2 \gamma$

$\lambda_2 \equiv \lambda_{rec}(x,p,q).x$ read $\leftarrow v\, d_{10}$

$\lambda_3 \equiv \lambda a.\lambda_5\lambda_4$ read if

$\lambda_4 \equiv \lambda\pi.a T_1 p \gamma$

$\lambda_5 \equiv \lambda\pi.a T_1 q \gamma$

$\lambda_6 \equiv \lambda_9 \lambda_7$ read if

$\lambda_7 \equiv \lambda\pi.a T_1 p \gamma d_8$

$d_8 \equiv a\, a\, E T_1 \sim \gamma \leftarrow v$

$\lambda_9 \equiv \lambda\pi.$

$d_{10} \equiv x T_1 p \gamma d_{11}$

$\delta_{11} \equiv x E$ print

INPUT	STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
true false true false false ⊢ ↑	ε_0	$S\ \lambda_3\ \lambda_6\ T_3\ \lambda_2\ \gamma\ \varepsilon_0$	0 : PE		⊢ ↑
	l_1	λ_3		~~l_1 = ω~~	
	$(l_1, \lambda_3\varepsilon_0, \lambda_6\varepsilon_0)\ \lambda_2\varepsilon_0$	γ			
	ε_1	x read $\leftarrow$ v d_{10} ε_1	1 → 0: x = l_1		(contd..

figure A4.0 continued

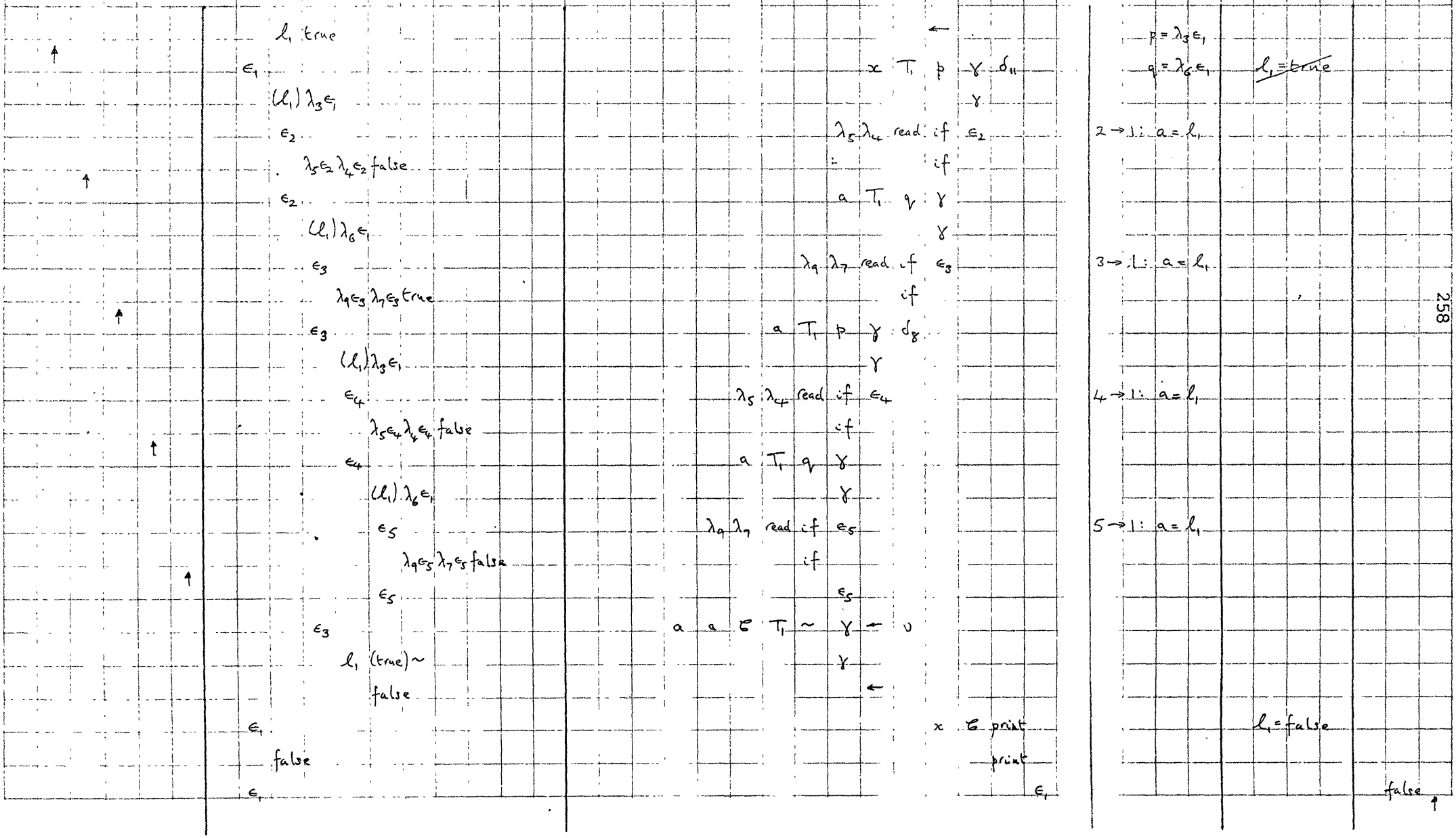


figure A.4.1

$\delta_1 \equiv d\pi.\ T_0\ \lambda_2\ \gamma$

$\lambda_2 \equiv \lambda\pi.\ S\ S\ \Omega\ \text{"bob"}\ \Omega\ T_3 \leftarrow \text{"jim"}\ \dot{S}\ \Omega\ \text{"sam"}\ \Omega\ T_3 \leftarrow T_3 \leftarrow \lambda_4\ T_2\ \delta_3$

$\delta_3 \equiv \delta(\text{tree}, \text{postwalk}).\ \text{tree}\ T_1\ \text{postwalk}\ \gamma$

$\lambda_4 \equiv \lambda\text{root}.\ \lambda_8\ \lambda_5\ \text{root}\ \Omega\ T_2 \neq \gamma\ \text{if}$

$\lambda_5 \equiv \lambda\pi.\ \text{root}\ 1\ T_1\ \sigma\ \mathfrak{C}\ T_1\ \text{postwalk}\ \gamma\ \delta_6$

$\delta_6 \equiv d\pi.\ \text{root}\ 2\ T_1\ \sigma\ \mathfrak{C}\ \text{pcount}\ \delta_7$

$\delta_7 \equiv d\pi.\ \text{root}\ 3\ T_1\ \sigma\ \mathfrak{C}\ T_1\ \text{postwalk}\ \gamma$

$\lambda_8 \equiv \lambda\pi.$

STACK	CONTROL	ENVIRONMENT	MEMORY	OUTPUT
ϵ_0	$T_0\ \lambda_2\ \gamma\ \epsilon_0$	0 . PE		⊢ ↑
$()\ \lambda_2 \epsilon_0$	γ			
ϵ_1	$S\ S\ \Omega\ \text{"bob"}\ \Omega\ T_3 \leftarrow \text{"jim"}\ \dot{S}\ \Omega\ \text{"sam"}\ \Omega\ T_3 \leftarrow T_3 \leftarrow \lambda_4\ T_2\ \delta_3\ \epsilon_1$	1 → 0 :		
l_1	S		~~$l_1 = \Omega_0$~~	
$l_2\ (\Omega, \text{bob}, \Omega_0)$	$\leftarrow$		~~$l_2 = \Omega_0$~~	
$l_2\ \text{jim}$	S		$l_2 = (\Omega, \text{bob}, \Omega_0)$	
$l_3\ (\Omega, \text{sam}, \Omega_0)$	$\leftarrow$		~~$l_3 = \Omega_0$~~	
(l_2, jim, l_3)	$\leftarrow$		$l_3 = (\Omega, \text{sam}, \Omega_0)$	
l_1	λ_4		$l_1 = (l_2, \text{jim}, l_3)$	
$(l_1, \lambda_4 \epsilon_1)$	δ_3			
ϵ_1	tree T_1 postwalk γ	(1) tree = l_1		
$(l_1)\ \lambda_4 \epsilon_1$	γ	(1) postwalk = $\lambda_4 \epsilon_1$		
ϵ_2	$\lambda_8\ \lambda_5\ \text{root}\ \Omega\ T_2 \neq \gamma\ \text{if}\ \epsilon_2$	2 → 1 : root = l_1		
$\lambda_8 \epsilon_2\ \lambda_5 \epsilon_2\ (l_1, \Omega_0) \neq$ true	γ			
ϵ_2	if			
$l_1\ (1)$	root $1\ T_1\ \sigma\ \mathfrak{C}\ T_1$ postwalk $\gamma\ \delta_6$			
$l_{1,1}$	σ			
$(l_2)\ \lambda_4 \epsilon_1$	$\mathfrak{C}$			
ϵ_3	γ			
	$\lambda_8\ \lambda_5\ \text{root}\ \Omega\ T_2 \neq \gamma\ \text{if}\ \epsilon_3$	3 → 1 : root = l_2		(cont'd ...

figure A4.1 continued

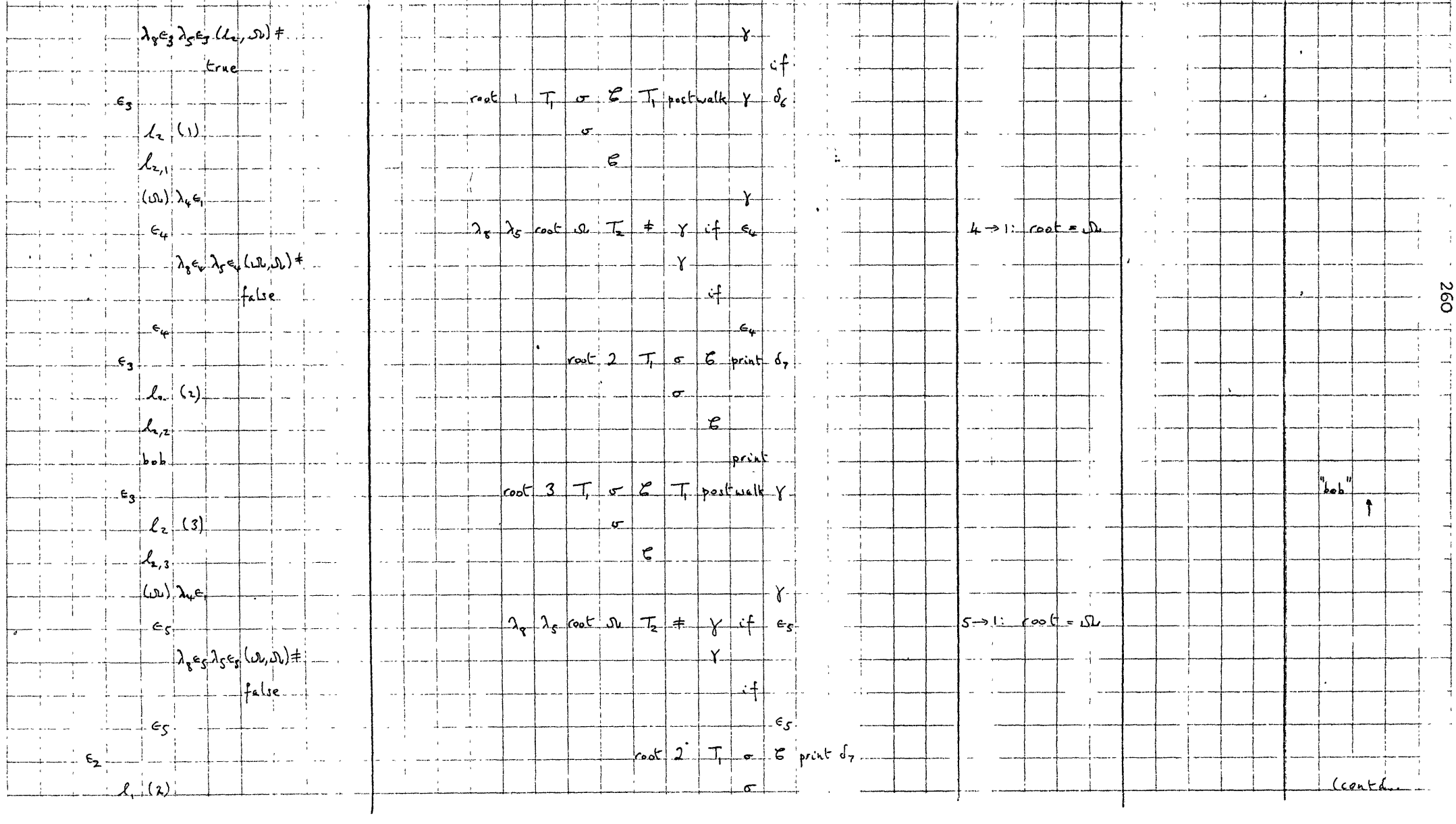


figure A41 continued (2)

$\ell_{1,2}$

jim

ϵ_2

ℓ_1 (3)

$\ell_{1,3}$

$(\ell_3)\ \lambda_4 \epsilon_1$

ϵ_6

$\lambda_8 \epsilon_6 \lambda_5 \epsilon_8 (\ell_3, \omega_1) \neq$

true

ϵ_6

ℓ_3 (1)

$\ell_{3,1}$

$(\omega_1)\ \lambda_4 \epsilon_1$

ϵ_7

$\lambda_8 \epsilon_7 \lambda_5 \epsilon_7 (\omega_1, \omega_1) \neq$

false

ϵ_7

ϵ_6

ℓ_3 (2)

$\ell_{3,2}$

sam

ϵ_6

ℓ_3 (3)

$\ell_{3,3}$

$\mathcal{E}$

print

root 3 T_1 σ $\mathcal{E}$ T_1 postwalk γ

σ

$\mathcal{E}$

:

γ

λ_8 λ_5 root ω_2 T_2 $\neq$ γ if ϵ_6

γ

if

root 1 T_1 σ $\mathcal{E}$ T_1 postwalk γ δ_6

σ

$\mathcal{E}$

γ

λ_8 λ_5 root ω_2 T_2 $\neq$ γ if ϵ_7

γ

if

ϵ_7

root 2 T_1 σ $\mathcal{E}$ print δ_7

σ

$\mathcal{E}$

print

root 3 T_1 σ $\mathcal{E}$ T_1 postwalk γ

σ

$\mathcal{E}$

6→1: root = ℓ_3

7→1: root = ω_2

"jim"

"sam"

(cont-d..

figure A.4.1 continued (3)

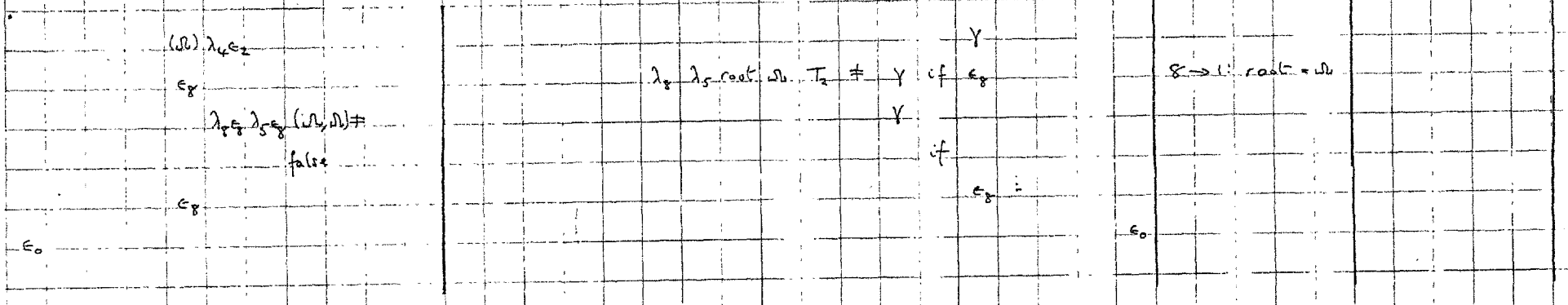


figure A.4.2

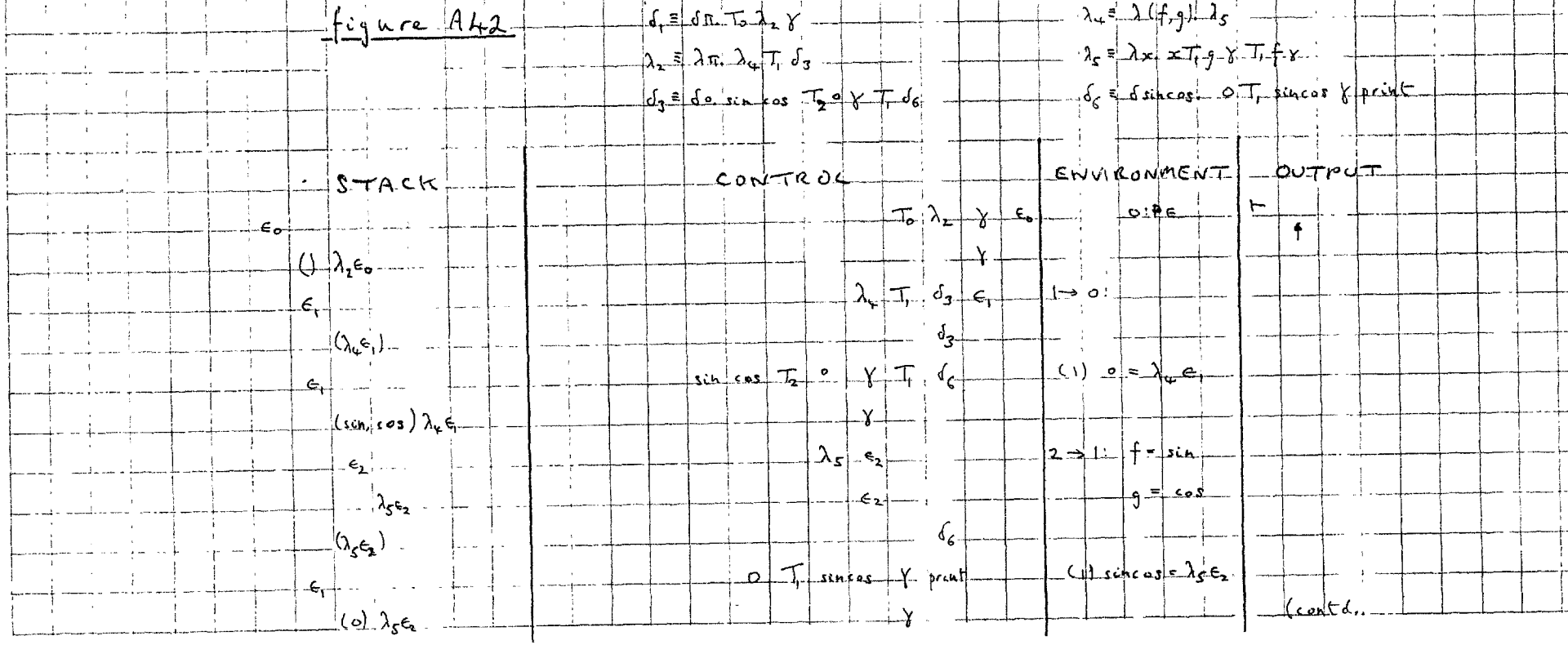


figure A42 continued

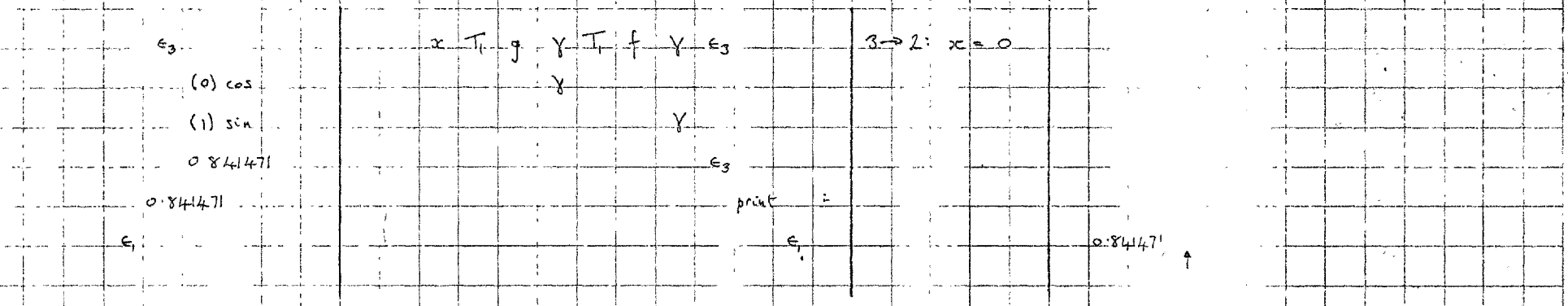


figure A43

$\delta_1 \equiv \delta\pi.\ \begin{Bmatrix}\delta_2\\ \delta_3\end{Bmatrix}\delta_4$

$\delta_2 \equiv \delta\pi.\ \left\{\begin{array}{l} x\ i\ \mathcal{E}\ T_1\ \sigma \\ \left\{\begin{array}{l} i \\ i\ \mathcal{E}\ 1\ T_2 + \gamma \end{array}\right\} \leftarrow \mathcal{E} \end{array}\right\} \leftarrow \nu$

$\delta_3 \equiv \delta\pi.\ \left\{\begin{array}{l} x\ i\ \mathcal{E}\ T_1\ \sigma \\ \left\{\begin{array}{l} i \\ i\ \mathcal{E}\ 1\ T_2 + \gamma \end{array}\right\} \leftarrow \mathcal{E} \end{array}\right\} \leftarrow \nu$

$\delta_4 \equiv \delta\pi.\ i\ \mathcal{E}\ \text{print}\ x\,1\ T_1\ \sigma\ \mathcal{E}\ \text{print}\ x\,2\ T_1\ \sigma\ \mathcal{E}\ \text{print}\ x\,3\ T_1\ \sigma\ \mathcal{E}\ \text{print}$

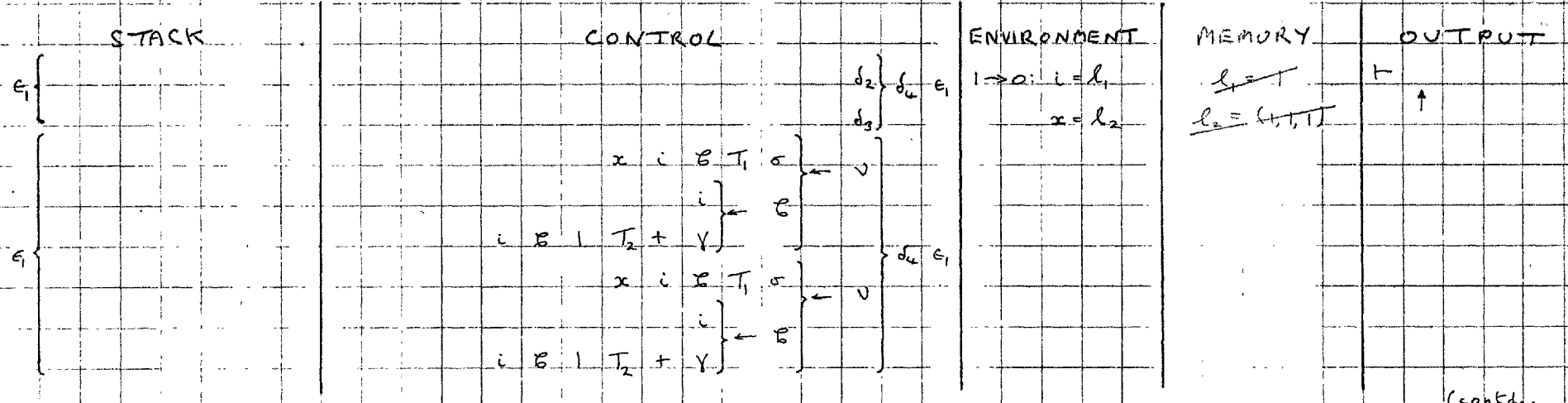
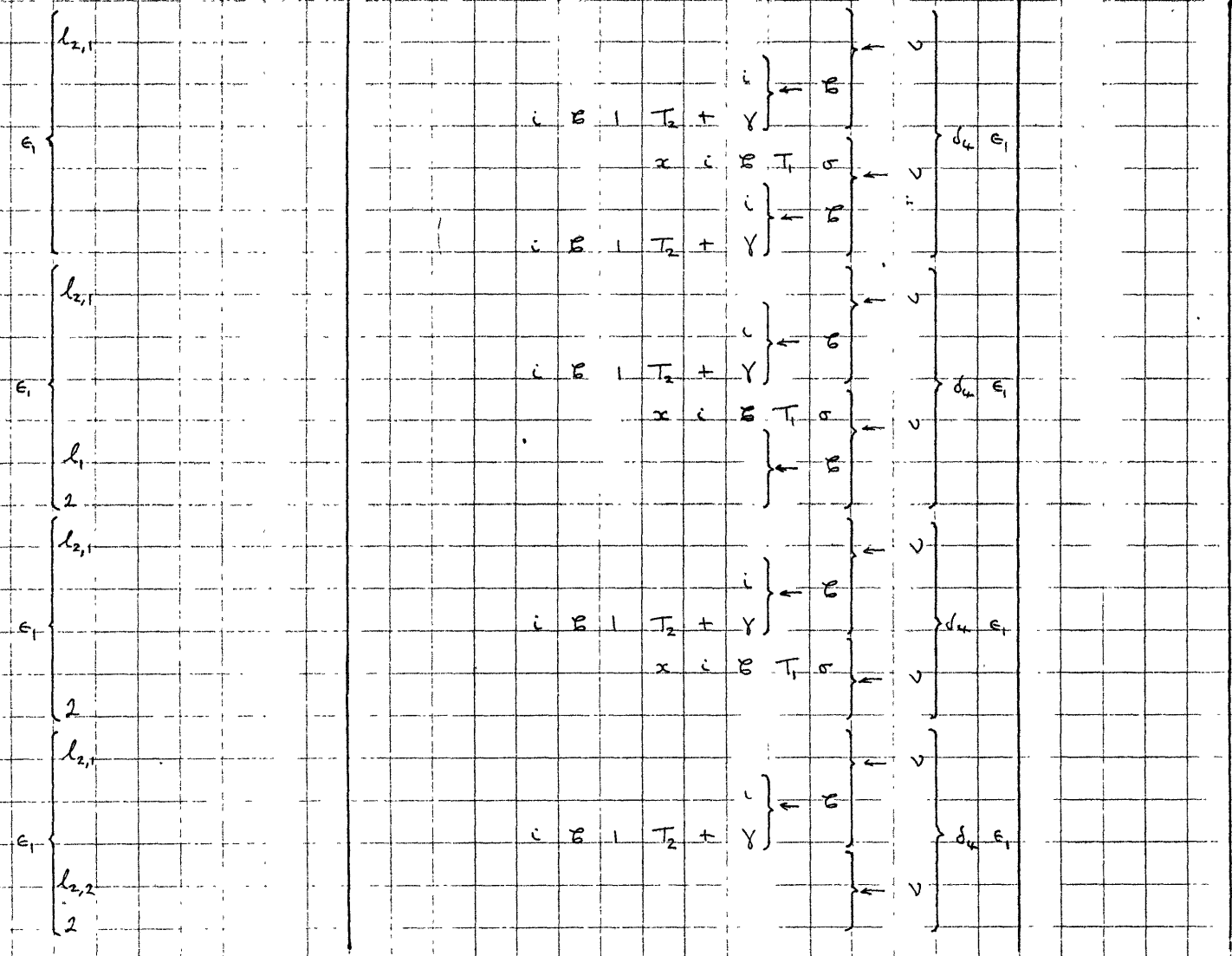


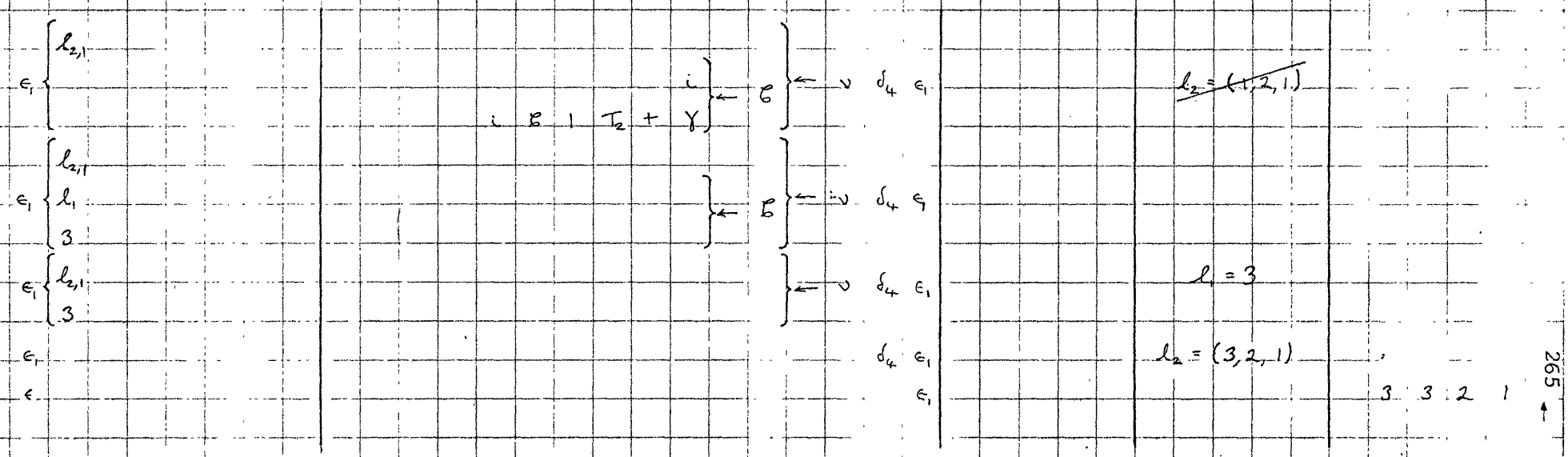
figure A43 continued



~~$l_1 = 2$~~

(cont'd...

figure A43 continued (2)



APPENDIX B

Figures B1 – B14

figure B1 (line 1)

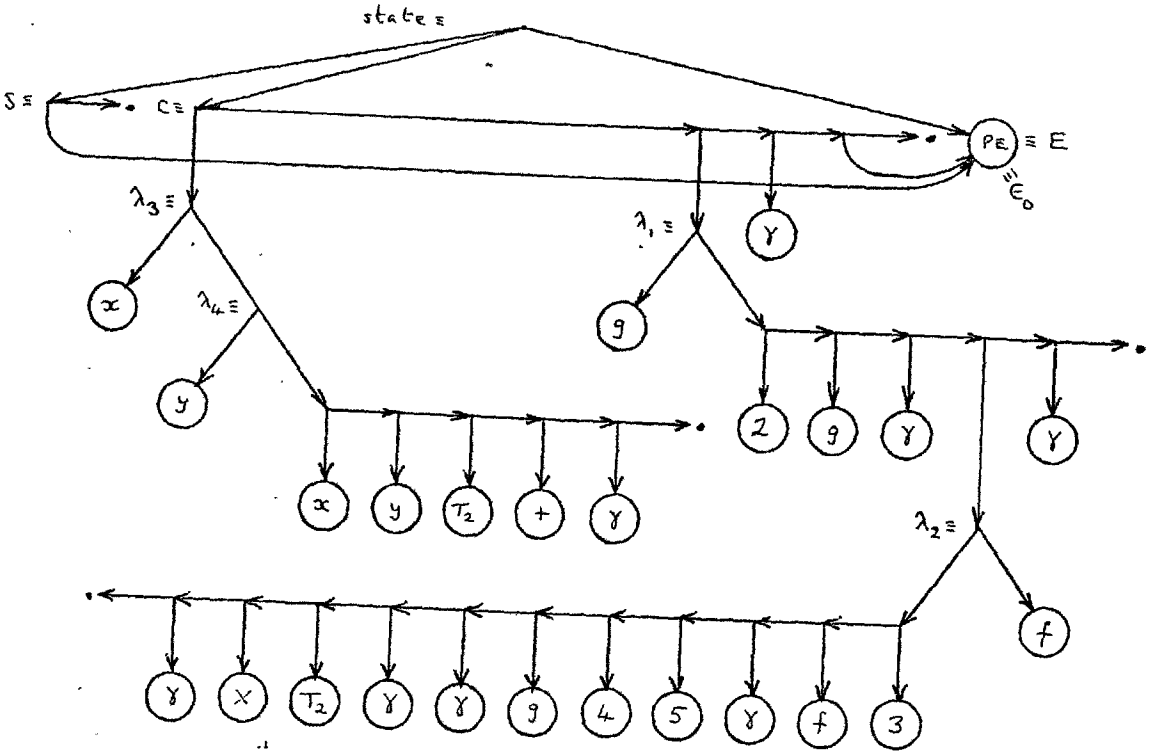


figure B2 (line 2)

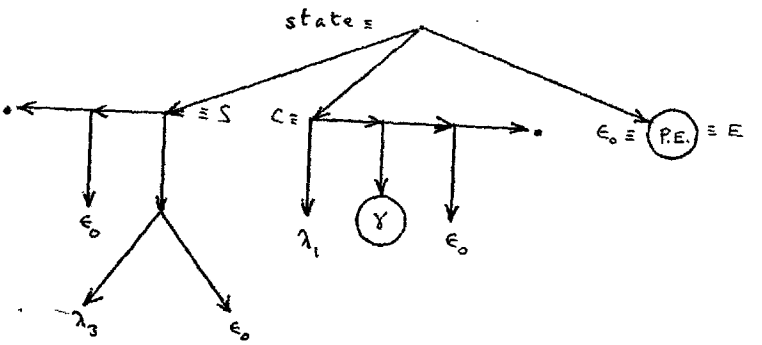


figure B3 (line 3)

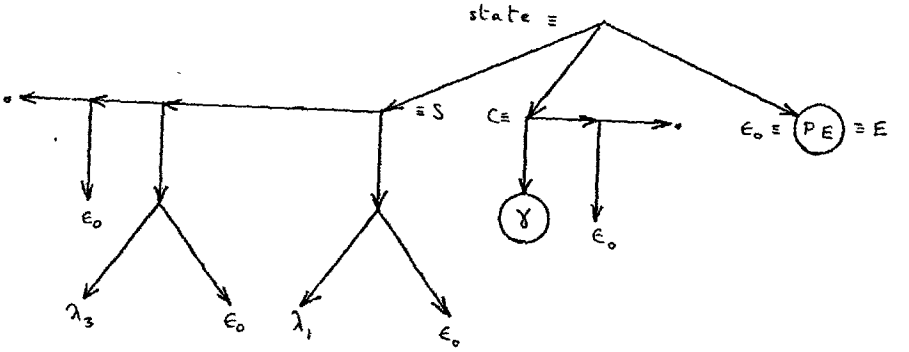


figure B4 (line 4)

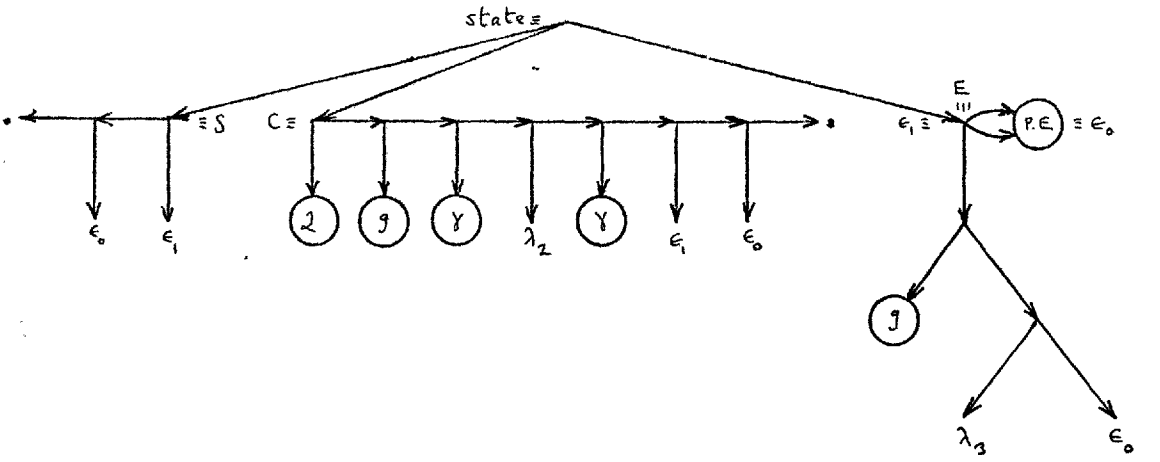


figure B5 (line 6)

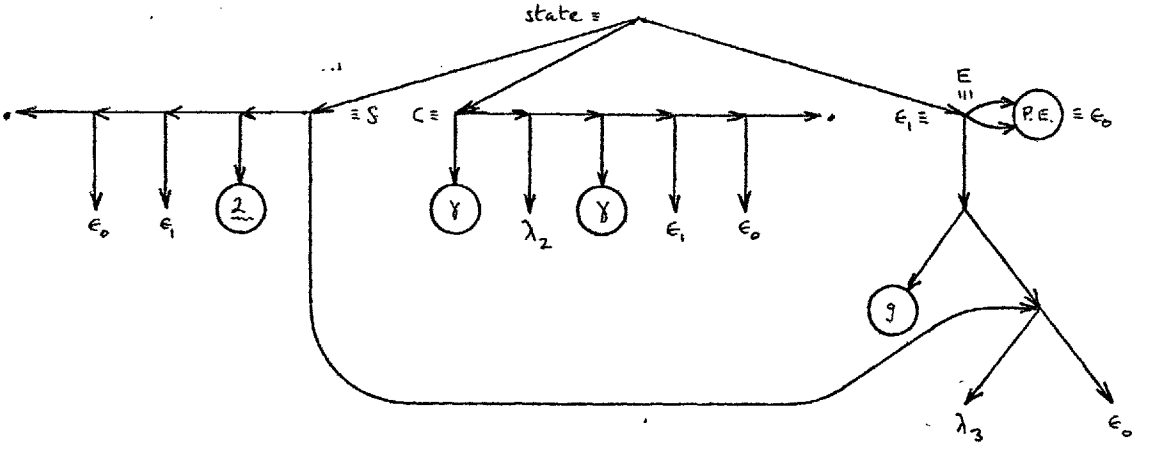


figure B6 (line 7)

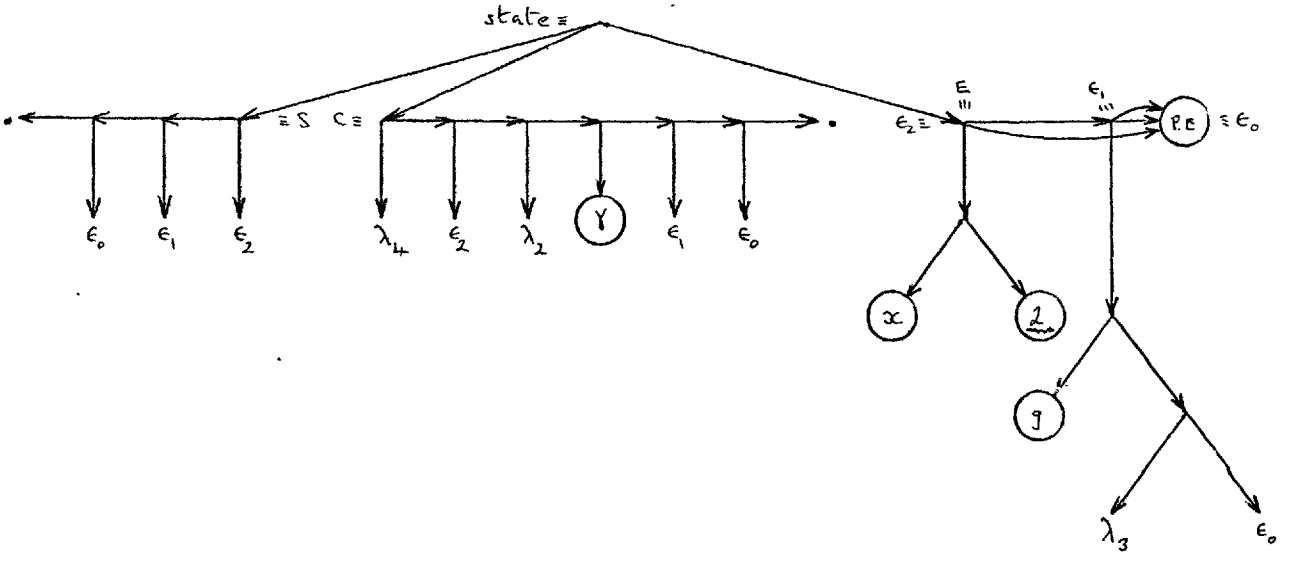


figure B7 (line 8)

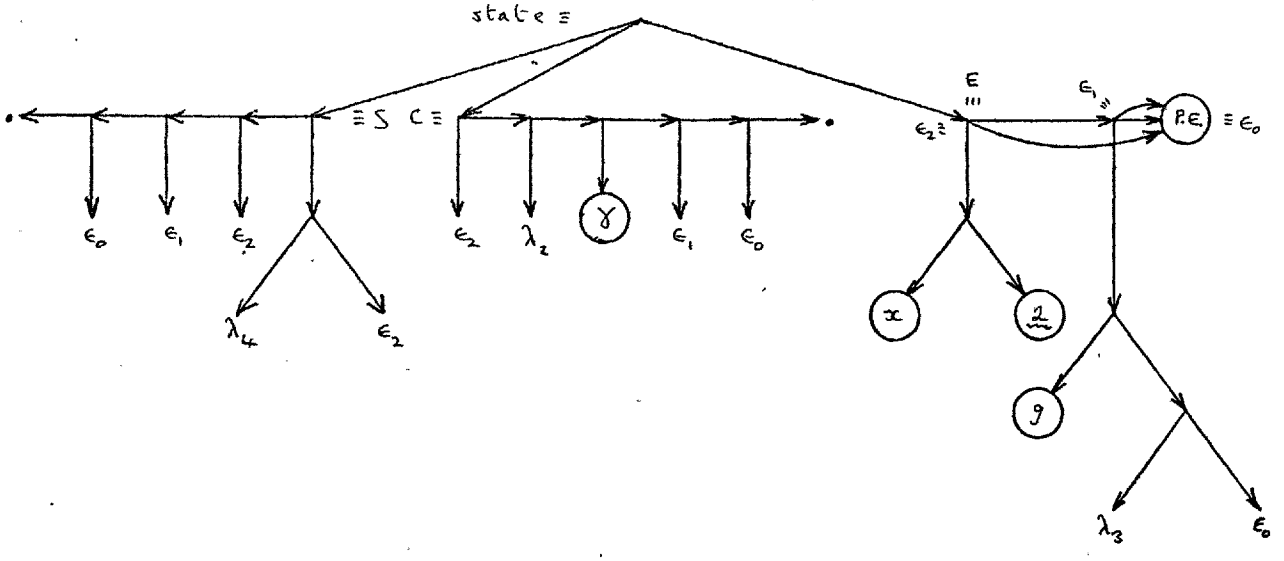


figure B8 (line 9)

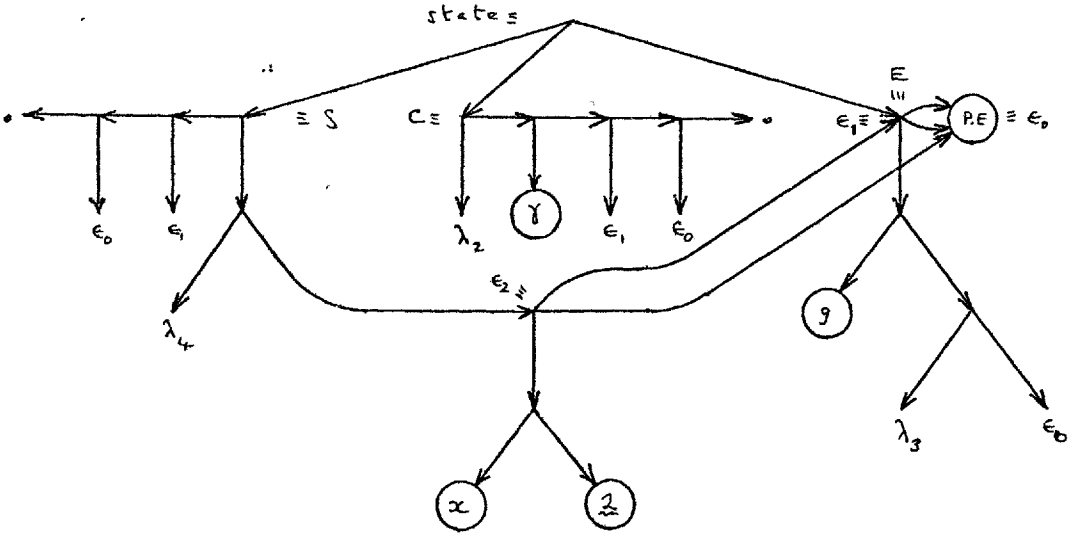


figure B9 (line 10)

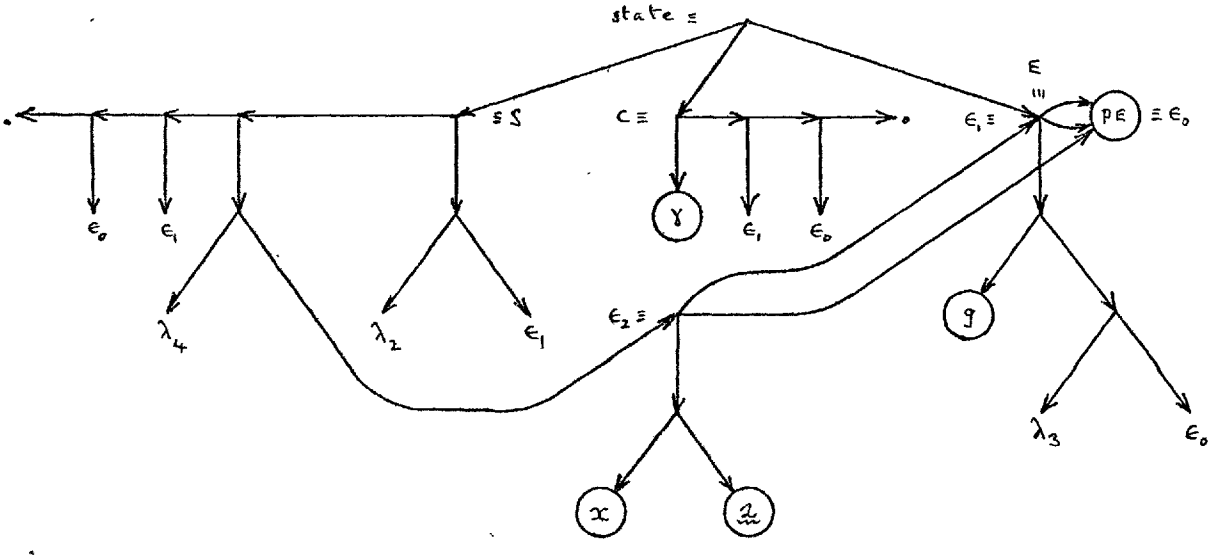


figure B10 (line 11)

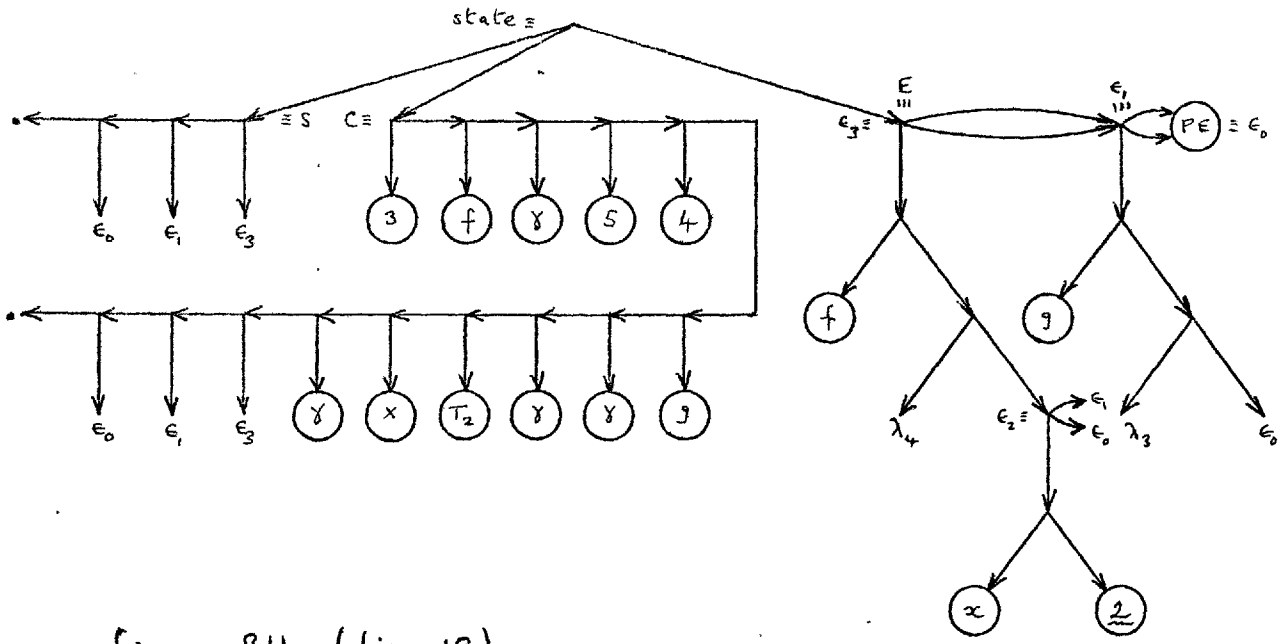


figure B11 (line 13)

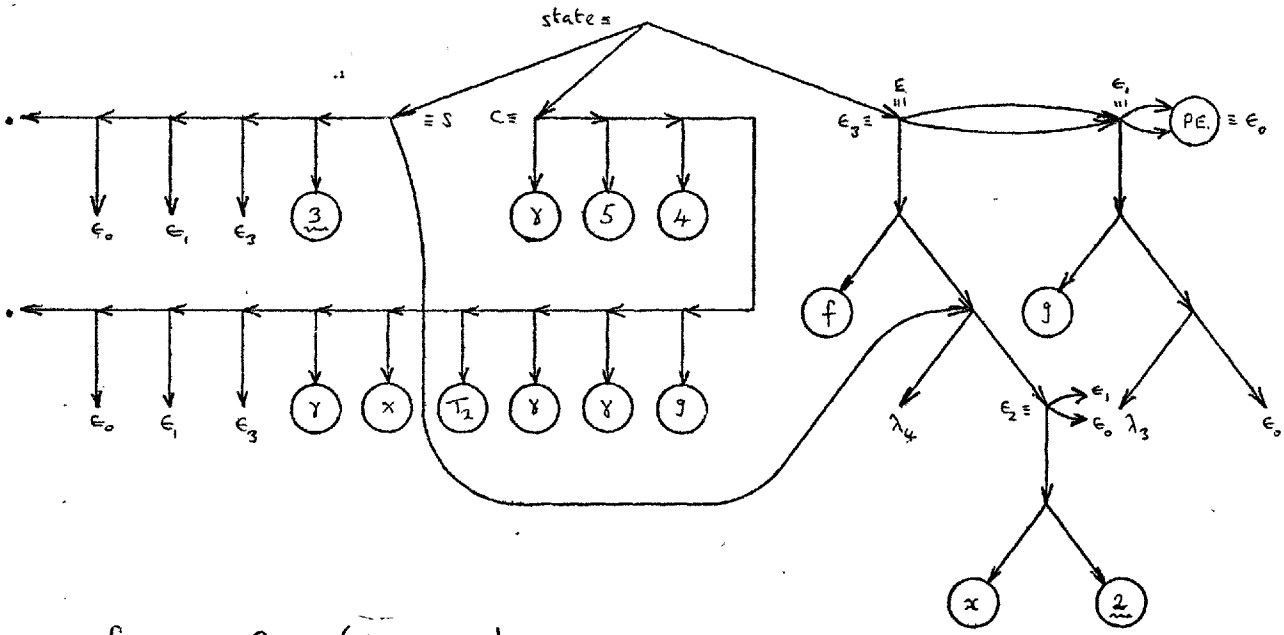


figure B12 (line 14)

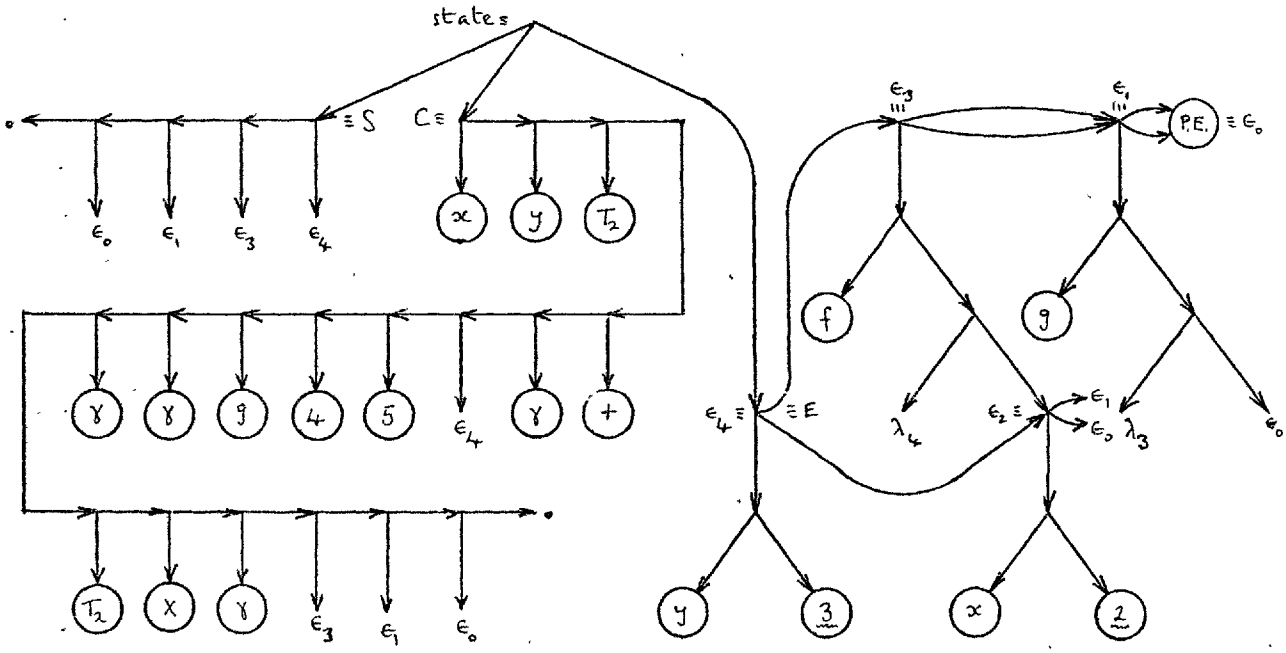


figure B13 (line 16)

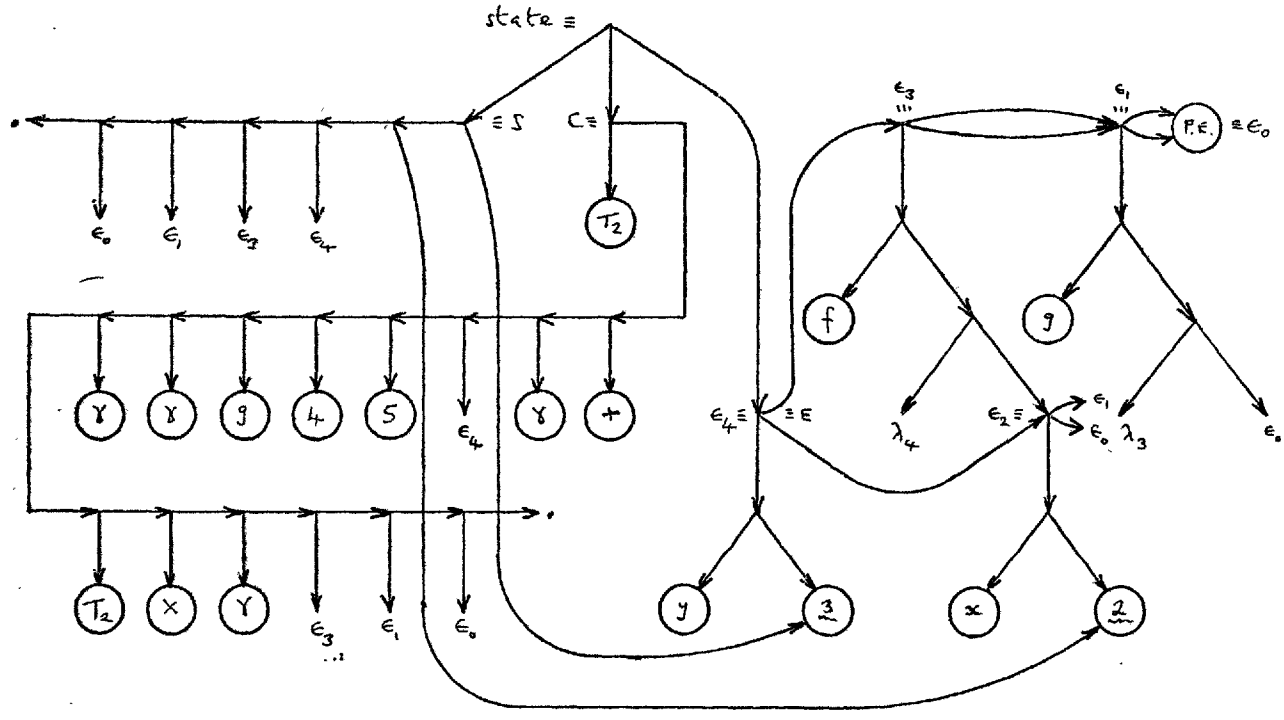
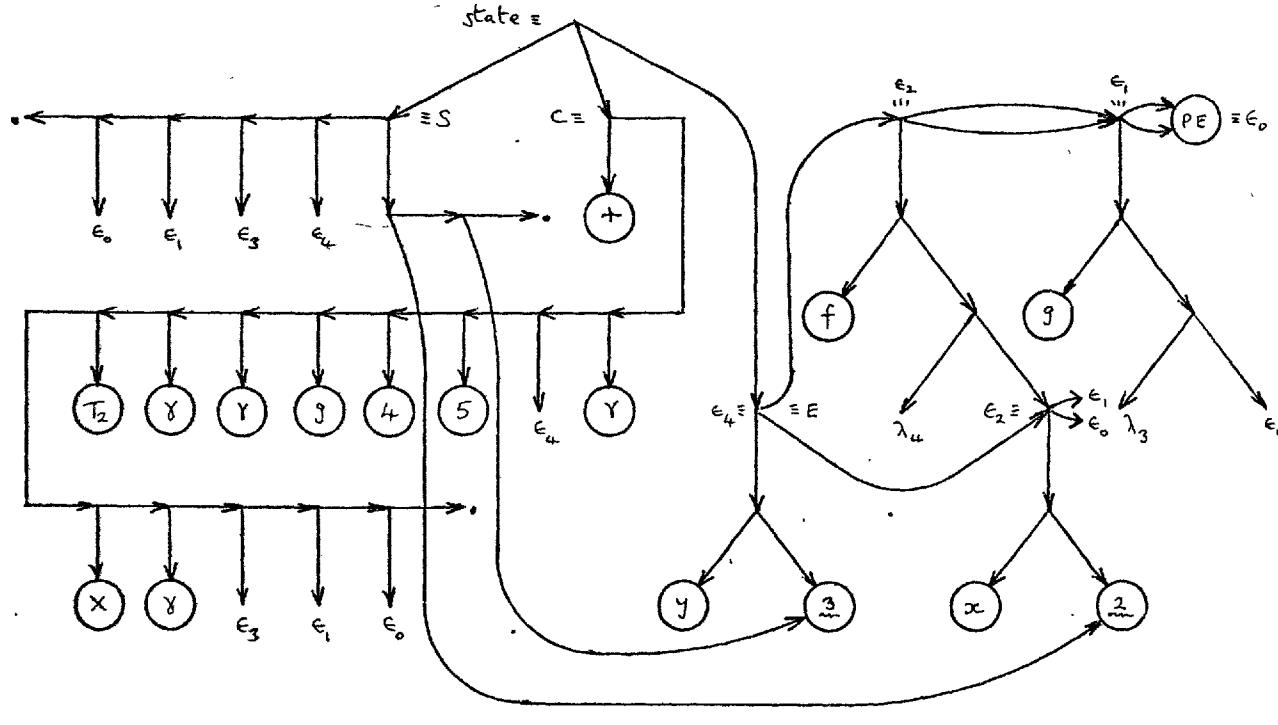


figure B14 (line 16)



References

B1 Bell, M.G., Lecture Notes, The Institute of Computer Science, 1969, 1970.

B2 Böhm, C., "The CUCH as a Formal and Description Language", Formal Language Description Languages for Computer Programming, Steel, T.B. ed., North Holland, 1966, pp. 179-197.

B3 Burstall, R.M., "Semantics of Assignment", Machine Intelligence 2, Edinburgh University, pp. 3-20.

C1 Church, A., The Calculi of Lambda-Conversion, (Annals of Mathematics Studies No. 6), Princeton University Press, 1941.

C2 Curry, H.B. and Feys, R., Combinatory Logic, North Holland, 1958.

D1 Dijkstra, E.W., "Co-operating Sequential Processes", Programming Languages, Genuys, F. ed., Academic Press, 1968, pp. 14-112.

E1 Evans, A., PAL – A Reference Manual and a Primer, M.I.T. Department of Electrical Engineering, September 1968.

F1 Fischer, M.J., "Lambda Calculus Schemata", Sigplan Notices Vol. 17 No. 1 Jan. 1972, and Sigact News No. 14 Jan. 1972, pp. 104-109.

L1 Landin, P.J., "The Mechanical Evaluation of Expressions", Computer Journal, Vol. 6, pp. 308-320.

L2 Landin, P.J., "A Correspondence Between ALGOL 60 and Church's λ-notation, Parts I & II", CACM, Vol. 8 Nos. 2 & 3, 1965, pp. 89-101 & pp. 158-165.

L3 Landin, P.J., "The Next 700 Programming Languages", CACM, Vol. 9 No. 3, 1966, pp. 157-166.

L4 Lee, J.A.N., Computer Semantics, Van Nostrand Reinhold, 1972.

L5 Lucas, et al., Method and Notation for the Formal Definition of Programming Languages, IBM Laboratory Vienna, TR 25.087.

L6 Lucas, P. and Walk, K., "On the Formal Description of PL/I". Annual Review in Automatic Programming Vol. 6 (Part 3), Pergamon Press, 1969, pp. 105-182.

M1 McCarthy, J., "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, Braffort, P. & Hirschberg, D. eds., pp. 33-70.

M2 McCarthy, J., "Towards a Mathematical Science of Computation", Information Processing 1962, Popplewell, C.M. ed., pp. 21-28.

M3 McGowan, C., "Correctness Results of λ-calculus Interpreters", Ph.D. thesis, Cornell University, 1971.

M4 McGowan, C., "The Correctness of a Modified SECD Machine", Second ACM Symposium on Theory of Computing, 1970, pp. 149-157. Also M3.

M5 McGowan, C., "The "most recent" error: its causes and correction", Sigplan Notices Vol. 17 No. 1 Jan. 1972, and Sigact News No. 14 Jan. 1972, pp. 191-202.

M6 Morris, J.H., "Lambda-Calculus Models of Programming Languages", Ph.D. thesis, M.I.T., 1970.

M7 Morris, J.H., "A Bonus from Van Wijngaarden's Device", CACM, Vol. 15 No. 8, 1972, p. 773.

N1 Neuhold, E.J., "The formal description of programming languages", IBM Systems Journal, No. 2, 1971, pp. 86-111.

R1 Randell and Russell, ALGOL 60 Implementation, Academic Press, 1964.

R2 Rosenbloom, F.C., The Elements of Mathematical Logic, Dover Publications, 1950.

R3 Rosser, J.B., "A Mathematical Logic Without Variables, I & II", Annals of Mathematics Vol. 36 No. 1, 1935 pp. 127-150, and Duke Mathematical Journal Vol. 1, 1935 pp. 328-355.

S1 Standish, T.A., "A Data Definition Facility for Programming Languages", Ph.D. thesis, Carnegie Institute of Technology, 1967.

W1 Wegner, P., Programming Languages, Information Structures and Machine Organisation, McGraw Hill, 1968.

W2 Wegner, P., "Three Computer Cultures", Advances in Computers Vol. 10, Academic Press, 1970, pp. 7-78.

W3 Wegner, P., "The Vienna Definition Language", Computing Surveys, March 1972, pp. 5-63.

W4 Wijngaarden, A. Van, "Recursive Definition of Syntax and Semantics", Formal Language Description Languages for Computer Programming, Steel, T.B. ed., North Holland, pp. 13-24.

W5 Van Wijngaarden, et al., Report on the Algorithmic Language ALGOL 68, Numerische Mathematik, 14, 79-218(1969).

Abbreviations

ACM	Association for Computing Machinery.
CACM	Communications of the ACM.
IBM	International Business Machines.
M.I.T.	Massachusetts Institute of Technology.

Sigplan Notices 17(1) and Sigact News 14 was the Proceedings of an ACM Conference on Proving Assertions About Programs.