COMPUTER AIDED MANIPULATION OF SYMBOLS

A

THESIS

submitted in fulfilment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

in the

FACULTY OF SCIENCE

of the

QUEEN'S UNIVERSITY OF BELFAST

by

FREDERICK VALENTINE McBRIDE M.Sc.

MAY 1970

ACKNOWLEDGEMENTS

This work was supported by the grant of a Postgraduate Studentship from the Ministry of Education for Northern Ireland.

The project was carried out in the Department of Computer Science of the Queen's University of Belfast, initially under Professor J. C. Browne as Head of Department (now Director of the Computing Laboratory of the University of Texas) and subsequently under Professor C. A. R. Hoare.

The author wishes to express his gratitude to his supervisor, Professor R. M. Pengelly, for his advice and encouragement. The author also wishes to thank Dr. D. J. T. Morrison for many constructive discussions and suggestions, and the staff of the University's Computing Laboratory, whose assistance during the course of this work was greatly appreciated.

C O N T E N T S

ABSTRACT

This thesis is concerned with an investigation into the techniques of computer aided manipulation of symbols. The overall objective of this study is to develop a general purpose, on-line symbol manipulation system incorporating features which enable the user to modify and extend the manipulative capabilities of the basic processor. These requirements can be met only by the choice of a low-level language (LISP 1.5 was a convenient starting choice in this instance), in which the user can apply the primitive facilities provided to the construction of the particular capability required.

However, it is felt that the descriptions of many desirable, higher level capabilities (such as differentiation, integration and simplification), afforded by existing symbol manipulation systems, are so complex that the tasks of modification and extension are made impractical if not impossible, especially in an on-line environment. Thus, more 'natural' modes of description are provided by adding matching processes to the standard LISP interpreter and by introducing a slightly altered syntax into the LISP 1.5 language.

The extended LISP system is embedded within an interactive environment to enable the user to guide the manipulative processes being performed. The fundamental importance of such an interactive capability arises from the fact that no program can currently match the power of its human user to recognize 'significant' or 'useful' items of information.

CHAPTER I

Introduction

This thesis is concerned with an investigation into the techniques of computer aided manipulation of symbols. One of the results of this examination was the development of a programming system, whose description provides much of the content of this report. This chapter introduces the objectives of the investigation, together with an indication of the methods adopted in the attempt to achieve these goals in the subsequent programming system. To provide a basis for the ensuing discussion there follows an outline of the general concepts of symbol manipulation.

1.1 Symbol manipulation

The term 'symbol manipulation' is used to mean a variety of things. Some think of it as a synonym for 'string processing', that is, the manipulation of a sequence of characters, where each character is normally treated as an individual. Others connect the term with the phrase 'list processing', where this refers to the techniques of processing information stored in a computer in some non-contiguous manner, with pointers from each piece of data to the next. 'Non-numerical computations' and 'any manipulations, other than arithmetic, performed on a computer' are yet other suggested meanings.

Although there is to be no attempt to define the term rigorously, this thesis considers symbol manipulation to be a branch of computing science concerned with the processing of unpredictably structured data. In most commercial and scientific computer applications, the data to be processed is of known length and format. In contrast, the size and format of the data involved in symbol manipulation are not known in advance and often vary

greatly during the running of the program. This point of view in no way contradicts the other suggested alternatives, for clearly, data which does not have a pre-determinable format will require dynamically allocatable storage such as that provided in list or string processing systems. These systems differ with respect to the generality of the lists upon which they operate. LISP,[12], SLIP,[24], IPL,[16], and L6,[10], are list processing languages which operate on lists in their most general form, while string processing languages, such as SNOBOL,[6], and COMIT,[26], use single level lists only, these single level lists being called strings.

One particular field of application for symbol manipulation systems, which has aroused great interest in recent years, is that of algebraic manipulation.† Following Sammett[21], algebraic manipulation is considered as "the computer processing of formal mathematical expressions without any particular concern for their numeric values ... this term in no way excludes the use of trigonometric or other types of functions." It must be stressed that inclusion in this field is based more on intent than on the actual techniques employed during the processing; for example, the internal operations performed by most systems designed specifically to manipulate polynomials or rational functions, are more numeric than symbolic, but, because the motivation is to operate on algebraic expressions, such systems are considered within the field of algebraic manipulation. However, having stated these exceptions, in general, algebraic manipulation may be considered as a particular field of application for symbol manipulation systems where the internal lists or strings represent formal mathematical expressions.

Although the programming system, to be described later, was not designed specifically with algebraic manipulation in mind, it does possess a

†For a survey of algebraic manipulation see Sammett[20,21].

distinct bias towards this field. Therefore, it is worthwhile to consider this area of symbol manipulation in greater detail, with particular reference to the levels of capability to be supplied to a user.

1.2 Levels of capability

In nearly every area of computer processing, and algebraic manipulation is no exception, two distinct philosophies or attitudes can be identified. The first, or lower level approach, is based on the concept of minimum assumption and maximum flexibility, sometimes called the 'primitive tools' approach. This type of system provides only basic processing functions from which the user is expected to build up his own desired capabilities. In the context of algebraic manipulation, list and string processing systems lie in this category. The alternative attitude is to provide the user with capabilities specific to his requirements. From the point of view of algebraic manipulation, this higher level approach might involve the provision of commands such as "differentiate", "integrate" or "simplify".

The main failing of higher level languages, such as FORMAC [1], MATHLAB, [4,5], REDUCE [9], and Martin's Symbolic Mathematical Laboratory [14], is that, while they provide some very useful specific capabilities, they do not make allowance for either the definition of new capabilities or the alteration of existing ones in a straightforward fashion. Indeed, a change which might appear relatively trivial on the surface will often involve a chain of complex modifications which only someone who is familiar with the system's design features could hope to perform.

The alternative is to adopt the more flexible, primitive tools approach. Engeli, [3], expressed his opinion as follows:- "In spite of the inclusion of a large number of capabilities, no system will ever satisfy the needs of all its users unless those users are given the means to expand the system.

Any features added using the definitional capabilities necessarily involve a high degree of interpretiveness, often down to the level of axioms."

Such flexibility cannot be bought cheaply however, for the user must pay the price of an increased burden of 'awareness'.† Supporters of the higher level approach argue, undoubtedly with some justification, that this burden lowers a user's problem solving potential, for too much of his effort is spent on the definitions and modifications of his desired capabilities and not enough on the problem in hand. Furthermore, they continue, the majority of users are really only interested in solving their own particular problem, and therefore are prepared to accept the supplied, specific capabilities and do not want to concern themselves with the creation of new capabilities or the amendment of existing ones.

While these arguments have considerable merit, their counterparts cannot be ignored. Modularity in a language, as well as in its implementation, is needed, for as new topics, methods and notations are developed, the system will need to be readily extendible to deal with them. Also, the solution of certain problems in algebraic manipulation depend on the context of evaluation rather than on some set of pre-determinable rules. The most oft quoted example of this nature is that of algebraic simplification. Who can predetermine if $a(b + c)$ is simpler than $ab + ac$? Indeed, in certain contexts, it could be argued that $a + 0$ is a simpler form than a, and $\text{cosec}^2 x - \text{cosec } x \cot x$ than $\frac{1}{1 + \cos x}$. In the second of these examples, while $\text{cosec}^2 x - \text{cosec } x \cot x$ may appear more cumbersome than $\frac{1}{1 + \cos x}$, the former is immediately integrable, whereas the normal method for the latter would involve a substitution for $\tan \frac{x}{2}$.

---

†'awareness' - term introduced by Weizenbaum to indicate the attention to detail which must be maintained in any given situation.

1.3 The purpose of this project

The most obvious resolution of the conflict, outlined in the preceding section, is to provide a programming system which has not only a wide range of specific capabilities but which also allows the creation of new features and the alteration of existing ones in a straightforward manner. It would also appear that the main obstacle, preventing the realisation of such a solution, is the wide gap between the capabilities of currently available list and string processing languages and most desirable, higher level capabilities. In other words, programs, written in existing symbol manipulation languages to describe such processes as integration, simplification or even less difficult problems such as differentiation, are so complex that no user could be expected to alter or add to them. Clearly, if this solution is to be pursued any further, this gap must be narrowed, and, as it is not meaningful to change the specifications of the higher level capabilities, the only possibility left open is to seek for a more natural form of description. Thus, this became the major objective of this project.

Earlier it was suggested that a system which demands a high awareness will lead to a reduction in a user's problem solving potential. In the context of programming languages, intuition suggests that awareness is the complement of naturalness, that is, if naturalness of description can be increased, then awareness must be decreased. Therefore, it would appear that if a more natural form of description can be achieved then there should be a resultant increase in a user's problem solving potential.

Two aspects of naturalness are considered, namely, transparency and conciseness, and techniques, to advance both facets, are developed with respect to an existing symbol manipulation language, LISP 1.5,[12]. This language was chosen because of its widespread use in many impressive projects, thereby permitting the possibility of making fairly direct comparisons with existing

programs. Further, (and more basically) a LISP interpreter was available for the ICL 1907 at the Queen's University of Belfast, having been implemented by the author for another project, [11].

So far, this statement of purpose has been concerned only with the language component of a programming system. However, all such systems possess a second component, which will be considered in this thesis as the environment in which a user's evaluations are to take place. The study of this component divides into three sections. The first part consists of the defining and editing facilities required to permit the creation and modification of the desired capabilities. In the second section comes an examination of methods whereby the user can exert more control over the context of evaluation, thus increasing the practicality or usefulness of the system. Finally, facilities are investigated which will allow the system and user to interact in such a way that each can receive guidance from the other. Both components of the programming system, language and environment, are further discussed in the next section.

1.4 The programming system

The programming system is based on a LISP interpreter, which, in turn, is modelled on the LISP 1.5 interpreter implemented for the IBM 7090 by McCarthy et al., [12].

1.4.1 Language

The interpreter has been extended primarily by the introduction of pattern matching facilities. It has become clearly established that pattern-directed languages are convenient vehicles for the description of many symbol manipulation algorithms, and, in particular, those relating to algebraic manipulation. For example, even some highly command orientated languages,

such as FORMAC, include some pattern recognition facilities (e.g. the PART command). Thus, in the context of the LISP language, it would appear that a reasonable first step in the search for increased naturalness would be the introduction of a predicate function, which could be supplied with two arguments, a pattern and an argument expression, and, whose operation would be to determine, according to some matching algorithm, whether the given expression matches the given pattern.

A highly sophisticated example of this approach is the function schatchen, developed by Moses and utilised in his very impressive symbolic integration system, SIN,[15]. This matching function is programmed with a wealth of knowledge concerning the behaviour of algebraic operators, including such properties as commutativity associated with addition and multiplication and the special relationships of zero and one with regard to algebraic operations. At the other end of the matching scale, there is the standard LISP function pair, which, when supplied with two lists as arguments, creates an association list by pairing the corresponding elements of the two given lists. The pattern matching facility developed for this programming system has no in-built knowledge of the special behavioural characteristics of algebraic operators, in fact, it does not possess a knowledge even of the existence of such operators. However, provision is made whereby the existence and special characteristics of any operator may be indicated in a straightforward manner. Thus, this system allows a user controlled flexibility ranging from the basic pair operation to something approaching the sophistication of schatchen.

The introduction of such a pattern matching facility does not, in itself, change the descriptive formalisms of the LISP language, that is, new functions must still be created in terms of existing ones through the method of composition or by using the standard conditional expression formalism. Such an introduction merely makes available a new and powerful predicate for use

within the latter method of definition. Thus a second and possibly more significant step in the search for increased naturalness is suggested. Instead of implementing the matching facility merely by providing a predicate, the process is embedded in the interpreter as a new evaluation procedure. This permits an extension to be made to the syntax of LISP, consisting of the introduction of a new function type, named RULE, which is based on an extended conditional expression formalism. Thus, the new matching procedure, named match, performs an evaluation task for RULE type functions analogous to that performed by the function evcon for CONDitional expressions. This extension in semantics and syntax in no way affects the use of standard LISP function types; standard LISP is a subset of the programming language available to the user.

1.4.2 Environment of evaluation

Three areas are investigated with regard to this component; firstly, the definition and amendment of those capabilities desired by a user; secondly, the environment for the actual use of the capabilities once created, and the user's control over the context of evaluation; and finally, interactive facilities which permit a user to exert control over the evaluation procedures while they are in progress and the improvement of some of the diagnostic features in a search for a more helpful system.

This last area of investigation assumes that the system should operate on-line. There are several reasons why symbol manipulation systems, in common with other programming systems, benefit from an on-line environment. The advantages which are most frequently quoted are rapid turnaround, ease of debugging and ease of program alteration. Another reason, peculiar to symbol manipulation systems, arises from the necessity that, in certain instances, a user must be aware of intermediate results before a decision

can be made about the subsequent procedure. This also leads to the adoption of a command orientated control structure, that is, read a command, evaluate the issued command, display the value, and then invite the user to give his next command and so on.

With regard to the first area of investigation, a comprehensive set of defining and editing commands are introduced, through which RULE type functions may be created and amended. Because of the interpretive nature of the evaluation process, definitions are held internally as list structures. Thus the problem of amendment reduces to the relatively simple tasks of deleting or inserting a sub-list of a list in some indicated position. With conventional LISP formalisms, the most difficult part of such operations is finding a suitable method of indicating the point in the list structure at which the insertion or deletion is to take place. In this system, this particular difficulty is substantially overcome by the use of labels in the new RULE type functions.

The examination of the second area is concerned with two features, namely, storage management and identification bindings. With regard to the former, the facilities not only enable the user to exert more control over the management of his in-core working space, but also give him access to any number of private disc files on which he may save any information he wishes to safeguard, and from which, such information may subsequently be unsaved to create some desired context of evaluation. The identification facilities operate in a similar manner to ALGOL or FORTRAN assignment statements, and when coupled with a simple pre-processor permit easy reference to the results of previous calculations.

The third and final area of investigation considers the introduction of interactive facilities which are designed to enable the user and system to communicate useful information one to the other. In some of the cases where

a conventional LISP system can only produce an error diagnostic and then abandon the execution, this system has built-in halts and associated queries, allowing the user, in many instances, to correct the fault and restart the evaluation process. Furthermore, the user is provided with the ability to program a halt or a query to take place during the matching process and thus can guide the actual evaluation procedures of the system. A variety of restart facilities are also provided, corresponding to the different ways in which a user might wish to recommence the execution.

1.5 Layout

A detailed account of the pattern matching facilities introduced, together with the changes these involve, is given in the next three chapters. Chapter II describes the basic matching algorithm and the new RULE function type, along with the alterations to the interpreter caused by their implementation. Two well known LISP examples, differentiation and the Wang Algorithm for the Propositional Calculus, are included as illustrations of the use of some of the new features. Some of the inadequacies of the basic matching system, especially with regard to algebraic operators, are discussed in Chapter III and a possible solution to some of the problems encountered is described. Chapter IV presents a description of the operation of the pattern-matching facilities in terms of AND-OR goal trees.

All the facets of the evaluation environment, introduced in §1.4.2, are presented in more detail in Chapter V.

The construction of a three stage symbolic integration function, as an illustration of the use of this system, is considered in Chapter VI. The first stage is based on a very simple table look-up scheme. The second stage utilises the method of 'derivative-divides'. The third stage is an implementation of the EDGE (EDucated GuEss) heuristic, which is based on the Liouville theory of integration, and which was developed by Moses for his SIN program.

In the seventh and final chapter, the author presents his conclusions and some suggestions for possible future developments.

Four appendices are also included. Appendix A gives the definitions and actions of all the functions involved in the creation of the evaluation environment, while Appendix B performs a similar task for the functions involved in the pattern-matching processes. An account of the overall operation of the system, together with an M-expression definition ofthe modified LISP interpreter, is presented in Appendix C. Finally, the listings of the actual performances of all examples mentioned in the text can be found in Appendix D.

CHAPTER II

Introduction of a Matching Process into LISP

2.1 Introduction

This chapter is concerned with the introduction of a simple matching process coupled with a new function type into the standard LISP interpreter. These extensions gain for the user a considerable increase in both transparency and conciseness of description, as can be observed by comparing the two illustrative examples, included at the end of the chapter, with the corresponding programs written in standard LISP. The next section gives some background material about the existing LISP descriptive formalisms and suggests a possible extension to the conditional expression form of definition. This is followed by two sections describing the actual matching algorithm adopted and the format of the new RULE function type. The remaining section details the alterations to the interpreter and gives definitions of the functions involved in the newly embedded matching process.

2.2 Descriptive formalisms

There are two methods provided in LISP by which new functions may be defined in terms of existing ones. The first of these methods is called "composition".

If all the variables occurring in a form e are among $x_1, \ldots, x_n$ then a function h may be defined by writing

$$h(x_1, \ldots, x_n) = e .$$

Then, if $f_1, f_2, \ldots, f_m$, are all the functions occurring in e, the function h is said to be defined by composition from $f_1, \ldots, f_m$

example $\quad x \leqslant y = x<y \vee x=y .$

This formalism permits only the definition of a rather limited class of functions. A more interesting and useful method of definition is based on the conditional expression formalism, [13]. A conditional expression has the form

$$(p_1 \rightarrow s_1;\ p_2 \rightarrow s_2;\ \ldots;\ p_n \rightarrow s_n),$$

where each p_i is an expression whose value may be truth, T, or falsity, F, and each s_i may be any expression. This form corresponds to the ALGOL 60 reference language expression

$$\underline{\text{if}}\ p_i\ \underline{\text{then}}\ s_i\ \underline{\text{else if}}\ p_2\ \underline{\text{then}}\ s_2\ \ldots\ \underline{\text{else if}}\ p_n\ \underline{\text{then}}\ s_n.$$

The value of a conditional expression is the value of the s_i corresponding to the leftmost p_i with value T.

example $(4 < 3 \rightarrow 7;\ 2 > 3 \rightarrow 8;\ 2 < 3 \rightarrow 9;\ 4 < 5 \rightarrow 7) = 9.$

Examples of the use of this method to define two well known functions are

$$|x| = (x < 0 \rightarrow -x;\ x > 0 \rightarrow x)$$

$$\delta_{ij} = (i = j \rightarrow 1;\ i = j \rightarrow 0)\ .$$

It is normal to make use of the truth value T to simplify these conditional forms where possible, thus the definition for $|x|$ might be rewritten as

$$|x| = (x < 0 \rightarrow -x;\ T \rightarrow x).$$

LISP allows recursive references to the function being defined when using this second method. Thus the factorial of a non-negative integer would be defined by

$$n! = (n = 0 \rightarrow 1;\ T \rightarrow n.\ (n - 1)!)\ .$$

Although these methods give LISP very powerful definitional capabilities, the resulting function descriptions are not always natural. An inspection of the design of many symbol manipulation algorithms indicates that the

construction of the predicates may be divided into two distinct phases: first, the isolation of specific structures and second, the application of constraints to the constituent elements of these structures. With the methods previously described, these operations must be represented by either a single complex predicate or a chain of simpler predicates. However, a more natural function description is possible if a suitable pattern matching system is introduced to isolate structures, leaving the predicates to describe only the elemental constraints. Such a definition might then take the form of an extended conditional expression like

$$(f_1, p_1 \rightarrow s_1; f_2, p_2 \rightarrow s_2; \ldots; f_n, p_n \rightarrow s_n) ,$$

where each f_i is a pattern or dummy form. The value of such an expression for a particular argument expression e would be the value of the s_i corresponding to the leftmost f_i which successfully "matches" e and for which the associated predicate p_i has the value T. An ALGOL-like expression for this form of definition would be

if $f_1 || e \wedge p_1$ then s_1 else if $f_2 || e \wedge p_2$ then s_2 ... else if $f_n || e \wedge p_n$ then s_n,

where the symbol '||' is to be read as 'matches'.

2.3 A matching algorithm

The matching algorithm introduced, which is similar to one used in the FAMOUS system of Fenichel, [7], is as follows. If a form f is to match an expression e then,

(a) if f is a number, the name of a defined function or the name of a constant, then e must be identical to f,

examples f = 2 matches only e = 2

f = + matches only e = + (assuming + is a defined function)

f = T matches only e = T .

The last example is one of a constant, that is, an atom with an APVAL indicator on its property list. Other in-built APVAL's of the system are F, *T* , NIL and OBLIST, and these atoms should be avoided when constructing forms unless their constant properties are explicitly desired.

(b) any atomic form, apart from those described in (a), matches any expression,

example f = x matches e = x, y, x+y, $e^{\log z}$ and all others.

(c) if f is a quotation of another form g, then the expression e must be identical to g,

example f = "x matches only e = x.

(d) if f consists of a list of elements $f_1, f_2, \ldots, f_n$ then

(i) e must consist of a list of elements $e_1, e_2, \ldots, e_n$, and

(ii) for i = 1, 2, ..., n, f_i must match e_i, and

(iii) if g is an atom which occurs more than once in f, then the corresponding subexpressions in e must be identical,

example f = (u v w) matches e = (x y z)

but f = (u u) does not match e = (x y)

however f = (u v w) matches e = (x x x) .

In introducing this particular matching algorithm, the author does not mean to suggest that it is the only possible (or even the best possible) one. However, it is suggested that such an introduction is useful and the illustrations included in the final section of this chapter will hopefully justify this opinion.

In general, numbers, defined functions and constants are only expected to match themselves; this is covered by (a) which is equivalent to the implicit quotation of these atom types. The "automatic" matching process, described in (b), is directly analogous to the pairing operation between the

LAMBDA variable list and the argument list in the evaluation of EXPR's in standard LISP. Explicitly quoted expressions are catered for by (c). Section (d) gives the structural matching criterion and also states a uniqueness condition, designed to avoid ambiguity.

2.4 RULE's

It was suggested earlier (at the end of §2.2) that if a suitable matching process were introduced, such as one based on the algorithm just described, then more natural function definitions would be possible. This claim is founded on the hypothesis that, if the predicate elements in a conditional expression are divided into two distinct parts, a test on the overall structure of the given argument expression followed by tests on the individual elements of the structure, then a more transparent and, in many cases, more concise function description is obtained. The section that follows introduces such a new descriptive formalism.

The new function type is characterised by the indicator RULE on the property list of an atom.

RULE - S-expression defining a function whose name is the atomic symbol on whose property list the indicator RULE appears.

The body of the definition bears a close resemblance to the form of the extended conditional expression, which was discussed previously, that is

$$(f_1, p_1 \rightarrow s_1; f_2, p_2 \rightarrow s_2; \ldots; f_n, p_n \rightarrow s_n) .$$

The actual S-expression has the form

```
(DARG (D1 D2 ... DN)
A1 (ASSERTION 1)
A2 (ASSERTION 2)
.
.
.
AM (ASSERTION M))
```

where the atom DARG serves a similar purpose for RULE'S as LAMBDA does for EXPR's; (D1 ... DN) is a list of dummy arguments and A1, ..., AM are called assertion labels and are present primarily to alleviate the problem of amendment in the on-line environment (cf. §5.2). An assertion label may be any allowable LISP atom.

The format of an individual assertion is

(form f, substitute s, predicate p) ,

that is, a three-element list. The predicate has been positioned third in this list, so that a two-element list can be taken to denote an implicitly true predicate.

2.5 Alterations to the interpreter

2.5.1 Changes to apply and eval

Having introduced a new function type and a matching algorithm which is to form the basis of the evaluation procedure for this new function type, there remain two aspects of the system as yet undescribed. These are, firstly, the alterations to the interpreter caused by the embedding of a matching process, and secondly, the definition of the functions actually utilised in this matching process. Unfortunately, to understand the former, a knowledge of the LISP functions apply and eval is required. (See pp. 70, 71 of the LISP Programmer's Manual, [12].) A complete M-expression definition of the modified interpreter is given in Appendix C of this thesis.

In all, three extensions are needed, two in apply and one in eval. Because of the similarity between RULE's and the CONDitional expression type of EXPR, it is instructive to present these alterations in conjunction with the corresponding instructions for handling EXPR's.

In apply, after the line

get [fn; EXPR] → apply [expr; args; a] ;

the new line

get [fn; RULE] → apply [rule†; args; a] ;

is inserted, and after the line

(i) eq[car [fn]; LAMBDA] → eval[caddr[fn]; nconc[pair[cadr[fn]; args]; a]] ;

the new line

(ii) eq[car[fn]; DARG] → eval[match[cddr[fn]; args]; alist] ;

is added.

In eval, after the line

get [car [form]; EXPR] → apply [expr; evlis [cdr [form]; a]; a] ;

the new line

get [car [form]; RULE] → apply [rule†; evlis [cdr [form]; a]; a] ;

is inserted.

Perhaps the simplest way to highlight both the similarities and the differences between the methods used by the interpreter to handle EXPR's and RULE's is to trace the operation of the interpreter for both cases.

Consider a typical EXPR definition body, which has the form

(LAMBDA (L1 ... LN)(COND (p_1 s_1)(p_2 s_2) ... (p_m s_m))).

This is processed first by line (i) of apply, which passes control to eval with a first argument (or form) of

(COND (p_1 s_1) ... (p_m s_m))

and a second argument (or a) which is the result of pairing the elements of the LAMBDA list (L1 ... LN) with their corresponding elements in the given

---

† Here the apparently undefined variable rule = value of get [fn; RULE]
= (DARG (D1 ... DN) A1(ASSERTION 1) ... AM(ASSERTION M))

argument expression and then appending the resultant list to the current association list (a-list). The relevant line in eval is

eq [car [form]; COND] → evcon [cdr [form]; a] ;

and through this, control passes to evcon with a first argument

$((p_1\ s_1) \ldots (p_m\ s_m))$

and a second argument which is still the a-list created earlier. Now each p_i will be evaluated in sequence (starting from the leftmost) with respect to the a-list until one with value T is found, and then the value of the corresponding s_i, again with respect to the a-list, is taken as the value of the CONDitional expression. These evaluation procedures may, and often do, involve recursive re-entry into the interpreter.

A typical RULE definition body takes the form

(DARG(D1 ... DN) $lab_1(f_1\ s_1\ p_1)$ lab2(f2 s2 p2) ... $lab_m(f_m\ s_m\ p_m)$).

It is first handled by line (ii) of apply, which passes control to eval with, as the first argument, the value of

match [$(lab_1(f_1\ s_1\ p_1) \ldots lab_m(f_m\ s_m\ p_m))$; args] ,

and as second argument, the value of the special atom ALIST. The value of this call to match is the s_i corresponding to the first encountered f_i, p_i combination for which the f_i matches the given argument expression, according to the algorithm presented in §2.3, and the value of the p_i with respect to the a-list, created during the matching process, is T. This a-list is transmitted back to apply from match via the special atom ALIST. Thus, the final value of the RULE is the value of this s_i again with respect to the a-list created during the matching process. Again, recursive re-entry to the interpreter is allowed during any of these evaluation procedures.

From these operational descriptions, it is apparent that match performs an evaluation task for RULE's analogous to the combination of pair and evcon for CONDitional expression type EXPR's.

2.5.2 The matching functions

The matching process is invoked from line (ii) of apply whenever the atom DARG is discovered as the first element of the function list. In reality, two new functions are involved, match and m2. The latter is a predicate, which is a direct implementation of the matching algorithm, described in §2.3, together with the code necessary for the task of creating the a-list. The operation of match is to send each form element in turn along with the given argument expression to m2. If this produces a value T, then the corresponding predicate element is tested with respect to the created a-list, which is transmitted through the atom ALIST. If this also yields a T value, then the corresponding substitute element is returned as the value of match. However, if either of these evaluations fail to produce T, then the process restarts with the next assertion. The M-expression definitions of match and m2 are as follows

```
match  [assertions; args] = [prog [u]
       u: = cdr [assertions] ;
  L1:  alist: = NIL;
       m2 [caar [u]; args] → [null [cddar [u]] V eval [caddar [u]; alist]
                                        → return [cadar [u]]];
       null [cdr [u]] → L2;
       u: = cddr [u] ;
       go [L1] ;
  L2:  print [NO MATCH FOR] ;
       print [args] ;
       return [NIL]]
```

```
m2[f;e] = [
null[f] → null[e] ;
atom[f]→numberp[f]∨ funp[f]→eq[f;e] ;
        null[sassoc[f;alist;NIL]]→prog2[alist:=cons[cons[f;e];alist];T];
        T→equal[cdr[sassoc[f;alist;NIL]];e]] ;
eq[car[f]; QUOTE]→[equal[cadr[f];e]→m2[cadr[f];e] ;
                   T → NIL] ;
T → m2[car[f]; car[e]] ∧ m2[cdr[f]; cdr[e]]]
```

The function, funp, used in m2, is a simple predicate, whose value is T if its atomic argument has any of the indicators APVAL, EXPR, SUBR, RULE, FEXPR or FSUBR on its property list, and F otherwise.

One final aspect of match requires further explanation. If, when evcon is processing a CONDitional expression, no true proposition can be found, then the interpreter outputs the error diagnostic A3 and abandons that particular evaluation. However, if match cannot find a true form-predicate combination, then the message NO MATCH FOR, followed by the argument expression for which no pattern is available, is printed out. The value NIL is then substituted for this expression and the overall evaluation procedure is continued. The usefulness of this procedure is illustrated in the differentiation example presented in the next section.

2.6 Examples

Two examples, differentiation with allied simplification and the Wang Algorithm for the propositional calculus, are presented for the purposes of illustration. These particular problems were selected because of the existence of well known LISP and SNOBOL programs for differentiation,[13,2], and a LISP program for the Wang Algorithm,[12]. This presentation assumes the existence of defining and editing features which have not yet been described

(see §5.2). Furthermore a notation, which is not available in practice, has been adopted to make the exposition clearer, that is, instead of writing

label(form, substitute, predicate)

a single assertion is written as

label: name [form] → substitute when predicate

where name is the name of the RULE being defined, and where the form, substitute and predicate are written in LISP M-expression format. Also, the commands are presented in a linearised infix notation, which is not available in practice.

2.6.1 Differentiation with allied simplification

The problem was to construct a function, named d, which would differentiate its first argument with respect to its second, and simultaneously, to build the necessary algebraic simplification rules. However, the system has no in-built knowledge of the behaviour or existence of the standard operators, and so the first task was to introduce six new functions +, -, *, /, ↑ and #, where the first five had their usual meaning and the sixth, #, represented unary minus. Initially, each of these was defined as a single assertion RULE, that is, + was defined by

last: + [a; b] → list [+; a; b]

and - was similarly defined by

last: - [a; b] → list [-; a; b]

as were *, / and ↑, while # was defined by

last: # [a] → list [#; a]

as # takes only one argument.

The first attempt to construct d could now begin, and this was

d1: d[y;x] → 0 when np[y]

d2: d[x;x] → 1

d3: d[+[u;v];x] → +[d[u;x]; d[v;x]]

d4: d[*[u;v];x] → +[*[u;d[v;x]];*[v;d[u;x]]]

d5: d[/[u;v];x] → /[-[*[v;d[u;x]]; *[u;d[v;x]]]; ↑[v;2]]

d6: d[↑[u;v];x] → *[*[v;↑[u;-[v;1]]]; d[u;x]] when np[v]

where np is a predicate which yields T if its argument is a number and F otherwise.

The command

d(#7 *z +3, z)

was given, and the response was

!! no match for

d(#7 *z, z)

nil + 0.

This prompted two additions, firstly, an assertion for d to deal with # terms, thus

d7: d[#[y];x] → #[d[y;x]]

was appended to the current assertion list of d, and secondly, an assertion for +,

a1: +[a;0] → a

which was added to +'s list of assertions before its last assertion. Now the same command produced

#(7*1 +z *0),

which suggested two new assertions for *,

a1: *[a;1] → a

a2: *[a;0] → 0

which were inserted before *'s last assertion. A further repeat of the command yielded

#7.

The command

d(6*z ↑ 3 +2 *z, z)

was given, and the response was

6*3*z ↑(3 - 1) + 2.

When the assertion

a1: - [a;b] → difference [a;b] when np [a] ∧ np [b]

was added to -, and

a3: * [a; * [b;c]] → * [times [a;b];c] when np [a] ∧ np [b]

was added to *, where difference and times are standard LISP functions which compute the difference and product of their numeric arguments respectively, then the value for this command became

18*z ↑2 + 2.

Next, the command

d(t/(t - 1), t)

was given, and this produced the response

!! no match for

d(t - 1, t)

((t - 1) - (t * nil))/(t - 1) ↑ 2.

This prompted the addition to d of

d8: d [- [u;v];x] → - [d [u;x];d [v;x]]

and then the command returned

((t - 1) - t)/(t - 1) ↑ 2.

The assertion

a2: - ⊢[a;b];a] → #[b]

was added to -, and the response advanced to

#1/(t - 1) ↑ 2.

The command

d(8*x↑2 + 8/x↑2, x)

was then issued, and the response received was

16 * x ↑ 1 + (0 - 16*x↑1)/(x ↑ 2) ↑ 2.

When the assertions

a1: ↑[a;1] → a

a2: ↑[↑[a;b];c] → ↑[a;*[b;c]]

were added to ↑, and the assertion

a3: -[0;b] → #[b]

was added to -, the command's response became

16*x + #(16 * x)/x ↑(2*2).

Then the assertion

a4: *[a;b] → times[a;b] when np[a] ∧ np[b]

was added to *, and the assertions

a1: /[#[a];b] → #[/[a; b]]

a2: /[*[a;b];↑ [b;c]] → /[a; ↑[b;-[c;1]]]

were added to /, and a further repeat of the command produced

16*x + #16/x ↑ 3.

Finally, an assertion for +, which stated

a2: + [a;#[b]] → - [a;b]

advanced the value to

16*x - 16/x ↑ 3.

At this stage two new RULE's were introduced, sin and cos, each of which was defined by a single assertion,

last: sin [a] → list [SIN;a]

for sin, and

last: cos [a] → list [COS;a]

for cos. Also, two new assertions were appended to d to cater for sin and cos terms,

d9:d[sin[u];x] → * [d[u;x]; cos [u]]

d10: d[cos[u];x]→ * [d[u;x]; # [sin[u]]]

The next command issued was

d(# cos(x) + sin(x), x)

and this yielded

#(1 * # sin(x)) + 1 * cos(x).

After the assertion

a5: * [1;a] → a

had been added to *, and

a1: # [#[a]] → a

had been added to #, the response became

sin(x) + cos(x).

The command

d(sin(4 * y), y)

produced

4 * cos(4 * y).

The command

d(sin(x) * cos(x), x)

gave the response

sin(x)* # sin(x) + cos(x) * cos(x).

The assertions

a6: * [a; # [b]] → # [* [a;b]]

a7: * [a;a] → ↑ [a;2]

were added to *, and then the command returned a value

sin(x) ↑ 2 + cos(x) ↑ 2.

Another assertion

a3: + [#[a];b] → - [b;a]

was added to +, and then the final response for this command was

cos(x) ↑ 2 - sin(x) ↑ 2.

The last example tried was

d(sin(t)/cos(t),t)

and this command yielded

(cos(t) ↑ 2 - # sin(t) ↑ 2)/cos(t) ↑ 2.

When the assertion

a4: - [a; #[b]] → + [a;b]

was added to -, and

a4: + [↑[cos [a];2] ; ↑[sin [a]; 2]] → 1

was added to +, the response to this command became

1/cos(t) ↑ 2.

The exercise was terminated at this point, not because any great difficulty was envisaged in further extensions, but rather because of a growing disquiet about the ad hoc nature of the simplification rules that were being produced. Also, shortcomings of this matching system were becoming apparent, especially with regard to such properties as commutation for + and * and the relationships involving the numbers 0 and 1. Thus, for example, not only was the assertion

a1: * [a;1] → a

needed by *, but also

a5: * [1;a] → a.

These shortcomings are further discussed in the next chapter. Appendix D contains a listing of the actual performance of this exercise, together with a display of the state of the RULE's d, +, -, /, ↑ and # at the time of termination.

2.6.2 The Wang algorithm for the propositional calculus

A description of both the theory and the LISP program for the algorithm may be found in the LISP Programmer's Manual on pp. 54-55, [12]. Some quotations from that section follow to explain the algorithm briefly.

"The Wang algorithm is a method of deciding whether or not a formula in the propositional calculus is a theorem.

"There are eleven rules of derivation. An initial rule states that a sequent with only atomic formulae (proposition letters) is a theorem if and

only if a same formula occurs on both sides of the arrow. There are two rules for each of the five truth functions - one introducing it into the antecedent, one introducing it into the consequent.

"The rules are so designed that given any sequent, we can find the first logical connective, and apply the appropriate rule to eliminate it, thereby resulting in one or two premises which, taken together, are equivalent to the conclusion. This process can be continued until we reach a finite set of sequents with atomic formulae only. Each connective-free sequent can then be tested for being a theorem or not, by the initial rule. If all of them are theorems then the original sequent is a theorem and we obtain a proof; otherwise we get a counterexample and a disproof."

In the coding to follow, the initial rule is simulated by a single assertion, which is labelled true or false, placed at the end of arrow's assertion list, so that it will only be attempted when all the logical connectives have been removed.

The other ten rules are simulated directly and are given the same labels as they have in Wang's description, namely, P_{2a}, P_{2b}, ..., P_{6a} and P_{6b}. The four arguments of the RULE arrow are

l1 - atomic formulae on the left side of arrow,

l2 - other formulae on the left side of arrow,

r1 - atomic formulae on the right side of arrow,

r2 - other formulae on the right side of arrow,

and these correspond directly to arguments a1, a2, c1 and c2 for the function th in the LISP program. The assertions labelled stkrhs and stklhs perform the "stacking" operations for the right and left sides of the arrow respectively.

The function test, written as an EXPR, merely places the given sequent into r2, that is, to the right of the arrow, and sets l1, l2 and r1 to be nil. The predicate joint, also coded as an EXPR, gives T if its arguments

intersect and F otherwise. The M-expression definitions of joint and test are as follows

```
joint [x;y] = [null [x] → F ;
               member [car [x];y] → T ;
               T → joint [cdr [x];y]]
test [s] = arrow[NIL; NIL; NIL; list [s;NIL]]
```

The definition of the RULE arrow is given in the notation introduced for the differentiation example.

```
arrow [l1;l2;r1;r2] =
stkrhs: arrow[l1;l2;r1; [x;r2]] → arrow[l1;l2;cons[x;r1];r2] when atom[x]
stklhs: arrow[l1; [x;l2];r1;r2] → arrow[cons[x;l1];l2;r1;r2] when atom[x]
   p2a: arrow[l1;l2;r1; [not[p];r2]] → arrow[l1;list[p;l2];r1;r2]
   p2b: arrow[l1; [not[p];l2];r1;r2] → arrow[l1;l2;r1;list[p;r2]]
   p3a: arrow[l1;l2;r1; [and[a;b];r2]] → and[arrow[l1;l2;r1;list[a;r2]];
                                             arrow[l1;l2;r1;list[b;r2]]]
   p3b: arrow[l1; [and[a;b];l2];r1;r2] → arrow[l1;list[a;list[b;l2]];r1;r2]
   p4a: arrow[l1;l2;r1; [or[a;b];r2]] → arrow[l1;l2;r1;list[a;list[b;r2]]]
   p4b: arrow[l1; [or[a;b];l2];r1;r2] → and[arrow[l1;list[a;l2];r1;r2];
                                           arrow[l1;list[b;l2];r1;r2]
   p5a: arrow[l1;l2;r1; [implies[a;b];r2]] → arrow[l1;list[a;l2];r1;list[b;r2]]
   p5b: arrow[l1; [implies[a;b];l2];r1;r2] → and[arrow[l1;list[b;l2];r1;r2];
                                                 arrow[l1;l2;r1;list[a;r2]]]
   p6a: arrow[l1;l2;r1; [equiv[a;b];r2]] → and[arrow[l1;list[a;l2];r1;list[b;r2]];
                                               arrow[l1;list[b;l2];r1;list[a;r2]]]
   p6b: arrow[l1; [equiv[a;b];l2];r1;r2] →and[arrow[l1;list[a;list[b;l2]];r1;r2] ;
                                              arrow[l1;l2;r1;list [a;list[b;r2]]]]
true or false: arrow[l1;l2;r1;r2] → joint[l1;r1] .
```

Now if test is given a logical sequent as its argument, a value T or F will be returned according to whether the sequent is a theorem or not.

Some minor additions and a slight reorganisation converts this program into a much more useful one. Not only can the validity of a theorem be checked but also the steps taken to reach the proof or disproof can be displayed, accompanied by the label of the Wang rule used for each step.

The definition of joint remains unaltered, but test is redefined as

```
test[s] = arrow [START; NIL; NIL; NIL; list [s;NIL]]
```

The RULE arrow now takes five arguments, that is, a new label argument (set initially to START by test) is added to the front of the original four. Arrow only performs the stacking operations and gives a print out of each step; the actual work of the algorithm is performed by a new RULE function, named arr. The definitions of these two RULE's are as follows.

```
arrow [label;l1;l2;r1;r2] =
stkrhs: arrow [label;l1;l2;r1; [x;r2]] → arrow [label;l1;l2;cons [x;r1];r2] when
                                                        atom [x]
stklhs: arrow [label;l1; [x;l2];r1;r2] → arrow [label;cons [x;l1];l2;r1;r2] when
                                                        atom [x]
printout: arrow [label;l1;l2;r1;r2] → prog2[print[list[label;col;l1;scol;l2;
                                        point;r2;scol;r1]]; arr[l1;l2;r1;r2]],
```

where the M-expression col translates into the S-expression (QUOTE :)
and the M-expression scol translates into the S-expression (QUOTE ;)
and the M-expression point translates into the S-expression (QUOTE →).

```
arr [l1;l2;r1;r2] =
p2a: arr [l1;l2;r1; [not [p];r2]] → arrow [P2A;l1;list [p;l2];r1;r2]
p2b: arr [l1; [not [p];l2];r1;r2] → arrow [P2B;l1;l2;r1; list|p;r2||
p3a: arr [l1;l2;r1; [and [a;b];r2]] → and [arrow [P3A1;l1;l2;r1;list [a;r2]];
                                        arrow [P3A2;l1;l2;r1;list [b;r2]]]
```

```
p3b: arr[l1;[and[a;b];l2];r1;r2] → arrow[P3B;l1;list[a;list[b;l2]];r1;r2]
p4a: arr[l1;l2;r1;[or[a;b];r2]] → arrow[P4A;l1;l2;r1;list[a;list[b;r2]]]
p4b: arr[l1;[or[a;b];l2];r1;r2] → and[arrow[P4B1;l1;list[a;l2];r1;r2];
                                      arrow[P4B2;l1;list[b;l2];r1;r2]]
p5a: arr[l1;l2;r1;[implies[a;b];r2]] → arrow[P5A;l1;list[a;l2];r1;list[b;r2]]
p5b: arr[l1;[implies[a;b];l2];r1;r2] → and[arrow[P5B1;l1;list[b;l2];r1;r2];
                                      arrow[P5B2;l1;l2;r1;list[a;r2]]]
p6a: arr[l1;l2;r1;[equiv[a;b];r2]] →and[arrow[P6A1;l1;list[a;l2];r1;list
                                                [b;r2]];
                                      arrow[P6A2;l1;list[b;l2];r1;list
                                                [a;r2]]]
p6b: arr[l1;[equiv[a;b];l2];r1;r2] → and[arrow[P6B1;l1;list[a;list[b;l2]];
                                                r1;r2];
                                      arrow[P6B2;l1;l2;r1;list[a;list
                                                [b;r2]]]]
true: arr[l1;l2;r1;r2] → prog2[print[VALID];T] when joint [l1;r1]
false: arr[l1;l2;r1;r2] → prog2[print[INVALID]; F] .
```

Now, say the command (in S-expression format)

```
TEST ((IMPLIES P(OR P Q)))
```

were given, that is, is $P \supset PVQ$ a valid theorem, then the response would be

```
(START: NIL; NIL → ((IMPLIES P(OR P Q)) NIL); NIL)
(P5A: (P); NIL   → ((OR P Q) NIL); NIL)
(P4A:  (P); NIL   → NIL; (Q P))
VALID
*T*
```

Some further examples and the listings of the definitions are presented in Appendix D.

CHAPTER III

An Extension of the Matching Process

3.1 Introduction

The matching system, described in Chapter II, employs a left-to-right one-to-one matching process. The provision of this process as an available LISP evaluation procedure coupled with the introduction of the new RULE function type, based on the extended conditional expression formalism, permits descriptions of many symbol manipulation algorithms which are undoubtedly more natural than those possible using standard LISP. However, in certain contexts, predominantly those involving the standard algebraic operators, the definitions of the sets of assertions required for the solutions of even relatively straightforward problems tend to become rather cumbersome. Protagonists of higher level manipulation languages (and possibly even some unbiased observers) would argue that the facilities so far described, do not allow for significant problems to be attempted, especially in an on-line environment. The experiments performed by Fenichel using the FAMOUS system, [7] whose main evaluation procedures are based upon a matching algorithm which is very similar to that described in §2.3, would appear to vindicate this point of view. Although he re-programmed the solutions to some problems previously written in LISP, including the Wooldridge-Russell Simplify system, [25], and also simulated the AUTSIM facility of FORMAC, [1], the descriptions demanded a considerable attention to detail; much more awareness, in fact, than might be deemed bearable by a user concerned primarily with the solution of his own particular problem.

As an illustration of some of the difficulties, consider a function linear [x; e] which is to determine whether or not e is linear with respect to x. (For the purposes of this illustration, the only operators involved in e are + and *.) Using the notation, introduced in §2.6, the forms and

predicates of such a RULE could be written as follows

a1: linear [x;x]

a2: linear [x;*[a;x]] when free [a;x]

a3: linear [x;*[x;a]] when free [a;x]

a4: linear [x;+[x;b]] when free [b;x]

a5: linear [x;+[b;x]] when free [b;x]

a6: linear [x;+[*[a;x];b]] when free [a;x] ∧ free [b;x]

a7: linear [x;+[b;*[a;x]]] when free [a;x] ∧ free [b;x]

a8: linear [x;+[*[x;a];b]] when free [a;x] ∧ free [b;x]

a9: linear [x;+[b;*[x;a]]] when free [a;x] ∧ free [b;x]

Here, nine assertions are required to describe a function, where a user might have hoped that one out of the last four would have been sufficient. However, if such a state is to be made possible, then, either the evaluation system must be aware that + and * are commutative operators and have in-built knowledge of the special relationships concerning the numbers 0 and 1, or else, some mechanism must be available through which such properties may be indicated to the system. To be consistent with the earlier decision to adopt a maximum flexibility approach, the second of these two possibilities is investigated.

Thus, means are provided whereby a user may indicate some properties as commutativity and the equivalence of such S-expressions as A, (+ A 0) and (* A 1). Notice that it is not possible to provide this information in assertional form associated with the RULE's + and *, because such assertions would only be processed during evaluations concerning + and *. Furthermore, the inclusion in + of an assertion

+ [a;b] → + [b;a]

makes the already possible event of infinite recursion extremely likely.

The next section introduces a new type of entity, called a transformation, through which a user may inform the system of those properties he wishes to be considered. The following two sections contain descriptions of the alterations to the format of assertions and the operation of the extended matching process respectively. The final section of this chapter presents a definition of linear, which utilises transformations, and gives a method of generating those left-to-right assertions equivalent to a single assertion with associated transformations.

3.2 Transformations

The format of a transformation is the same as that previously described for an assertion, namely,

label (form, substitute, predicate),

but the evaluation process is somewhat different. The objective of applying a transformation to an argument expression is to obtain an equivalent reconstitution of the expression; thus, after the form-match and the predicate testing have been performed as for assertions, the value is given simply by a direct replacement of the variables in the substitute by their a-list bindings. Atoms without bindings in the a-list remain unaltered. Thus, a transformation to indicate that + is a commutative operator might be written as

t1: + [a;b] → + [b;a]

and one to show the equivalence of A and (+ A 0) as

t2: a → + [a;0]

then t1 applied to an S-expression (+ x y) would yield the reconstitution (+ y x) and t2 applied to the same expression would give (+(+ x y)0).

If a transformation is desired, in which part of the substitute needs to be evaluated, then, this is indicated by using the atom eval. Thus, if one wishes to reconstitute an expression involving exponentiation to an even

power as the square of its square root, then the desired effect would be achieved by the transformation

t3: ↑[a;n] → ↑[↑[a;eval[quotient[n;2]]];2] when even[n] .

For example, t3 applied to an S-expression (↑ x 6) would yield the reconstitution (↑(↑ x 3)2). (See §6.5(iii) for an example of the use of this particular transformation.)

All transformations in the system are placed on the property list of the atom TRFS, which is treated as a RULE type function by the defining and editing facilities. (These facilities are described in §5.2.)

3.3 Alterations to assertions

A fourth element, called a transformation list or t-list, is added to the assertion body structure, which now becomes

label (form, substitute, predicate, t-list).

It is through the t-list element that the transformations, which are to be associated with a particular assertion, are indicated. Note that there are now four possible assertional formats. Firstly, the most general expression as just described. The second possibility is

label (form, substitute, T, t-list).

Here, although no predicate list is desired, because of the existence of the t-list element, a T must be present in the third element position. The third possibility is

label (form, substitute, predicate).

In this case there is no t-list element present and so the process is as described in Chapter II, that is, a left-to-right matching process followed by associated predicate testing. The fourth, and final possibility, is

label (form, substitute).

No predicate or t-list elements are present and so the system employs the left-to-right matching process with implicitly true predicates.

In structure a t-list resembles an a-list, that is, it is a list of dotted pairs. The first element of each pair is a RULE name and the second element or binding is a list of transformation labels. Thus, a typical t-list element might be

((R1.(L1 L2 L3))(R2.(L4 L5 L6)))

where R1 and R2 are the names of RULE's and L1, ..., L6 are the labels of transformations which should be present on the property list of TRFS. It is normal to use LISP's list notation in preference to its dot notation, and so the above t-list would more usually be written as

((R1 L1 L2 L3)(R2 L4 L5 L6)).

In the M-expression notation introduced earlier, such a t-list is described by

r1 [l1;l2;l3] ∧ r2 [l4;l5;l6].

3.4 Operation of the extended matching process

The general strategy which the matching system employs to utilise transformational information in the processing of an assertion is as follows. As before, the left-to-right matching process (described in Chapter II) is used; if the match fails however, an attempt is made to reconstitute the argument subexpression at the level of failure, by applying, from the t-list of the assertion under consideration, those transformations whose labels are associated with the RULE governing† the corresponding subform. If no suitable reconstitution can be found, the process returns to the previous higher structural level and searches for an alternative match using those transforma-

†A RULE with name r is said to "govern" a form f, if f is a list whose first element is r.

tions as yet unattempted at that level. This procedure is continued until either a successful match for the entire form is obtained or until all possible combinations of transformations have been attempted and found to fail. (A more detailed account of the operation of this matching process is presented in the next chapter, and the definitions and actions of the actual functions involved are given in Appendix B.)

It is important to note that the RULE being defined may be referenced in the t-lists of its own assertions. In effect, the form of each assertion is treated as if this top level RULE had been added to the front of it. Thus, an assertion in + which states

a1: + [a;0] → a

may have a t-list + [t1] added to it, where TRFS has a transformation

t1: + [a;b] → + [b;a]

and then not only will (+ x 0) be reduced to x, but also (+ 0 x), where x may be any expression.

With the introduction of transformations, the timing of predicate evaluation becomes more critical. For the left-to-right matching system, the timing is inconsequential (apart from the organisational argument that an attempted match could be concluded earlier, if a predicate evaluation produced a false return), because both the matching process and the predicate evaluation are unique pass or fail tests and, if either fails, the evaluation process continues with the next assertion. However, for the extended matching system, the timing is important since the point of failure determines at which sub-expression level reconstitution attempts are to commence. Thus, before any binding for a form variable is accepted, the predicate element of the assertion is searched for any predicate expression associated with that variable, and, if one is found, it must have value T when evaluated with respect to the proposed binding. To facilitate the searching part of this more dynamic

evaluation of predicates, the format of the predicate element is altered to become a list of dotted pairs (as in an a-list). The first element in each pair is a form variable and the second is a predicate expression. So, a predicate element demanding that A and B should both be numbers, which previously would have been written as

(AND(NP A)(NP B))

is now described by

((A.(NP A))(B.(NP B)))

or, using list notation, by

((A NP A)(B NP B)).

Predicate expressions which reference two or more form variables must be bound to the variable which occurs furthest to the right in the form.

The predicate element will hereafter be referred to as the p-list, and a complete assertion written as

label: name [form] → substitute when p-list with t-list

where name is the name of the RULE being defined, and where the form, substitute, p-list and t-list will be written in M-expression format.

3.5 The generation of equivalent left-to-right assertions

The motivation behind the introduction of transformations is to gain conciseness of description. A property, which is indicated by a TRFS transformation, may thereafter be associated with any assertion of any rule. This capability, when used to describe frequently occurring properties, leads to a considerable reduction in the actual physical size of function definitions. Taking the function linear as an example, all those cases which previously required nine assertions, can now be covered by the single assertion

a1: linear[x;+[*[a;x];b]] → T when free[a;x] ∧ free[b;x]
with +[t1;t2] ∧ *[t4;t5]

where TRFS has among its transformations

t1: +[a;b] → +[b;a]

t2: a → +[a;0]

t4: *[a;b] → *[b;a]

t5: a → *[1;a]

However, one clear disadvantage, incurred by the introduction of transformations, is the resultant reduction in the transparency of function descriptions. For example, it is not obvious that this single assertion definition of linear, along with its associated transformations, does cater for all the possible arguments which can be covered by the previously given nine left-to-right assertions. In other words, whereas the effects of left-to-right assertions are transparently obvious, the same cannot be claimed for transformational assertions, indeed, the ability to predict the behaviour of the latter implies a prior and quite detailed knowledge of the operative characteristics of the extended matching process. Clearly, in these circumstances, if a method exists whereby those left-to-right assertions, which are equivalent to a single assertion with associated transformations, could be generated, then the problem is considerably reduced, and, fortunately, such a method does exist and its processes are relatively straightforward.

The left-to-right matching process may be considered as a special case of the transformational matching process, where each RULE in an assertional form has the identity transformation, I, associated with it. Further, the general strategy of the extended matching process was earlier stated as:- attempt left-to-right match, and, if failure occurs, try to reconstitute the argument expression. Then, remembering that the form of each assertion is treated as if the RULE being defined were added to the front of it, the

RULE's involved in the single assertion definition of linear are linear, + and * and their respective associated transformations are I; I, t1, t2 and I, t4, t5. As each involved RULE appears only once in the form, the number of possible combinations of transformations is nine. Thus, the table:-

	linear	+	*	generated forms	implications
(1)	I	I	I	linear[x;+[*[a;x];b]]	-
(2)	I	I	t4	linear[x;+[*[x;a];b]]	-
(3)	I	I	t5	linear[x;+[x;b]]	a = 1
(4)	I	t1	I	linear[x;+[b;*[a;x]]]	-
(5)	I	t1	t4	linear[x;+[b;*[x;a]]]	-
(6)	I	t1	t5	linear[x;+[b;x]]	a = 1
(7)	I	t2	I	linear[x;*[a;x]]	b = 0
(8)	I	t2	t4	linear[x;*[x;a]]	b = 0
(9)	I	t2	t5	linear[x;x]	a = 1 , b = 0

may be generated. Consider, as an illustration, the generation of the form on line (5). The original or (I,I,I) form is linear[x;+[*[a;x];b]]. The form on line (5) comes from applying the transformations I, t1 and t4 in the reverse direction (that is, match with the substitute and replace by the form) to the subforms governed by linear, + and * in the original form. An I transformation gives no change. The subform governed by + is +[*[a;x];b], and applying t1 in a reverse direction yields +[b;*[a;x]]. Similarly, the subform governed by * is *[a;x] and a reverse application of t4 gives *[x;a]. When these reconstituted forms are combined, the resulting form is linear[x;+[b;*[x;a]]].

If, while performing a backward transformation, a variable disappears from the form, then its last binding is said to be an "implication", for example, on line (3)

$t5^{-1}(*[a;x]) = x$ with implication $a = 1$.

Implications must satisfy all predicate elements associated with the form variable concerned, otherwise the generated assertion is invalid.

The rationale underlying this generation method is fairly obvious. In practice, the matching system attempts to match a form f with some transformation of an expression e, say (e). In effect, this is equivalent to trying to match a form $\mathcal{T}^{-1}(f)$ with e, where $\mathcal{T}^{-1}(f)$ is the result of applying the transformation to f in the reverse direction. Thus, when all possible combinations of transformations are considered in this manner, all the left-to-right assertions which are equivalent to the original transformational assertion should be generated.

In any assertion, a transformation associated with a particular RULE may be utilised by the system in its attempt to match <u>any</u> subform governed by that RULE. This fact must be allowed for by the generation method. For example, consider an assertion in a RULE + which states

a1: $+[*[a;x];*[b;x]] \rightarrow *[+[a;b];x]$ <u>when</u> T <u>with</u> $*[t4;t5]$

where t4 and t5 are defined as before, then the table generated is:-

	+	$*^{(1)}$	$*^{(2)}$	generated forms	implications
(1)	I	I	I	$+[*[a;x];*[b;x]]$	-
(2)	I	I	t4	$+[*[a;x];*[x;b]]$	-
(3)	I	I	t5	$+[*[a;x];x]$	b = 1
(4)	I	t4	I	$+[*[x;a];*[b;x]]$	-
(5)	I	t4	t4	$+[*[x;a];*[x;b]]$	-
(6)	I	t4	t5	$+[*[x;a];x]$	b = 1
(7)	I	t5	I	$+[x;*[b;x]]$	a = 1
(8)	I	t5	t4	$+[x;*[x;b]]$	a = 1
(9)	I	t5	t5	$+[x;x]$	a = 1 , b = 1

where $*^{(1)}$ represents the first or leftmost occurrence of * in the form and $*^{(2)}$ the second occurrence of * in the form.

The inter-related concepts of naturalness, awareness, conciseness and transparency will be further discussed when the author presents his conclusions in the final chapter. Meanwhile, it is enough to say that the provision of this extended matching system affords descriptions which are more concise, but less transparent, than those possible using the simpler left-to-right matching system. Whether or not these more concise descriptions are also more natural would appear to depend upon the context of the evaluation and on the user's own prejudices.

One feature, which has been left undescribed in this account of the extended matching process, is the order in which the possible combinations of transformations are generated. An examination of this order is included in the next chapter where a more detailed description of the operational characteristics of the matching processes is presented.

CHAPTER IV

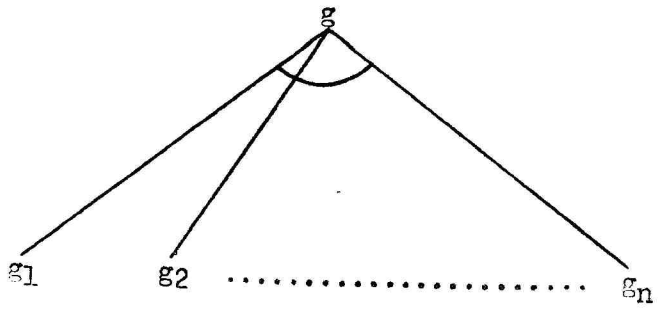
A Description of the Pattern Matching Processes using AND-OR Goal Trees

4.1 Introduction

The descriptions, presented in the previous two chapters, of the pattern matching processes utilised by this programming system have demanded a considerable knowledge of the operation of the LISP interpreter on the part of the reader. In contrast, the account to be given hereafter is detached from any particular ontology and therefore serves two useful purposes. Firstly, the processes may be better understood by readers who do not possess the necessary LISP background. Secondly, the methods of description employed yield a clearer insight into the problems of pattern matching in general, and these processes in particular, allowing for not only a convenient common ground on which existing pattern matching techniques may be compared, but also the highlighting of areas of possible extension for this system's processes. These latter advantages will be further discussed when the author presents his conclusions and suggestions for future development in the final chapter.

In most problem solving systems, the initial goal (the solution of a given problem) generates subgoals, which, in turn, may generate more goals and a certain hierarchy is created. Such a hierarchy is readily represented by a graph or tree growing downwards. The pattern matching processes used in this symbol manipulation system may be described in terms of goal trees, where between any goal and its generated subgoals there exists either an AND or an OR relationship.

AND - an AND relationship between a goal and at least two subgoals exists when the achieving of all the subgoals causes the achieving of the goal. This may be represented diagramatically by



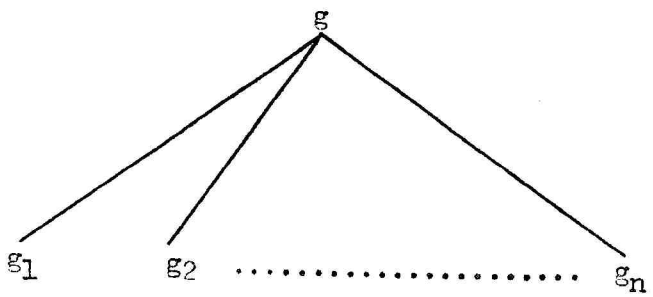
where the arc joining the n branches (n > 2) denotes the AND relationship. An alternative representation is

if $g_1 \wedge g_2 \ldots \wedge g_n$ then g

where this statement should be read as

if g_1 is achieved and g_2 is achieved .... and g_n is achieved
then g is achieved.

OR - an OR relationship between a goal and its subgoals exists when the achieving of any one of the subgoals causes the achieving of the goal. A pictorial representation of this relationship is



or it may be stated as

if $g_1 \vee g_2 \ldots \vee g_n$ then g.

4.2 The left-to-right matching process

Consider the matching algorithm employed by the process. (This algorithm was given earlier in §2.3, and is repeated here merely to ease the problems of referencing.) If a form f is to match an expression e then:

(a) if f is a number, the name of a defined function or the name of a constant, then e must be identical to f,

(b) any atomic form, apart from those described in (a), matches any expression,

(c) if f is a quotation of another from g, then the expression e must be identical to g,

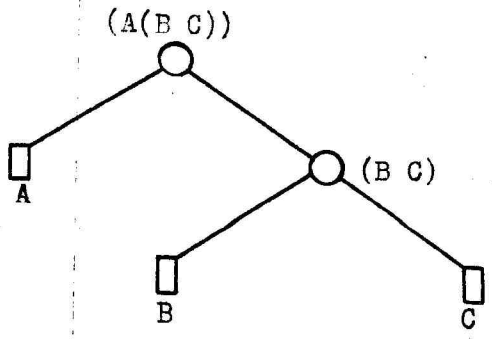
(d) if f is a list of elements $f_1, f_2, \ldots, f_n$, then

(1) e must be a list of elements $e_1, e_2, \ldots, e_n$, and

(2) for $i = 1, 2, \ldots, n$, f_i must match e_i, and

(3) if g is a name which occurs more than once in f, then the corresponding subexpressions in e must be identical.

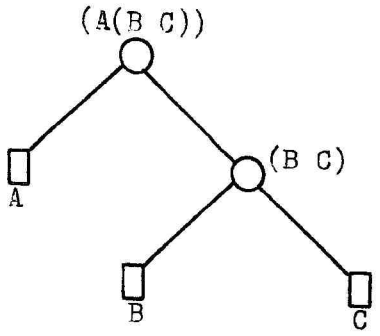
From this description of the algorithm, the form f is seen to be either an atom or a list of elements which may themselves be forms. Clearly then, the structure of a form may be readily represented by a tree. Introducing the symbols O ≡ list or node, and □ ≡ atom or terminator, then the form (A(B C)) may be represented by



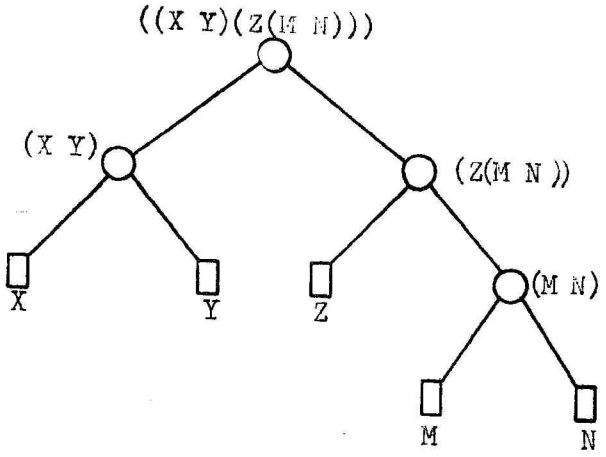
Then, the condition expressed in (d.1) may be restated as:- if f is to match e, then the tree of e must have branches corresponding to all the branches of f's tree, and furthermore, any extra branches occurring in e's tree must stem from nodes corresponding to the terminators of f. The latter part of this statement is necessary because under (b), an atom or terminator in f may match any expression. This statement may be considered as the condition for 'structural' matching. To illustrate how this criterion

operates, consider a form f = (A(B C)) and an expression e = ((X Y)(Z(M N))). The trees are:-

f ≡



and e ≡

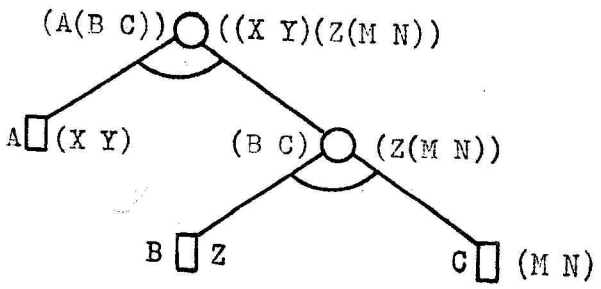


and, by inspection, these clearly satisfy the structural condition.

The condition (d.2) expresses an AND relationship between the goal of achieving a match between f and e, and the subgoals of achieving matches between f_1 and e_1, f_2 and e_2, ..., f_n and e_n. This may be written as:-

$$\underline{\text{if}}\ f_1 \,||\, e_1 \wedge f_2 \,||\, e_2 \ldots \wedge f_n \,||\, e_n\ \underline{\text{then}}\ f \,||\, e$$

where the symbol || is to be read as 'matches'. To illustrate this on a tree, e's tree is superimposed on that of f, and an AND-goal tree is generated:-



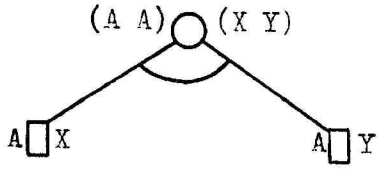
This particular AND-goal tree is interpreted as:-

$$(A(B\ C)) \,||\, ((X\ Y)(Z(M\ N))) = A \,||\, (X\ Y) \wedge (B\ C) \,||\, (Z(M\ N))$$

where $(B\ C) \,||\, (Z(M\ N)) = B \,||\, Z \wedge C \,||\, (M\ N)$.

So far, no mention of the ordering of the matching process has been needed. A truth value T (true) or F (false) might be connected with each terminator subgoal (that is, one with an atomic form) and the result of the match follows from the usual AND relationships between T and F. However, in any practical system, an ordering is necessary, and the one adopted here is from left to right; that is, if f is a list $(f_1, f_2, \ldots, f_n)$ and e is a list $(e_1, e_2, \ldots, e_n)$ then an attempt is made to match f_1 against e_1, if this proves successful then f_2 against e_2 is tried and so on; however, if <u>any</u> failure occurs the match is abandoned.

Now, because of (d.3), which essentially demands uniqueness, some record of the subexpressions of e associated with form terminators must be kept. These records are stored as an association list (a-list) in a dotted pair format. Thus, for the example f = (A(B C)) and e = ((X Y)(Z(M N))), the a-list formed would be ((A.(X Y)) (B. Z) (C.(M N))).† This raises a further complication, exemplified by the problem: match f = (A A) against e = (X Y). The goal tree is:-



or, in the other notation,

$$(A\ A)\ ||\ (X\ Y) \;=\; A\ ||\ X \wedge A\ ||\ Y\ .$$

Because of the ordering of the process, if A || X is T and yields a binding (A.X), then the only conclusion open for A || Y is F. However, there is no reason to suggest that A || X is anymore true than A || Y. Thus, it would appear that the value of A || X is something less definite than T, and this suggestion of truth, rather than a concrete affirmation will be

†N.B. This is clearly not the only reason for creating such an a-list, but even if the matching process was considered as an end in itself, the a-list would still be necessary because of the uniqueness criterion (d.3).

represented by P (possible). It would further appear that the value P depends on the bindings set up during the matching process, and so, in general, the value of a match will be represented by P{J}, where J represents a list of bindings which are considered as implications of the match. The AND relationships are extended to include P in the following way:-

$T \wedge P\{J\} = P\{J\}$,

$F \wedge P\{J\} = F$,

$P\{J\} \wedge P\{K\} = F$ if any binding in J conflicts with any in K ,

$= P\{J + K\}$ otherwise ,

where J + K indicates the union of the two lists J and K. Notice that $T \equiv P\{\ \}$, that is, a match is true if it is possible and there are no implications to be satisfied.

Thus, returning to the example of (A A) || (X Y), now

$$(A\ A)\ ||\ (X\ Y) = A\ ||\ X \wedge A\ ||\ Y$$

$$= P\{((A.X))\} \wedge P\{((A.Y))\}$$

$$= F$$

because of the conflict between the bindings of A, also

$$(A\ B)\ ||\ (X\ Y) = A\ ||\ X \wedge B\ ||\ Y$$

$$= P\{((A.X))\} \wedge P\{((B.Y))\}$$

$$= P\{((A.X)(B.Y))\} .$$

A T value can now only arise from matching identical atoms under condition (a), thus

$$(*\ A\ B)\ ||\ (*\ X\ Y) = *\ ||\ *\ \wedge A\ ||\ X \wedge B\ ||\ Y$$

$$= T \wedge P\{((A.X))\} \wedge P\{((B.Y))\}$$

$$= P\{((A.X)(B.Y))\} .$$

Condition (c) of the algorithm - if f is the quotation of another form g, then e must be identical to g - is the only section left to be considered.

This is represented by:-

$$("g)\ ||\ e\ =\ g \equiv e \wedge g\ ||\ e$$

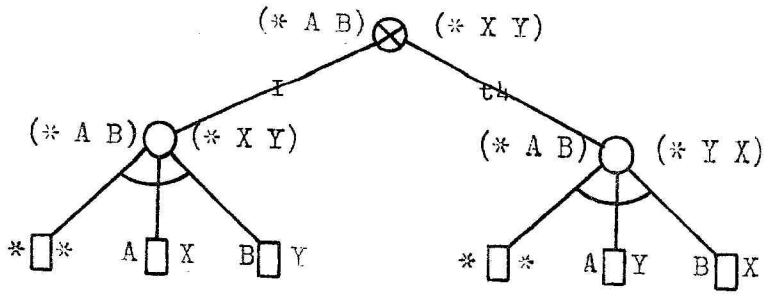
where the $g\ ||\ e$ is still necessary, for consider $f = (A("(A\ B)))$ and $e = (X(A\ B))$, then

$$\begin{aligned}(A("(A\ B)))\,||\,(X(A\ B)) &= A\,||\,X \wedge (A\ B) \equiv (A\ B) \wedge (A\ B)\,||\,(A\ B)\\ &= P\{((A.X))\} \wedge T \wedge P\{((A.A)(B.B))\}\\ &= F\end{aligned}$$

because of the conflicting implications $(A.X)$ and $(A.A)$.

4.3 The transformational matching process

To introduce an extra truth value P may seem unnecessary in the left-to-right matching process, because there, if a match exists it must be unique; but, when transformations are allowed, the value of a match may not be unique, and, in fact, it depends on the ordering of the search for allowable transformations. For example, consider a form $f = (*\ A\ B)$ and an expression $e = (*\ X\ Y)$ with a commutation transformation, $th: *[a;b] \rightarrow *[b;a]$, associated with subforms governed by $*$. Then, introducing a new symbol, $\otimes$ to represent a node whose form has associated transformations, the goal tree of this match is:-



where the symbol $\otimes$ now represents a node without transformations. Here the original goal generates two OR subgoals, the left one corresponding to the

identity transformation I being applied to (* X Y), and the right subgoal corresponds to th being applied to (* X Y). For the I node,

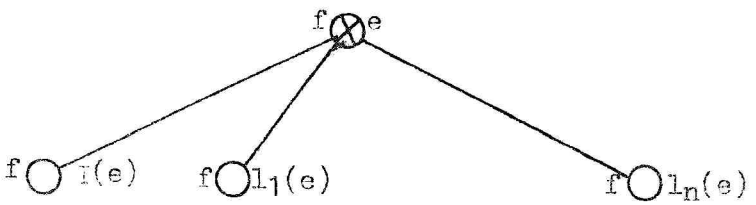
$$(* \ A \ B)||(* \ X \ Y) = *||* \wedge A||X \wedge B||Y$$
$$= P\{((A.X)(B.Y))\} ,$$

while for the th node,

$$(* \ A \ B)||(* \ Y \ X) = *||* \wedge A||Y \wedge B||X$$
$$= P\{((A.Y)(B.X))\} .$$

Both of these results are valid and so values of matches, involving transformations, are not necessarily unique.

To overcome this type of ambiguity, the system imposes a definite order on OR subgoal attempts, and this corresponds to the left to right order in which transformations are supplied by the user, remembering that the identity transformation, I, is implicit and is always considered first. Thus, if e is to be matched against a form f, which is governed by a rule ϕ with an associated transformation list $(l_1, l_2, \ldots, l_n)$, then the goal tree is:-

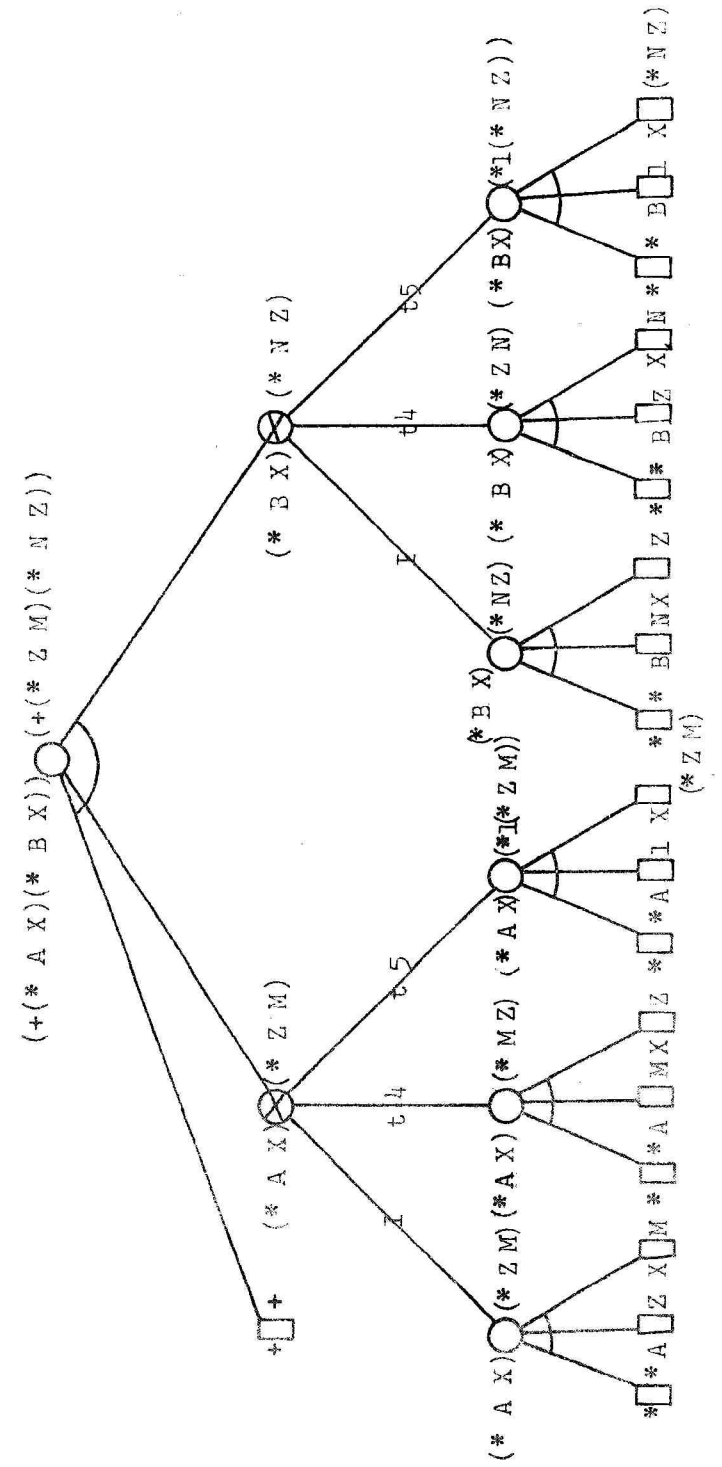


where $l_i(e)$ is the result of applying the transformation whose label is l_i to the expression e; this may also be written as:-

$$f \ \Phi \ e = f||e \vee f||l_1(e) \ \ldots \ \vee f||l_n(e)$$

where the symbol Φ represents matching at a node with associated transformations. The OR subgoal matches are attempted in a left to right order until one which is P is discovered. Thus the returned value for the previous example of (* A B)||(* X Y) would be P{((A.X)(B.Y))} .

One final difficulty, which may arise from the non-uniqueness of a transformational matching process, is that the first obtained P{J} value for a node may preclude any possibility of a non-conflicting P value for subsequent nodes. For example, given f = (+(* A X)(* B X)) and e = (+(* Z M)(* N Z)) where all subforms governed by * have an associated transformation list (t4, t5), where t4: *[a;b] → *[b;a] and t5: a → *[1;a], then the goal tree is:-



and the match proceeds as follows:-

(1) $(+(*\ A\ X)(*\ B\ X))\ ||\ (+(*\ Z\ M)(*\ N\ Z)) = +||+\ \wedge\ (*\ A\ X) \Phi (*\ Z\ M)$

$\wedge\ (*\ B\ X) \Phi (*\ N\ Z)$,

(2) first $\wedge$ subgoal on (1): $+||+ = T$,

and so current value of match is set to T ,

(3) second $\wedge$ subgoal in (1):

$(*\ A\ X) \Phi (*\ Z\ M) = (*\ A\ X)\ ||\ (*\ Z\ M)\ \vee\ (*\ A\ X)\ ||\ (*\ M\ X)$

$\vee\ (*\ A\ X)\ ||\ (*1(*\ Z\ M))$,

(4) first $\vee$ subgoal on (3): $(*\ A\ X)\ ||\ (*\ Z\ M) = P\{((A.Z)(X.M))\}$,

and value of match = $T \wedge P\{((A.Z)(X.M))\} = P\{((A.Z)(X.M))\}$,

(5) third $\wedge$ subgoal on (1):

$(*\ B\ X) \Phi (*\ N\ Z) = (*\ B\ X)\ ||\ (*\ N\ Z)\ \vee\ (*\ B\ X)\ ||\ (*\ Z\ N)$

$\vee\ (*\ B\ X)\ ||\ (*1(*\ N\ Z))$,

(6) first $\vee$ subgoal on (5): $(*\ B\ X)\ ||\ (*\ N\ Z) = P\{((B.N)(X.Z))\}$,

and value of match = $P\{((A.Z)(X.M))\} \wedge P\{((B.N)(X.Z))\} = F$,

because of the conflicting implications for X.

At this point, the matching process needs some indication as to which of these P values to reject. The following rules of operation are adopted:-

(a) all OR subgoals of the current node are considered under the assumption that previous bindings are correct; but, if this fails to produce a satisfactory match, then

(b) control returns to the previous node, and if any further OR subgoals exist, matching restarts at the next one after the last attempted.

Note that when there is a back track, as under (b), the implications must be reset to the state they were in before the previous node was entered. Thus, in this particular example, the process continues:-

(7) second V subgoal on (5): $(* \ B \ X) \,||\, (* \ Z \ N) = P\{((B.Z)(X.N))\}$,

and value of match = $P\{((A.Z)(X.M))\} \wedge P\{((B.Z)(X.N))\} = F$,

(8) third V subgoal on (5): $(* \ B \ X) \,||\, (*1(* \ N \ Z)) = P\{((B.1)(X.(* \ N \ Z)))\}$

and value of match = $P\{((A.Z)(X.M))\} \wedge P\{((B.1)(X.(* \ N \ Z)))\} = F$.

Now because no further V subgoals remain for (5), the process is forced to reconsider the value obtained for (3); two OR subgoals still remain untried at (3), and so the current value of the overall match is reset to T, and the process continues:-

(9) second V subgoal on (3): $(* \ A \ X) \,||\, (* \ M \ X) = P\{((A.M)(X.Z))\}$,

and value of match = $T \wedge P\{((A.M)(X.Z))\} = P\{((A.M)(X.Z))\}$.

Now, the match at (5) can be reconsidered under these new implications,

(10) first V subgoal on (5): $(* \ B \ X) \,||\, (* \ N \ Z) = P\{((B.N)(X.Z))\}$,

and value of match = $P\{((A.M)(X.Z))\} \wedge P\{((B.N)(X.Z))\}$

$= P\{((A.M)(X.Z)(B.N))\}$.

As all three of the $\wedge$ subgoals of the original goal are now satisfied, this value will be returned as the final result of the match.

4.4 Comments

The operational characteristics of these matching processes may be summarised as:-

(a) they are exhaustive, that is, all possible combinations of transformations will be tried,

(b) the nodes and terminators are generated and examined in a left to right order, and

(c) the search for a possible combination of transformations is on a "breadth first" basis, that is, a match is assumed valid for as long as possible, and only when no path remains available, will backtracking take place and then only to the node where the next path opens up.

This search characteristic is analogous to a nested loops operation, where the outermost loop indexes over the associated transformations of the leftmost subform, with the innermost loop corresponding to the rightmost subform. Thus, in terms of the tables of equivalent left-ro-right assertions (described in chapter 3) the last example generates.

order	*	*	*	generated forms	implications
1	I	I	I	(+(* A X)(* B X))	-
2	I	I	t4	(+(* A X)(* X B))	-
3	I	I	t5	(+(* A X)X)	B = 1
4	I	t4	I	(+(* X A)(* B X))	-
5	I	t4	t4	(+(* X A)(* X B))	-
6	I	t4	t5	(+(* X A)X)	B = 1
7	I	t5	I	(+ X(* B X))	A = 1
8	I	t5	t4	(+ X(* X B))	A = 1
9	I	t5	t5	(+ X X)	A = 1 , B = 1

where the order of lines 1 to 9 gives the order in which the combinations will be examined. From this table, the combination (I, t4, I) on line 4 is seen to give the correct equivalent left-to-right form for the last example.

However, the process does not generate all possible combinations blindly and then look for a valid one. In certain cases, it discovers that some of the combinations can be ignored and use is made of this fact. For example, given f = (+(* A X)(* B X)) and e = (+ Y(* 2 Y)), the match would be obtained at line 7 of the table, however, only lines 1 and 4 would previously have been examined and lines 2, 3, 5 and 6 would all have been ignored by the process; that is,

$$(+(* A X)(* B X)) \;||\; (+ Y(* 2 Y)) = +||+ \wedge (* A X) \uplus Y \wedge (* B X) \uplus (* 2 Y),$$

$$+||+ \quad = \quad T \ ,$$

$$(* \ A \ X) \ \ominus \ Y \ = \ (* \ A \ X)||Y \ V \ (* \ A \ X)||NIL \ V \ (* \ A \ X)||(* \ 1 \ Y) \ ,$$

$$(* \ A \ X) \ || \ Y \ = \ F \ ,$$

and so all combinations with second element I cannot possibly succeed, thus, lines 1, 2 and 3 can be ignored,

$$(* \ A \ X)||NIL \ = \ F \ ,$$

where $t4\{Y\} = NIL$, and so all combinations with second element t4 can also be ignored, that is, lines 4, 5 and 6,

$$(* \ A \ X)||(* \ 1 \ Y) \ = \ P\{((A.1)(X.Y))\} \ ,$$

therefore, only combinations with second element t5 need be further considered, that is, only lines 7, 8 and 9, and in fact, the combination on line 7 of (I, t5, I) is correct for this example.

Actually, even lines 1 and 4 are not fully examined; the process stops at the first elements of the second subforms of each of them. It should be noticed that the user exerts considerable control over this searching process, because his given form is taken first and then combinations are generated in an order corresponding to the left to right order in which he supplies the associated transformations.

CHAPTER V

Environment of Evaluation

5.1 Introduction

A programming system may be considered as possessing two main components, a programming language and an evaluation environment. Chapters II, III and IV are concerned primarily with the former, that is, the introduction of a new evaluation procedure based on pattern matching, together with a new function type, designed to extend the descriptive formalisms of LISP to allow for, what are hopefully, more natural descriptions of symbol manipulation algorithms. This chapter deals with the second and equally important component, the environment created by the system in which a user's evaluations are to take place. Most programming systems benefit from a feedback of user's experiences; unfortunately, up to the time of writing this report, the author has been the sole user of this system and so there has been no opportunity to assess user reaction. One consequence of this lack of use, however, is that the ensuing account may be taken as a reflection of the author's view of what a user requires of a programming system designed to manipulate symbols.

Two major design decisions, which were taken at the outset of this project, are responsible, to some extent, for all subsequent investigations and system provisions. Firstly, this system was to provide the means whereby higher level capabilities could be created and amended, and secondly, the system should operate in an on-line environment. The next section gives details of the defining and editing facilities provided to meet the first requirement. The identification and file storage facilities, described in sections 3 and 4 respectively, were designed to increase the usefulness or practicality of the system in its on-line environment, although the same features might also be considered useful in off-line operations. The final section presents interactive facilities which are only meaningful in an

on-line context. These consist of in-built system recovery features and user programmed halts and queries with appropriate restart facilities. An account of the definitions and operations of all functions, mentioned in this chapter, can be found in Appendix A.

5.2 Defining and editing

The definitional capability is based on the function defrules, which takes a single argument, a list of the form

$$((u_1\ v_1)(u_2\ v_2)\ \ldots\ (u_n\ v_n))$$

where each u_i is an atom, intended to be a rule-name, and each corresponding v_i is its S-expression definition. The function's action is to place each v_i on to the property list of its corresponding u_i with indicator RULE. The value returned will be the list

$$(u_1\ u_2\ \ldots\ u_n)$$

that is, the list of the names of the newly defined RULEs.

Having created his RULE definition, which consists of a list of assertions each with its own label, the user is provided with three amendment facilities. The function addrule permits a new assertion to be added to a RULE in any position within that RULE (including before the first existing assertion and after the last existing assertion). Notice that the order in which assertions occur is important. For example, if a RULE / contains an assertion to search for zero denominators, say

a1: /[a;0] → print[INFINITY - DIVIDING BY ZERO]

and the user wishes to include a new assertion to search for zero over zero expressions, say,

a2: /[0;0] → print[UNDEFINED - ZERO OVER ZERO]

then the new assertion, a2, must be inserted before a1, otherwise it will

never be used. One of the disadvantages of Fenichel's FAMOUS,[7], is a lack of this ability to determine the order of his equivalent to assertions; he treats them on a last in, first considered basis. The second editing function is called delrule and this allows for the deletion of any assertion from any RULE. The third and final amendment facility is provided through the function change; this permits the user to alter any of the four elements of a particular assertion, that is, either its form, substitute, p-list or t-list. It also allows the introduction of a new p-list or t-list element where one did not previously exist.

Two display features are now presented, for, although not directly involved in either the definition or amendment of functions, they are often useful in that context. A user, faced with an amendment problem, will often be unable to remember the exact details of the definition he wishes to alter, and so the function display is provided, which, when given a rule-name as its argument, will print out the assertions of that RULE, starting each assertion on a new line. A more specific display facility is also available through the function fetch, which prints out the single assertion indicated by the given arguments.

5.3 Identification bindings

5.3.1 Local identifications

As stated earlier (in §1.3) this system is command orientated. Associated with each command there may be a where list, that is, a list of dotted pairs, where the first element of each pair is an identifying atom and the second element or binding is any expression. Then, before control passes to the main evaluation routine, evalquote, with the issued command, all occurrences of an atom, which has a binding on the where list, are replaced by the corresponding binding. For example, if one wished to find $\frac{d}{dx}\left(\frac{x^2}{f} + f\right)$ where $f = 1 + x^2 \sin x$, then the command

D((+(/(+ X 2)F)F)X) WHERE ((F.(+(+ 1(*(+ X 2)(DEN X)))(/ 1 2))))

might be used. Bindings set up using the where facility are called "local" identifications because they only remain valid for the command cycle in which they are defined.

5.3.2 Global identifications

More permanent bindings may be set up by using the functions ident and identq. Each of these functions takes two arguments, the first being an atom which is to serve as an identifier, and the second being any expression. The function ident places its second argument on to the property list of its first argument under an IDEN indicator. The other function, identq, performs a similar task after it has evaluated his second argument. For example, the effect of the command

IDENT(X (CAR(QUOTE(A B C))))

would be to set the list (CAR(QUOTE(A B C))) on to the property list of X as an IDEN binding, but, if

IDENTQ(X (CAR(QUOTE(A B C))))

were issued, then first (CAR(QUOTE(A B C))) would be evaluated to yield A, and then this value would be set as the IDEN property of X. IDEN bindings are "global" identifications because once created they remain valid until deliberate action is taken to remove or overwrite them.

5.3.3 A pre-processor

To enable not only local but also global bindings to be referenced in a command through their identifiers, a simple pre-processor was implemented. This comprises two new functions evaluate and replace coupled with the introduction of the concept of "marked" functions. A function is marked by setting a certain flag in its property list header. All RULE's are automatically marked during definition by defrules. (To date, only one other

function, a SUBR named subs is marked, and, in fact, subs is a marked subst, the standard LISP 1.5 replacement function, that is, subs and subst are the same block of code referenced from different property lists - compare with evlis and list in standard LISP.)

The M-expression definitions of evaluate and replace are as follows:-

```
evaluate[u;v] =
    [marked[u] → evalquote[u;replace[v;wlist]] ;
        T   →   evalquote[u;v]] .

replace[v;a] =
    [atom[v] → [null[sassoc[v;a;NIL]] →[get[v;IDEN] → car[iden] ;
                                                T → v] ;
                        T → cdr[sassoc[v;a;NIL]]] ;
     eq[car[v];EVAL] → eval[cadr[v];a] ;
     eq[car[v];QUOTE] → cadr[v] ;
     T → prog2 [ { rplaca[v;replace[car[v];a]]
                 { rplacd[v;replace[cdr[v];a]] } ; v ] ]
```

The variable WLIST, introduced in evaluate, represents the where list associated with the command under consideration. If no where clause is present, then WLIST will have been set to NIL.

An input command is now sent to evaluate and not directly to evalquote as in standard LISP 1.5 systems. Control is passed on to evalquote after evaluate has replaced the second part of the command in those cases where the first part is a marked function. At the atomic level, replace searches first for local and then for global bindings, replacing the relevant section of the overall expression by the binding of one is found that is, a new list is not created but the input list is overwritten by the relevant bindings. Notice that in those instances where a referenced identifier has both a where binding and an IDEN property, the former will take precedence because of the search strategy of replace. At the non-atomic level, all expressions, prefixed by

the atom eval, will be replaced by their values. Replace also removes the outermost layer of QUOTE'S. This feature is included to enable references to identifiers to be made without these being replaced by the corresponding identification bindings.

To illustrate the use of this pre-processor, the function identq is re-examined. Although this is not a marked function, it does use evaluate to find the value of its second argument. Thus, the combination of commands

```
IDENT (Y (*(↑ X 3)(COS X)))
IDENTQ(DYDX (D Y X)))
```

would firstly, bind (*(↑ X 3)(COS X)) to Y, and then this binding would replace the occurrence of Y in the second command before the derivative is taken and subsequently bound to DYDX. Notice that if DYDX was replaced by Y in the second command to give

```
IDENTQ(Y (D Y X))
```

the first occurrence of Y would not be replaced by the binding because identq is not a marked function. The effect of this last command would be to over-write the current binding for Y by its derivative with respect to X.

5.3.4 LEE

At the end of every command cycle, the value just obtained is placed on to the property list of a special atom LEE (an acronym for Last Evaluated Expression) under an IDEN indicator. Thus the value of the previous command is always available, through LEE, to the current command. For example, if one wished to find the second derivative with respect to X of an expression currently bound to an identifier Y, then the commands

```
D(Y X)
D(LEE X)
```

would produce the desired result. If one now wished to store this result, giving it an identifier, then the command

IDENTQ(D2Y LEE)

could be used. Notice that the same overall effect could have been achieved by the commands

D(Y X)

IDENTQ(D2Y (D LEE X)).

5.4 Storage management

Most users of languages such as FORTRAN, ALGOL or assembly codes may find it difficult to appreciate the concern of list processing language programmers over the problems of storage management. For the former type of user, if the program with its associated data spaces can be fitted into the available machine space, then the problem is solved. However, the problem is only starting at that stage for list processing systems. All too often, situations arise where more time is being spent on storage reclamations (garbage collections in LISP) than on performing useful work. Less frequently, but still often enough to cause concern, a situation can arise where the available free space is so full of definitions and intermediate results that the system chokes and grinds to a halt. An attempt has been made to alleviate some of the problems involved and descriptions of the provided features follow in the next sub-section.

The remainder of the section is devoted to descriptions of file storage facilities, designed to enable a user to store function definitions and identification bindings on a disk file, from which they may subsequently be restored to core in order that some desired context of evaluation may be created. All user defined properties of any LISP atom may be saved and unsaved by the provided facilities, for example, RULE's, EXPR's and IDEN's.

Perhaps the easiest way to gauge the importance of these facilities is to consider operating without them. Then, the user is faced with the daunting task of defining all the functions and identification bindings, which are required for that session, before any useful work can be attempted. Another useful capability, provided by the introduction of these facilities, is that, definitions, which are not immediately required, can be stored on a disk file, thus freeing, at least temporarily, in-core storage which can be used as valuable working space.

5.4.1 In-core storage

In this programming system, as in most LISP systems, the garbage collector is invoked if, during a call to cons, the LISP cell construction function, it is discovered that the list of free space has been exhausted. The reclamation takes place in two distinct phases, firstly, the marking of all cells which are active, and secondly, the collection of all unmarked cells to form a new list of free space and the clearing of the marks in the active cells. This is an automatic reclamation procedure, that is, without any user action, garbage collection will occur in every command cycle during which the free space is exhausted.

A non-automatic or user initiated reclamation procedure is also provided through the function reclaim. This affords two main advantages. Firstly, it gives the user some control over the timing of the call to the garbage collector. Secondly, in the marking phase of the reclamation, only those cells which make up the new property lists, which have been defined since the session started, need be marked. Compare this with the same stage in the automatic procedure, where not only must new property list cells be marked but also all argument list cells, cells of lists emanating from special registers such as ALIST, WLIST, GOLIST, etc., all lists held on the push-down-stack and in certain temporary storage locations used by the system, then, it is clear that

the user initiated reclamation will be faster than the automatic one and that it will also collect a greater number of cells for the new list of free space.

While the increased control and efficiency of reclaim are attractive, in reality, the function is seldom used directly, because the user does not possess the information about the state of the free space which would be necessary to take advantage of these qualities. However, reclaim is often used in conjunction with another function called setgarb. To understand fully the advantages of this combination of functions, it is necessary to consider the control structure of the overall system, that is, a loop with three components, input, evaluation and output. The automatic reclamation procedure will be invoked (if at all) during the evaluation component of this loop, thus increasing the response time† (often by a factor of two) experienced by the user. It would clearly be more convenient if garbage collection were to occur at the end of the cycle, that is, between the output of the result and the invitation to input the next command, for two reasons. Firstly, at that point in the loop only reclaim and not the full garbage collection need be invoked. Secondly, a slightly longer delay between output and input is less likely to be resented or even noticed by a user when compared to an added delay between input and output, especially as, in the former case, the user will frequently need to study the output value before he can decide on his next input command. Towards these ends, setgarb is introduced. This function expects one argument, an integer, and its action is to place this number in a special register, named CELLMIN. Thereafter, the number of free cells still available is tested after every output, and, if the number has fallen below CELLMIN, then reclaim is invoked, otherwise control passes directly to the input phase. Thus, by a

†response time - delay between issuing a command and the result being displayed.

judicious choice for CELLMIN, a user can replace costly and inconvenient garbage collections with more efficient and less noticeable reclaims.

Both the automatic reclamation procedure and reclaim display the following message during operation

GARBAGE COLLECTION n CELLS

where n is the number of cells collected for the new list of free space. This print-out may be suppressed by use of the function verbos. A further call to verbos will restore the print-out.

Two further facilities, which are based on the functions remove are remprop, are introduced to allow for the removal of unwanted properties from the property lists of atoms. Remove takes one argument, which is a list of atoms, and its action is to overwrite the link from the atom's property list headers to the main bodies of their respective property lists. Remprop expects two arguments, which should be an atom and a property indicator, and its action is to remove the property with the given indicator from the property list of the given atom. It should be noticed that neither of these functions initiate garbage collection, they merely make the indicated space available should reclamation take place.

5.4.2 File storage

The facilities have been designed to permit the use of any number of 10K disk files, each divided into 80 buckets of 128 words. The first two buckets of every file contain a directory of the definitions present and a map of the storage allocation. Every definition stored on a file has an entry in the directory, which consists of its name and the address of the bucket on which the start of its definition body is to be found. The buckets, in which a particular definition is stored, are linked together to form a list which is simulated in the map. For example, a definition identified by the atom

FRED and occupying buckets 3, 16 and 27 would have an entry in the directory of

FRED 3

and in the map, positions 3, 16 and 27 would contain

where  is used to indicate the end of a list.

Before a file can be read from or written to, it must be opened, and this task is performed by the function getfile, which also reads into core the directory and map from buckets 1 and 2 of the indicated file. The name of the currently open file is held in a special register, named CURFILE, and the file name supplied by all the other functions, to be described hereafter, must be identical to that held in CURFILE, otherwise an error message will result. From this last statement, it is implicit that only one file may be open at any given time. If a user wishes to operate with a file other than the currently available one, then he must first close the latter by using the function closefile before the new one can be opened.

The main provisions of these file storage facilities are the ability to save definitions on disk files and that subsequently, that is, either later in the same run or in some future session, these same definitions may be unsaved to create some desired context of evaluation. These tasks are performed by the functions store and restore respectively. While in core, the definitions are held in the form of property lists of atoms, however, on disk, they are stored as character strings. The conversions are performed by the existing LISP input/output routines. Both store and restore take two arguments, a list of atoms and a file name. Store saves the property lists of the atoms in the given list on the indicated file. Restore resets the definition bodies, held on the indicated file and identified by the atoms in the given list, as property lists of the aforesaid atoms.

Once a definition is stored on a disk file, it cannot be overwritten without the user first setting the 'open' flag on the relevant entry of the directory. The function open is provided for this purpose. Its arguments are similar to those for store and restore and its action is to set the 'open' flag on all those entries, identified by the atoms in the given list, in the directory of the indicated file. Thus, before an existing definition can be overwritten, the user must perform a deliberate action and this safeguards valuable definitions against the possibility of being accidentally destroyed.

If certain definitions cease to be of value or if priorities demand that some space be created for more important definitions, then wipeout may be used. This function also expects two arguments similar to those required by store and restore and its task is to remove, from the directory of the indicated file, those entries corresponding to the atoms of the given list, and to add the space so created to the list of free buckets by updating the map appropriately.

It should be observed that the first argument of the four functions store, restore, open and wipeout is a list of atoms and not merely a single identifier. The reason is that, in the author's experience, definitions are handled in groups or packages and not individually. Furthermore, this experience has prompted the introduction of a package name facility, which is based on a new property indicator, PACK. Thus, for example, if the list (+ - * / ↑ #) was the PACK property of an atom OPS, then the command

RESTORE(OPS FILEA)

would produce the same effect as the command

RESTORE((+ - * / ↑ #)FILEA).

PACK properties are set up by using the standard LISP function deflist.

The final feature of these file storage facilities is a display function named dimp. Dimp expects one argument, which should be the name of the currently available file, and it produces a print-out of the directory and map of this file, giving the name of each definition together with the addresses of the buckets which it occupies on the disk.

5.5 Additional interactive facilities

While the facilities for identification bindings and storage management could conceivably be of use in standard off-line LISP systems, the facilities to be described in this section, are only meaningful in an on-line context, because they require user responses to queries or to requests for more information or missing function definitions.

5.5.1 In-built recovery facilities

These features are provided to enable an execution, which is halted because of the absence of necessary function definitions, to be restarted without the need to restate the original problem. Effectively, the provided facilities replace the LISP error diagnostics A2 and A9, which are described in the LISP 1.5 Programmer's Manual, [12], as follows.

A2 Function object has no definition - apply.

This occurs when an atomic symbol, given as the first argument of apply, does not have a definition either on its property list or on the a-list of apply.

A9 Function object has no definition - eval.

Eval expects the first object, on a list to be evaluated, to be an atomic symbol with a definition either on its property list or on the a-list of eval.

An occurrence of either of these failure conditions leaves a standard

LISP system with no option other than the abandonment of the command cycle in which the error is encountered.

In such cases, this system adopts the following sequence of operations. Firstly, a message is displayed which requests a definition for the offending atom. Once this print-out has been given, the system enters what is called a "recovery" phase. This involves safeguarding all the information which will be required by the system if the user decides to restart. Secondly, the requested definition is supplied either by a define or defrules command or possibly from a disk file by a restore instruction. Thirdly, the command cycle is restarted at the point of failure by issuing a restart instruction. The user has the option of abandoning the cycle at the second stage. To do this, a clear command must be given, otherwise the system will remain in recovery mode.

It is important to realise that these features only provide a single level recovery facility, that is, if an error occurs due to the absence of a function definition while the system is still in recovery mode, then although the message signalling the requirement of the definition will be output, the command cycle will automatically be abandoned. However, the original restart option still remains available.

The type of failure, considered so far, usually arises because either a typing error has been made resulting in an atom being mis-spelt, or, perhaps when trying to create a particular context of evaluation, a user may simply forget to provide a needed definition. It was the all too frequent occurrence of this latter cause of failure which led to the list argument and then subsequently to the package name feature being introduced into the file storage facilities described in §5.4.2.

With the introduction of the matching procedure, a further consequence of the absence of function definitions needs to be considered. In the matching algorithm, given in §2.3, it is stated that atoms which are the names of

functions may only match themselves whereas atoms which are free of special properties may match any expression. Thus, the failure to supply a definition to a function name, which is being used as a form variable in a RULE, will almost certainly result in an erroneous binding being produced during the matching process. To combat this possibility, all a-list bindings for atoms, which eval expects to be function objects, are tested in an attempt to decide upon their validity. If these tests are not conclusive, then the system will ask the user to verify or deny the validity of a particular binding. Given a verification, the system will accept the binding and proceed normally, however, given a denial, the a-list bindings arere-examined and definitions are requested for all atoms which do not already possess them and which occur in function object positions. At this point, the system enters a recovery phase and the operations continue as from the second stage which was described previously.

5.5.2 User programmed halts and queries

Leaving aside the thorny problems of round-off errors and tolerance limits, in general, computations involving numbers possess a preciseness which is lacking in manipulations of richer but vaguer symbols. Thus, programming systems designed for the field of numerical mathematics can readily determine whether one given number is equal to, or less than, or greater than another given number and so on. However, the problems confronting the symbol manipulation system are more troublesome, for example, how can it be determined if the symbol A is greater than the symbol B, or if the symbol C is less than the list of symbols (+ D E)? Faced with the necessity of obtaining answers to such questions, the system has little recourse other than to request more information from the user. This programming system provides two facilities, query and external, for this purpose.

The first of these, query, may be used as a predicate in any user defined function - normally in a RULE or an EXPR. It is coded as an FSUBR, whose arguments are evaluated by replace, which was described earlier in §5.3.3 Its action is to display the message QUERY?, followed by the evaluated arguments. The user is then invited to respond by typing either YES or NO. If the response is YES, then the value of the predicate is T (true), if NO then the value is F (false). Any response other than either YES or NO will be rejected and the user requested to ANSWER YES OR NO. Replace is used to evaluate the arguments because, given the current a-list as its second argument, it will replace those atoms in the arguments of query which have bindings on this a-list by those bindings and leave unchanged those atoms without bindings. For example, if the system is evaluating (QUERY IS X > Y ?), with a current a-list which includes the pairs (X.(* 2 A)) and (Y.B), then the result would

QUERY ?

IS (* 2 A) > B ?

where the system is now awaiting the user's reply.

The second function, external, may also be used as a predicate, but is more usually employed in a substitute element in either a RULE or an EXPR. In construction and operation, this function is very similar to query; it is also coded an an FSUBR whose arguments are evaluated through replace, and its action is to display EXTERNAL followed by the evaluated arguments. The user's response, which may be any LISP S-expression, will be taken as the value of external. It is important to realise that, unlike query, there is no in-built check to ensure that the user's reply is a sensible one; therefore, if external is being used as a predicate, the answer given must be either *T* or NIL. As an illustration of the use of external in its normal role in a substitute element, consider the evaluation of (EXTERNAL(+(* A Z)(* B Z)) OR (*(+ A B)Z) ?) with a current a-list of ((A.M)(B.N)(Z.X)), then the result

would be

```
EXTERNAL
(+(* M X)(* N X)) OR (*(+ M N)X) ?
```

where the system is now awaiting the user's response.

In Chapter I, an argument is advanced in favour of an on-line context for algebraic manipulation systems which suggests that, in many instances, a user needs to know the result of his current command before a decision can be taken as to what the next command should be. A direct extension of this concept is to allow a user to guide the actual evaluation process of the system, and this facility is now introduced through a function pause. Again, like query and external, this function is coded as an FSUBR which utilises replace to evaluate its arguments, and, its action is to display PAUSE, followed by the evaluated arguments. However, at this stage the system enters a "pause" mode, which is similar in many aspects to the recovery mode, described in §5.5.1. As with the latter, processing may be restarted by issuing a clear or a restart command. As before, a clear instruction will cause the abandonment of the command cycle, and a restart passes control back to the last entry of evaluate, which normally implies a re-execution of the current command, the only exception being if an identq is involved (see §5.3.3). In addition, a resume instruction is introduced, which operates with one or no argument. If no argument is given, then evaluation resumes at the assertion following the one containing the pause. If one argument is given, then this is taken to be the label of the assertion in the current RULE at which the user wishes to restart.

During the pause, the user may perform any operation he desires, including the deletion of the assertion containing the pause or the following one. However, if the execution of another pause is attempted while still in pause mode, the PAUSE message and the arguments will be displayed, but then PAUSE IN RECOVERY PHASE will be output and the command cycle abandoned. The original

pause still remains available.

To summarise, the functions query, external and pause enable a user to program halts and queries, which, in turn, permit the user and system to interact so that necessary information and guidance may be supplied one to the other.

CHAPTER VI

An Integration Experiment

6.1 Introduction

Perhaps the most frequently unfulfilled ambition of algebraic manipulation systems is the acquisition of an ability to perform symbolic integration. Indeed, to the author's knowledge, there exist only two symbolic integration programs with any valid claim to generality, namely Slagle's SAINT,[22], and Moses' SIN,[15]. (It is interesting to note that both of these programs are written in LISP.) Moses suggested a possible reason for this state of affairs when he wrote ... "the ease with which a symbolic integration program could be written in a proposed language for algebraic manipulation has become an informal test of the power of that language." Thus, in response to the challenge implicit in these remarks, this chapter describes an integration experiment. However, to avoid any future disappointment, it should be understood from the outset that the program, to be presented in some detail hereafter, is not intended as a ready made integration capability but rather represents an investigation into the feasibility of writing such a program in the language developed for this system.

The integration function, called int, expects two arguments, a variable of integration and an integrand. The program is organised in three sections. The first part consists of a simple table structure which is driven by pattern-matching, that is, if the given integrand is found to match an entry on the table, then the value can be given directly. It also performs some basic operations such as treating the elements of sums and differences separately and then combining the results. This section is extensively used by the other two parts of the program. The second section utilises the method of 'derivative-divides', and when coupled to the first section is approximately

equivalent to what Moses calls his first stage of SIN. The third section is based on the EDGE heuristic, and involves guessing the form of the integral and then attempting to obtain values for undetermined coefficients in that form.

6.2 Transformations

The transformations, on the property list of TRFS at the time of this experiment, were as follows:

t1: +[a;b] → +[b;a]

t2: a → +[a;0]

t3: a → +[0;a]

t4: *[a;b] → *[b;a]

t5: a → *[1;a]

t6: a → *[a;1]

t7: a → -[a;0]

t8: a → /[a;1]

t9: a → ↑[a;1]

t10: a → #[#[a]]

t11: ↑[a;n] → ↑[↑[a;eval[quotient[n;2]]]; 2]
when even[n]

t12: *[a;*[b;c]] → *[a;*[c;b]]

In fact, only transformations t1, t4, t6, t9 and t11 are actually utilised by the integration functions. However, the others are included because of their use in simplification, which, as before in the differentiation example described in §2.6.1, is performed by defining the RULE's +, -, *, /, ↑ and # appropriately. Transformations t1 and t4 express the property of commutativity for the operators + and * respectively. Transformations t6 and t9 give, as possible equivalents for any expression, the given expression

times 1 and the expression raised to the power 1 respectively. Transformation t11 was included to aid the search for a particular form of integrand in edge. Its purpose is to reconstitute expressions raised to an even power as the square of their square roots, for example, in §6.5(iii), it is used to give $(x^2)^2$ as an equivalent expression for x^4.

6.3 Table method

The assertions for this first section of int divide into two categories. Firstly, those assertions which do not involve any recursive re-entry into int, that is, if a match between the integrand and one of the forms of these assertions is obtained, then the value of the integral can be given directly. These assertions consist of the following:-

i1: int[x;e] → *[e;x] when free[e;x]

i2: int[x;↑[x;n]] → /[↑[x;+[n;1]];+[n;1]]
when free[n;x] ∧ neg[n;-1] with ↑[t9]

i3: int[x;log[x]] → *[x;-[log[x];1]]

i4: int[x;sin[x]] → #[cos[x]]

i5: int[x;cos[x]] → sin[x]

i6: int[x;tan[x]] → #[log[cos[x]]]

i7: int[x;exp[x]] → exp[x] .

Notice that i2 caters for $\int x\,dx$ by treating it as a special case of $\int x^n\,dx$, that is, through transformation t9, the integrand x is reconstituted as x^1, and hence matches the form of i2 successfully; the value is then given by the value of the substitute, $\frac{x^2}{2}$.

The second group of assertions, in this first section of int, are designed to deal with integrands which are sums or differences etc. They do so by treating each term separately and then combining the results. For the present experiment, only four such assertions are considered.

i8: int[x;+[a;b]] → +[int[x;a];int[x;b]]

i9: int[x;-[a;b]] → -[int[x;a];int[x;b]]

i10: int[x;#[a]] → #[int[x;a]]

i11: int[x;*[a;b]] → *[a;int[x;b]]

when free[a;x] with *[t4] .

Examples using this section are

$$\int x^{3/2}\,dx = \frac{2x^{5/2}}{5}$$

$$\int (e^x + \sin x)dx = e^x - \cos x .$$

Taken by itself, this first section of int can clearly only solve a very limited class of integration problems. However, its main purpose is to provide a basis for the other two sections of int, that is, both of these attempt to reduce the given integrand, by some means, to a form which can be handled by this table method.

6.4 Derivative-divides method

This section searches for integrals which are of the form

$$\int c\ op(f(x))\ f'(x)\ dx$$

where c is a constant, f(x) is an elementary expression in x, f'(x) is its derivative and op is an elementary operator. For this experiment, op must be one of the operators allowed for in the table section, that is, one of log, sin, cos, tan or exp; clearly, however, this set of operators could be easily extended. In addition, three more possibilities for op, involving the exponentiation operator, are catered for, namely, $f(x)^{-1}$, $f(x)^d$ with $d \neq -1$, and $d^{f(x)}$, where d is a constant.

Once it has been established that the integration problem is of the form above, then the solution is obtained by evaluating

$$\int c\ op(y)\ dy$$

and substituting f(x) for y in the result. Four assertions are required.

il2: int[x;*[op[a];b]] → *[dfact;subst[a;x;int[x;op[x]]]]
when opp[op] ∧ mdrv[x;a;b] with *[t4;t6]

il3: int[x;/[b;a]] → *[dfact;log[a]] when mdrv[x;a;b]

il4: int[x;*[↑[a;n];b]] → *[/[dfact;+[n;1]];↑[a;+[n;1]]]
when free[n;x] ∧ neq[n;-1] ∧ mdrv[x;a;b] with *[t4] ∧ ↑[t9]

il5: int[x;*[↑[c;a];b]] → *[/[dfact;log[c]];↑[c;a]]
when free[c;x] ∧ mdrv[x;a;b] with *[t4] .

The predicate function opp, used in il2, determines if its single argument is one of the allowable set of operators. The M-expression definition of the other new predicate function, mdrv, is as follows.

```
mdrv[x;a;b] = [prog[v]
              v: = /[b;drv[a;x]] ;
              free[v;x] → go[L1] ;
              return[F] ;
          L1: csetq[dfact;v] ;
              return[T]] .
```

The purpose of mdrv is to determine if b and $\frac{da}{dx}$ are equivalent except for a constant factor; if this is found to be the case, then the factor is placed on the property list of the atom DFACT as an APVAL and the value T is returned, otherwise the value is F.

With the inclusion of this method, int can now integrate examples like the following:-

$$\int \cos(2x + 3)dx = \tfrac{1}{2} \sin(2x + 3) ,$$

op = cos , f(x) = 2x + 3 , f'(x) = 2 , c = $\frac{1}{2}$.

The identity operator is handled as a special case of $f(x)^d$ with $d = 1$, thus

$$\int \sin x \cos x \, dx = \tfrac{1}{2} \sin^2 x ,$$

$$op = f(x)^d , \; d = 1 , \; f(x) = \sin x , \; f'(x) = \cos x , \; c = 1 .$$

$$\int x \, e^{x^2} \, dx = \tfrac{1}{2} e^{x^2} ,$$

$$op = \exp , \; f(x) = x^2 , \; f'(x) = 2x , \; c = \tfrac{1}{2} .$$

$$\int x \sqrt{1 + x^2} \, dx = \tfrac{1}{3}(1 + x^2)^{3/2} ,$$

$$op = f(x)^d , \; d = \tfrac{1}{2} , \; f(x) = 1 + x^2 , \; f'(x) = 2x , \; c = \tfrac{1}{2} .$$

$$\int \frac{e^x}{1 + e^x} \, dx = \log(1 + e^x) ,$$

$$op = f(x)^{-1} , \; f(x) = 1 + e^x , \; f'(x) = e^x , \; c = 1 .$$

$$\int x \cos x^2 \, e^{\sin x^2} \, dx = \tfrac{1}{2} e^{\sin x^2} ,$$

$$op = \exp , \; f(x) = \sin x^2 , \; f'(x) = 2x \cos x^2 , \; c = \tfrac{1}{2} .$$

6.5 The EDGE heuristic

This heuristic is introduced in Moses' doctoral thesis, Symbolic Integration, [15]. The method is based on the Liouville theory of integration, which shows that if a function is integrable in closed form, then the form of the integral can be deduced up to certain coefficients. No discussion of this theory is to be presented in this thesis, but detailed accounts are given by Ritt, [19], and Risch, [18].

Given an integrand in the form of a product, it is often possible to choose one factor which is outstanding in the sense that it is not contained in the other factors or their derivatives, nor can it be derived from the other factors or their derivatives through rational operations. For example, in $x \, e^x$, the factor e^x is outstanding, while in $x(1 + x^2)^{\frac{1}{2}}$, the outstanding factor is $(1 + x^2)^{\frac{1}{2}}$. However, it is not obvious which, if any, of the factors in $e^x \sin x$ is outstanding, since both are not derivable from one another. In this particular example, the method will lead to a solution

irrespective of which factor is chosen as outstanding, but only after a transposition (see later). Suppose that it is possible to decide on an outstanding factor, and that the integral can be solved in finite terms, then frequently, it is possible to make an educated guess as to the form the integral will take.

As an illustration, a function, called edge, was constructed, which will try guessing procedures if it finds any of the five forms, $e^{g(x)}$, $\log(g(x))$, $\frac{1}{1+g^2(x)}$, $\sin(g(x))$ and $\cos(g(x))$, to be an outstanding factor. Before giving the definition of edge and its linkage to the rest of int, it is instructive to consider the procedures adopted for outstanding factors of the forms above, along with some simple examples. (Most of what follows can be found, together with a more detailed discussion of the merits and demerits of the method, in Moses' thesis, [15].)

(i) $\int h(x)\ e^{g(x)}\ dx$

A good guess for integrals of this form is

$$\int h(x)\ e^{g(x)}\ dx = a(x)\ e^{g(x)} + b(x)$$

where $a(x)$ and $b(x)$ are the undetermined coefficients which are to be found, and where $a(x)$ will not involve $e^{g(x)}$. Differentiate throughout and the equation becomes

$$h(x)\ e^{g(x)} = a(x)\ g'(x)\ e^{g(x)} + a'(x)\ e^{g(x)} + b'(x)\ .$$

The value of $a(x)$ is found by equating the first coefficient of $e^{g(x)}$ on the right hand side with the coefficient of $e^{g(x)}$ on the left hand side. Thus

$$a(x) = \frac{h(x)}{g'(x)}\ .$$

The value of $b(x)$ is then obtained from

$$b(x) = - \int a'(x)\, e^{g(x)}\, dx$$

which, hopefully, will be a simpler problem than the original one.

example $\int x\, e^x\, dx$

$$\begin{aligned} \int x\, e^x\, dx &= a(x)\, e^x + b(x) \\ x\, e^x &= a(x)\, e^x + a'(x)\, e^x + b'(x) \\ a(x) &= x \\ a'(x) &= 1 \\ b(x) &= - \int e^x\, dx \,. \end{aligned}$$

The integral for $b(x)$ is certainly simpler than the original problem; its value may be found by repeating the above guessing procedure, or it will be given by the first section of int. Thus, by either method

$$b(x) = - \int e^x\, dx = - e^x \,,$$

and so

$$\begin{aligned} \int x\, e^x\, dx &= x\, e^x - e^x \\ &= (x - 1)\, e^x \,. \end{aligned}$$

(ii) $\int h(x) \log(g(x))dx$

A good guess for the value of this integral is

$$\int h(x) \log(g(x))\, dx = c \log^2(g(x)) + a(x) \log(g(x)) + b(x)$$

where c is a constant and $a(x)$ does not involve $\log(g(x))$. The $\log^2$ term is necessary (e.g. $\int \frac{1}{x} \log x\, dx$) but its coefficient need only be a constant, otherwise the derivative of the guessed form would contain a $\log^2$ term for which there is no corresponding term in the integrand. Differentiation leads to

$$h(x) \log(g(x)) = 2c \frac{g'(x)}{g(x)} \log(g(x)) + a(x) \frac{g'(x)}{g(x)} + a'(x) \log(g(x)) + b'(x)$$

or, collecting terms in $\log(g(x))$ on the right hand side

$$h(x)\log(g(x)) = \log(g(x))\left[2c\frac{g'(x)}{g(x)} + a'(x)\right] + a(x)\frac{g'(x)}{g(x)} + b'(x) .$$

Dropping the functional characterization of $a(x)$, $b(x)$ etc., and equating coefficients of $\log(g(x))$, the following equation for $a(x)$ is obtained

$$a' = h - 2c\frac{g'}{g}$$

$$a = \int h\,dx - 2c\log g .$$

Now the fact that $a(x)$ is independent of $\log(g(x))$ is used to find the value of c, that is, if $\int h(x)\,dx$ has a term involving $\log(g(x))$ then c is chosen so as to cancel this term, otherwise $c = 0$. The value of $b(x)$ is then found from the relationship

$$b' = -a\frac{g'}{g} .$$

<u>example</u> $\int (x + \frac{1}{x})\log x\,dx$

$$\int (x + \tfrac{1}{x})\log x\,dx = c\log^2 x + a\log x + b$$

$$(x + \tfrac{1}{x})\log x = (\tfrac{2c}{x} + a')\log x + \tfrac{a}{x} + b'$$

$$a = \int (x + \tfrac{1}{x})\,dx - 2c\log x$$

$$= \tfrac{x^2}{2} + \log x - 2c\log x$$

$$2c = 1 \;,\; c = \tfrac{1}{2} \;,\; a = \tfrac{x^2}{2} .$$

$$b' = -\tfrac{a}{x} = -\tfrac{x}{2}$$

$$b = -\tfrac{x^2}{4}$$

and so

$$\int (x + \tfrac{1}{x})\log x\,dx = \tfrac{1}{2}\log^2 x + \tfrac{x^2}{2}\log x - \tfrac{x^2}{4} .$$

(iii) <u>$\int \frac{h(x)}{1 + g^2(x)}\,dx$</u>

If complex constants are ignored, then the $\frac{1}{1 + g^2(x)}$ factor can only originate from two possible sources, $\log(1 + g^2(x))$ and $\arctan(g(x))$. In

either case the coefficients must be constants, otherwise the derivatives would contain terms which are more complex than those found in the integrand. Thus, a good guess for this form of integral is

$$\int \frac{h(x)}{1 + g^2(x)}\,dx = c \log(1 + g^2(x)) + d \arctan(g(x)).$$

Differentiation yields

$$\frac{h}{1 + g^2} = \frac{2cg'g}{1 + g^2} + \frac{dg'}{1 + g^2}$$

$$\frac{h}{g'} = 2cg + d$$

where c and d are constants.

example $\int \frac{x}{1 + x^4}\,dx$

i.e. $h = x$, $g = x^2$ and $g' = 2x$

$$\tfrac{1}{2} = 2cx^2 + d$$

$$d = \tfrac{1}{2} \text{ , } c = 0$$

and so

$$\int \frac{x}{1 + x^4}\,dx = \tfrac{1}{2} \arctan(x^2)$$

(iv) $\int h(x) \sin(g(x))\,dx$

For this form, edge guesses

$$\int h(x) \sin(g(x))\,dx = a(x) \cos(g(x)) + b(x) .$$

Differentiation yields

$$h \sin g = - a g' \sin g + a' \cos g + b' .$$

Equating coefficients of $\sin g$ and the value for a is given by

$$a = - \frac{h}{g'}$$

and b is then obtained from the relationship

$$b = - \int a' \cos g \, dx$$

which, hopefully, will be a simpler problem than the original one.

(v) $\int h(x) \cos(g(x)) \, dx$

The procedure is very similar to that for $\sin(g(x))$ factors.

$$\int h(x) \cos(g(x)) \, dx = a(x) \sin(g(x)) + b(x).$$

Differentiation yields

$$h \cos g = a g' \cos g + a' \sin g + b'$$

$$a = \frac{h}{g'}$$

and $$b = - \int a' \sin g \, dx .$$

example $$\int x^2 \sin x \, dx$$

$$\int x^2 \sin x \, dx = a \cos x + b$$

$$x^2 \sin x = - a \sin x + a' \cos x + b'$$

$$a = - x^2 , \; a' = - 2x$$

$$b = 2 \int x \cos x \, dx .$$

$$\int x \cos x \, dx = a_1 \sin x + b_1$$

$$x \cos x = a_1 \cos x + a_1' \sin x + b_1'$$

$$a_1 = x , \; a_1' = 1$$

$$b_1 = - \int \sin x \, dx$$

$$= \cos x .$$

and so

$$\int x^2 \sin x \, dx = - x^2 \cos x + 2x \sin x + 2 \cos x$$

$$= (2 - x^2) \cos x + 2x \sin x .$$

The M-expression definition of the function edge, created to handle the forms discussed above, is as follows.

```
edge[x;i] =

e1: edge[x;*[h;exp[g]]] → prog[[a];
                a: = /[h;drv[g;x]];
                return[-[*[a;exp[g]];int[x;*[drv[a;x];exp[g]]]]]
                    when T with * [t4]

e2: edge[x;*[h;log[g]]] → prog[[a;b;c];
                c: = int[x;h];
                a: = simp[subst[Q;log[g];c]];
                c: = /[/[-[c;a];log[g]];2];
                b: = int[x;/[*[a;drv[g;x]];g]];
                return[+[*[c;↑[log[g];2]];+[*[a;log[g]];b]]]
                    when T with * [t4]

e3: edge[x;*[h;/[1;+[1;↑[g;2]]]]] → *[dfact;list[ARCTAN;g]]
                    when mdrv[x;g;h] with +[t1] ∧ *[t4] ∧ ↑[t11]

e4: edge[x;*[h;sin[g]]] → prog[[a;b];
                a: = #[/[h;drv[g;x]]];
                b: = int[x;#[*[drv[a;x];cos[g]]]];
                return[+[*[a;cos[g]];b]]]
                    when T with * [t4]

e5: edge[x;*[h;cos[g]]] → prog[[a;b];
                a: = /[h;drv[g;x]];
                b: = int[x;#[*[drv[a;x];sin[g]]]];
                return[+[*[a;sin[g]];b]]]
                    when T with * [t4] .
```

The linkage between the rest of int and edge is managed by adding another assertion to int which states

i16: int[x;i] → edge[x;i] when query[TRY EDGE(x;i) ?] .

Thus edge can only be entered from int if both methods embodied in the first two sections fail and if the user gives the answer YES to the query TRY EDGE(x;i) ?, where x and i will be replaced by the actual arguments. Notice that edge re-enters int and does not call itself directly from any of its own assertions. If one final, default assertion is added to int, stating

last: int[x;i] → list[INT;x;i]

then, the example $\int e^x \sin x \, dx$ which was discussed earlier, can be reconsidered. The methods of the first two sections cannot handle this form and so the user will be asked if he wishes edge to be tried with $\int e^x \sin x \, dx$. If he answers YES, then one of the subproblems generated will be $\int e^x \cos x \, dx$, and again the first two sections of int cannot cope, and so the user will again be queried as to whether he wishes edge to be tried. If once more he answers YES, then one of the subproblems generated this time will be $\int e^x \sin x \, dx$ and for the third time the user will be asked to decide whether or not edge should be used. However, this time he should answer NO, realising that the method has generated the original problem and so the final value will be given as

$$\int e^x \sin x \, dx = e^x(\sin x - \cos x) - \int e^x \sin x \, dx$$

and the answer may then be obtained by transposition,

$$\int e^x \sin x \, dx = \tfrac{1}{2} e^x(\sin x - \cos x).$$

Appendix D contains listings of the function definitions and examples given in this text.

CHAPTER VII

Conclusions and Possible Developments

7.1 Conclusions

At the outset of this report, it was stated that this thesis is concerned with an investigation into the techniques of computer aided manipulation of symbols. The overall objective of this study is to develop a general purpose, on-line symbol manipulation system incorporating features which enable the user to modify and extend the manipulative capabilities of the basic processor. These requirements can be met only by the choice of a low level language (LISP 1.5 was a convenient starting choice in this instance), in which the user can apply the primitive facilities provided, to the construction of the particular capability required.

However, it was felt that the descriptions of many desirable, higher level capabilities (such as differentiation, integration and simplification), afforded by existing symbol manipulation systems, were so complex that the tasks of modification and extension were made impractical if not impossible, especially in an interactive environment. Thus, a search for a more natural mode of description was undertaken along with an examination of the environment of evaluation, with the aim of developing a programming system which would be modular, practical, interactive and helpful.

7.1.1 Language

The conditional expression forms the basis of LISP's descriptive power, and, when this formalism is coupled with the use of recursive references to the function being defined, the resulting definitional capability is both elegant and powerful. The quest for increased naturalness involved extensions to both the semantics and syntax of standard LISP. The former was effected by the embedding of a pattern matching process as a basic evaluation procedure of

the LISP interpreter. The alteration to syntax is founded on an extended conditional expression formalism and involved the creation of a new type of function, called a RULE.

In Chapter II, a simple matching process was introduced and its usefulness illustrated by two sample programs, the Wang Algorithm for the propositional calculus and differentiation with allied simplification. Both examples exhibit a considerable gain in both transparency and conciseness over the corresponding programs written in LISP 1.5. It is particularly instructive to compare Wang's formulation of his algorithm with the LISP 1.5 coding presented in the Programmer's Manual, [12], and the program given in §2.6.2. While the transparency of Wang's description is lost in the complexity of the LISP 1.5 code, the transcription into the new RULE type functions is straightforward, each Wang elimination rule being described directly by a single assertion. Likewise, the descriptions of the differentiation and simplification RULE's of §2.6.1 bear a striking resemblance to normal mathematical notation (leaving aside the Polish prefix notation).

However, the construction of the simplification RULE's brought to light certain inadequacies of the left-to-right matching process, especially with regard to the handling of algebraic operators. For example, in §2.6.1, to define that the product of any expression and unity is equal to the expression itself, two assertions were required, namely

$$a1: \quad *[a;1] \rightarrow a$$

and

$$a5: \quad *[1;a] \rightarrow a$$

It seems that these shortcomings stem from an expectation, on the part of the user, that either the system should possess an in-built knowledge of certain properties of the algebraic operators (such as the commutativity of + and *), or else some mechanism should be available whereby such properties can be

communicated to the system. To avoid constraining assumptions, the latter course was adopted. New entities, called transformations, were introduced to allow for the expression of these properties, which could then be communicated to the system by associating the appropriate transformations with a particular assertion. These innovations achieve a further gain in conciseness, which is exemplified by a comparison of the two definitions of linear, presented in Chapter III, and by the simplification RULE's developed for the integration experiment and given in Appendix D with the corresponding RULE's constructed in §2.6.1.

Throughout this project, the concept of minimum assumption and maximum flexibility has been consistently adhered to. Thus, the matching processes developed provide a user controlled flexibility, ranging from the basic pair operation to the more sophisticated transformational procedure. For example, the pairing operation performed during the evaluation of any EXPR

$$\lambda\ [[x_1;\ \ldots;\ x_n];e]\ ,$$

is simulated by match, when the above definition is transcribed into the single assertion RULE

$$\text{label:}\quad [x_1;\ \ldots;\ x_n] \rightarrow e$$

where e is the definition body, which may take the form of either a composition or a conditional expression. A comparison, on the grounds of naturalness, of the respective merits of the left-to-right and transformational matching processes presents some difficulties. While the latter clearly demands a greater awareness of the operational characteristics of the matching process, and so, in that sense, must be considered less natural than the former, it also presents, in many cases, a more convenient and less demanding vehicle for the description of desired capabilities.

7.1.2 Environment

The investigations of this component were directed towards the development of a programming system possessing the characteristics of modularity and extendability, practicality in an on-line environment, and interactivity and helpfulness.

Modularity and extendability were achieved by the provision of a relatively simple set of defining and editing functions. The problem of amendment is eased by the association of a label with each assertion and by the extra transparency afforded by the introduction of the new RULE type function. The experiment, reported in §2.6.1, where RULE's to perform differentiation and simplification were systematically built up, illustrates the usefulness of these facilities. Clearly, the functions int and edge created for the integration experiment in Chapter VI, could also be readily extended by the use of these facilities.

The practicality of a programming system is closely related to the user's ability to create and control the context of evaluation. In this area, an effective referencing capability was developed consisting of local and global identification binding facilities together with a simple pre-processor. No attempt was made to create a nested referencing capability, since it is the author's opinion that such a capability is incongruous in a command orientated, on-line environment. The most important single feature, which enhances the usefulness of this system in its on-line environment, is the provision of file storage facilities whereby the user may save and recreate any particular context of evaluation. In addition, the handling of in-core storage was examined and facilities which allow for a more efficient management of working space were developed.

The final area of investigation led to the development of features which make use of the possibility for interaction provided by the system's on-line

environment. To this end, certain standard LISP diagnostics were replaced by more meaningful messages, and recovery facilities were introduced which allow this system to be reinstated in circumstances where a standard LISP interpreter abandons the execution. Furthermore, the introduction of programmed halts and queries permits genuine two-way interaction between the user and the machine, enabling both guidance and information to be transmitted one to the other.

The overall objectives of the developments arising from these investigations were influenced by the opinion, expressed by Weizenbaum, [25], as follows:- "It appears that the best way to use a truly interactive man-machine facility is not as a device for rapidly debugging a code representing a fully thought solution to a problem, but rather as an aid for the exploration of problem solving techniques." The combination of the interactive features with the defining and editing facilities provided by this system go some way towards the fulfilment of these objectives.

7.1.3 Efficiency

The optimisation of efficiency, in any problem solving situation, is concerned with the most economical utilisation of the abilities of man and the available machine. Some authors have suggested that this task of optimisation involves a conflict of interest between the man and the machine, that is, between ease of axiomization and efficiency of operation.

The primary concern, with respect to man, is the availability of a natural language in which to generate programs - this aspect has formed the major part of this investigation. With regard to the machine, efficiency involves the twin aspects of space and time. Because RULE type definitions are more concise than the corresponding LISP 1.5 definitions, the definitional space required is considerably reduced; however, during the matching

processes, the production and rejection of many trial a-lists uses up the working space at a more rapid rate. Because the on-line, operational environment, provided by the Queen's University of Belfast Computing Laboratory,† restricts overall program size to around 15K, which, in effect, limits the combined definitional and working space for this system to 3,000, 2-word cells, the former advantage by far outweighs the latter disadvantage. For example, the integration experiment, reported in Chapter VI and Appendix D, required a definitional space of less than 2,000 cells, whereas, the author believes that the corresponding conventional LISP definitions would need a space of at least 15,000 cells. It is much more difficult to come to any definite conclusions with regard to time efficiency because of the lack of information on which to make comparisons. In fact, the only programs, which are available to allow such an assessment (and which fit into the restricted space of the system) are the LISP 1.5 and RULE type definitions of the Wang Algorithm. Despite more frequent invocations of the garbage collector, the evaluation of the latter is slightly faster than the former; if time spent in storage reclamations were disregarded, a considerable gain in speed would result.

The basic objective of this project - the development of a modular and interactive programming system - necessarily implies that its primary concern should be with ease of axiomization rather than efficiency of machine operation. However, despite this emphasis, a significant gain in machine efficiency can also be claimed.

7.2 Possible developments

This section contains a discussion, of a rather speculative nature, of possible developments of this system.

---

†The on-line operational environment is provided by the Multi-plexed Console System developed at Q.U.B. by J. Warne and J. Hall - no publications.

7.2.1 Language

With regard to language, two possible areas of development are considered: the first is concerned with the assignment of types to form variables in RULE definitions, and the second with search strategies in the matching process.

At present, each constraint on a form variable in an assertion must be expressed in the predicate element associated with that particular assertion. This leads to the repeated use of simple constraints such as np[n], atom[x] and fixp[i]. Such repetition could be eliminated by the introduction of a declaration mechanism for assigning types to variables (analogous to that adopted in CONVERT, [8]). For example, a declaration list of the form

(NP(M N) , ATOM(X Y) , FIXP(I J K))

might be associated with a RULE, and thereafter each occurrence of the variables m or n, x or y, and i, j or k, in the form of any assertion of that RULE, will have associated with it an implicit constraint np, atom and fixp respectively. Notice that, since any predicate (including user defined ones) may be used to indicate a type, the allowable set of types would be unlimited.

The matching processes, described in this thesis, provide convenient tools for the analysis of tree structures. In Chapter IV, a suitable notation for the description of this analysis, is introduced. Other systems, which employ similar techniques and where this notation might be useful, are SAINT, [22], SIN, [15], and the Logic Theory Machine, [17]. In SAINT, the attempted integration procedure is also described in terms of AND-OR goal trees. Here, however, by using a heuristic decision procedure to consider all OR subgoals of the current goal, together with all previously generated and as yet unattempted OR subgoals, an effort is made to determine the most probable path to success. In SIN, the last generated node is selected as the one to be examined next; this strategy is referred to as "depth first."

The effect is to force the examination of a single path until either a solution or an impasse is reached. The Logic Theory Machine employs a "breadth first" strategy, similar to that used in this project.

One possible area of investigation, which might prove profitable, would be an examination of the use of heuristics (in a similar fashion to that employed by Slagle in SAINT) to increase the efficiency of the matching processes. Because the application of each assertion can be regarded as an OR subgoal of the total goal of applying a RULE to a given set of arguments, it seems that a logical extension of this investigation would be to consider the use of similar techniques to order the application of assertions according to their probability of success, that is, the one deemed most probable being tried first, the least probable being attempted last. Such developments would involve a considerable reorganisation within the existing system.

Another worthwhile objective would be the development of a method, whereby the number of assertions which need to be considered in any RULE evaluation, might be reduced. Currently, the list of assertions of a given RULE is searched serially; this procedure becomes increasingly prohibitive as the number of assertions grows larger. This situation would be alleviated if the assertions could be grouped, according to certain properties, and a method found whereby the group (or groups) which are relevant to a particular argument expression may be derived. A possible starting point for such an inquiry might be the hash coding scheme, developed by Martin, [14], for the examination of the equivalence of algebraic expressions.

7.2.2 Environment

This section discusses two possible developments with regard to the environment of evaluation; the first is concerned with extending the existing identification and referencing facilities, and the second with improving the user interface.

The present identification facilities allow a user to assign an atomic identifier to any expression, while the facilities afforded by the pre-processor, permit such an expression to be referenced by means of its identifier. Two extensions are suggested. Firstly, a "functional" identification facility could be introduced whereby a user would be able to issue commands such as

$$\frac{d}{dx}\left(\frac{g(x^2,\ 3)}{f(2x)}\right) \quad \underline{\text{where}} \quad g(x,y) = xy\ \sin(xy) \quad \text{and} \quad f(u) = 1 - u\ \cos\ u\ .$$

Secondly, a facility allowing for the communication of external constraints could be provided; this would permit the evaluation of commands such as

$$\int \frac{dr}{r\sqrt{2hr^2 - a^2}} \quad \underline{\text{where}} \quad h > 0\ .$$

As with the existing referencing facilities, these new features would be best implemented by extending the current pre-processor rather than by altering the standard LISP interpreter.

A major drawback of the present user interface is the use of polish prefix notation in LISP S-expressions. Clearly, the system would benefit from a more suitable input/output language. To be consistent with the approach so far adopted in this project, a facility should be provided whereby the user can define a notation which is particularly appropriate to the range of problems on which he is engaged. Preliminary investigations with a pre-determined language indicate that the basis for such a facility could be provided by a relatively simple top-down recognizer.

7.3 Epilogue

While the author recognizes the subjective qualities of naturalness and awareness, he believes that the programming system, developed for this project, provides a convenient vehicle for the creation, amendment and

utilisation of many desirable, higher-level capabilities. The inclusion of a wide range of specific, higher-level capabilities within the present system, would lead to a programming system, which would go some way towards the resolution of the conflict between the lower and higher-level approaches, discussed in the introductory chapter.

APPENDIX A

Functions of the Evaluation Environment

A.1 Defining and editing functions

defrules [x] : SUBR pseudo-function

The arguments of defrules, x, is a list of pairs

$$((u_1\ v_1)(u_2\ v_2)\ \ldots\ (u_n\ v_n))$$

where each u is a name and each v is an S-expression definition for a RULE. For each pair, defrules puts a RULE indicator on the property list of u pointing to v. The new property is added to the front of the existing property list. The value returned is the list of u's.

n.b. defrules [x] = deflist [x;RULE] .

addrule [name; label; newassert] : SUBR pseudo-function

The three arguments of addrule are the name of a RULE, a label within that RULE (or NIL), and a new assertion which is to added to the RULE. The action of the function is to insert the new assertion into the named RULE immediately in front of the assertion with the indicated label. If the label argument is NIL, then the new assertion is added after the last existing assertion. No value is returned.

delrule [name; label] : SUBR pseudo-function

This pseudo-function deletes the assertion, whose label is given as the second argument, from the RULE whose name is the first argument. Again no value is returned.

change [name; label; type; newelement] : SUBR pseudo-function.

This routine changes the type element of the labelled assertion of the

named RULE to be the new element. The altered assertion is displayed as the value. Change is perhaps a misnomer, as in some circumstances this function will insert a new element where one did not previously exist. The type, given by the third argument, must be one of four possible atoms (otherwise an error message will be output):-

(i) FORM - change the form of an assertion

(ii) SUBS - change the substitute of an assertion

(iii) PRED - either change an existing p-list or insert a new one

(iv) TRFS - either change an existing t-list or insert a new one or if neither a p-list or a t-list is currently in existence, then insert a T predicate prior to the insertion of the new t-list.

fetch [name; label] : SUBR

The definition of the RULE, named by the first argument, is searched for an assertion whose label is the second argument. If such a label is found, the value of fetch is the assertion definition (i.e. (form, subs, p-list, t-list)), otherwise the value returned is NIL.

display [name] : SUBR pseudo-function

The execution of display causes the assertions of the RULE, named by the argument, to be printed out with each assertion starting on a new line.

A.2 Identification functions

ident [ob; val] : SUBR pseudo-function

This function is used to create an IDEN binding on the property list of an atomic symbol. The first argument, ob, should be an atomic symbol; the second argument, val, which is returned as the value, is the indicated IDEN property.

identq [ob; val] : FSUBR pseudo-function

The action of identq is similar to that of ident except that the second argument is evaluated before the IDEN property is set up.

```
identq [ob; val] =
        [atom [val] → [get [val;IDEN] → ident [ob;car [iden]];
                       T → ident [ob;eval [val;NIL]]];
         T → ident [ob;evaluate [car [val]; cdr [val]]]].
```

where [x] : SUBR pseudo-function

The argument of where, x, is a list of dotted pairs

$$((u_1 \cdot v_1)(u_2 \cdot v_2) \ \ldots \ (u_n \cdot v_n))$$

where each u is an atomic identifier and each v is any S-expression. Its action is to place the address of this list into a reserved location called WLIST. (For details of the use of this function, see Appendix C.)

A.3 Storage management functions

A.3.1 In-core

reclaim [ ] : SUBR pseudo-function

The execution of this function causes garbage collection to occur. No value is returned, but a message giving the number of cells collected for the new list of free space is output.

setgarb [n] : SUBR pseudo-function

This function expects its argument, n, to be an integer. Its action is to store this number in a reserved location, called CELIMIN, and to set a system flag, called GARBIND. (For details of the use of this function, see Appendix C.)

verbos [ ] : SUBR pseudo-function

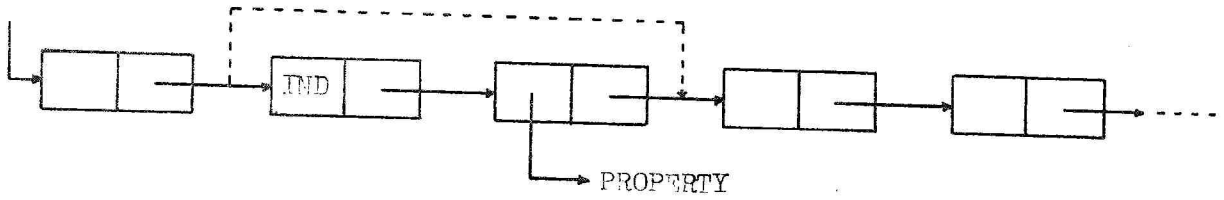
The action of this function is to suppress the garbage collection print-out. A further call to verbos will reinstate the print-out. No value is returned.

remove [x] : SUBR pseudo-function

The argument, x, should be a list of atomic symbols. The execution of remove causes all the properties of all the atomic symbols in x to be removed, with the exception of their print-names. Value is x.

remprop [x; ind] : SUBR pseudo-function

This pseudo-function searches the property list of x (which should be an atomic symbol), looking for all occurrences of the indicator ind. When such an indicator is found, its name and the succeeding property are removed from the list. The two "ends" of the list are rejoined as shown below. No value is returned.



A.3.2 Disk files

getfile [name] : SUBR pseudo-function

getfile expects its argument to be the name of a disk file. Its action is to place this name in a reserved location called CURFILE, and then to open the indicated file and read in the directory and map held on buckets 1 and 2.

n.b. The file-name supplied to all the succeeding functions, described in this section, must be identical to that held in CURFILE, otherwise an error message will result.

closefile [name] : SUBR pseudo-function

This function closes the indicated file.

store [x; name] : SUBR pseudo-function

The argument, x, should be a list of atomic symbols. This function causes the property list of each of these atomic symbols to be written on to the indicated file. However, if it is found that any of these symbols is already present in the directory, then the existing definition cannot be overwritten unless the 'open' flag is set. In the case of a conflict with the open flag not set, the user is informed, by a suitable message, that his definition has not been stored. New entries on the directory and map are created for previously unencountered symbols. The conversion from the internal list structures of the property lists to the character strings held on the file is performed by the standard LISP output routines.

open [x; name] : SUBR pseudo-function

The argument, x, should be a list of atomic symbols. The action of this function is to set the open flag on all those entries in the directory corresponding to the atomic symbols given in x.

restore [x; name] : SUBR pseudo-function

Again, x should be a list of atomic symbols. The definitions associated with these atomic symbols are read in from the named file and set up as property lists of the aforesaid atomic symbols. The conversion from character string to list structure is performed by the existing LISP input routines.

wipeout [x; name] : SUBR pseudo-function

Once more, x should be a list of atomic symbols. The effect of this function is to remove from the directory of the named file, those entries

relating to these atomic symbols. The space on the file, which is thus freed, is added to the list of available buckets by updating the map appropriately.

n.b. It should be observed that the functions store, open, restore and wipeout expect as a first argument a list of atomic symbols. An alternative is that this argument may be a single atomic symbol, providing this symbol possesses a PACK property. In this case, the value indicated by the PACK replaces the supplied first argument and the processes then proceed as described previously.

dimp [name]

: SUBR pseudo-function

This function causes a print-out of the directory and map of the named file, giving the names of the definitions present and the buckets which each occupies on the disk file.

A.4 Interactive functions

The functions, to be described in this section, are all FSUBR's which utilise replace (see Appendix C) for the evaluation of their arguments.

query [s]

: FSUBR predicate

The action of this function is to display QUERY?, followed by the value of the argument s. The user is then invited to respond. If he replies by typing YES, then query returns the value *T*; if the reply is NO, the value returned is NIL. Any response other than YES or NO will cause ANSWER YES OR NO to be displayed, followed by a further invitation to respond.

external [s]

: FSUBR

The action of this function is to display EXTERNAL, followed by the value of the argument s. The user is then invited to reply, and his response will be taken as the value of external.

pause [s] : FSUBR pseudo-function

The action of this function is to display PAUSE, followed by the value of the argument s. The system then enters a "pause" mode, which can only be terminated by the execution of one of the functions restart, clear or resume (to be described in the next section). During the pause, the user may perform any operation he desires, including the creation of new definitions and the amendment of existing ones, even to the extent of deleting the assertion in which the pause occurred. However, if the execution of another pause is attempted while still in pause mode, the display of PAUSE and the evaluated argument will be given as before, but then a message PAUSE IN RECOVERY PHASE will be output and this second pause mode automatically terminated. The original pause is still valid.

A.5 Recovery functions

restart [ ] : SUBR pseudo-function

This function resets the system to the state that it was in before the recovery mode was entered, and then recommences the execution from that point. Recovery mode is entered either through pause or because an error condition has been encountered which has an associated in-built system halt and query (see §5.5.1).

clear [ ] : SUBR pseudo-function

The action of this function is merely to terminate the current recovery mode.

resume [ ] or resume [label] : SUBR pseudo-function

This function may only be used when the system is in pause mode. If no argument is supplied, then execution recommences at the assertion

following the one which contained the pause. If an argument is supplied, then this is taken to be the label of the assertion within the RULE, which contained the pause, at which execution is to be resumed.

APPENDIX B

Functions of the Matching Processes

In this appendix, the functions of the transformational matching processes are described in a mixture of M-expressions and English. (Remember that the left-to-right matching process, which is fully described in Chapter II, is a special case of the general transformational processes where each governing atomic symbol has only the identity transformation, I, associated with it.) The M-expression definitions should not be taken too literally. The functions involved have been coded in the ICL assembly language, PLAN, and in many cases where a recursion is suggested by the definitions, the actual program is a store and transfer. The functions to be described are match, m1, m2, m3 and trans.

```
match [asslist;args] = [prog[v]
      v: = cdr[asslist];
      args: = cons[currule;args];
L0:   alist: = NIL;
      plist: = NIL;
      tlist: = NIL;
      null[cddar[v]] → go[L2];
      eq[caddar[v];T] → go[L1];
      plist: = caddar[v];
L1:   null[cdddar[v]] → go[L2];
      tlist: = cadddar[v];
L2:   m1[cons[currule;caar[v]];args] → return[cadar[v]];
      null[cdr[v]] → go[L3];
      v: = cddr[v];
      go[L0];
```

```
L3:  print[IIØ MATCH FØR];
     print[args];
     return[NIL]].
```

The first argument of match, asslist, should be a list of assertions of the form

$$(\text{label}_1(f_1, s_1, p_1, t_1) \ldots \text{label}_n(f_n, s_n, p_n, t_n))$$

and the second argument, args, should be the argument expression to which these assertions are to be applied. In addition, match utilises four reserved locations CURRULE, ALIST, PLIST and TLIST. The last two are employed for the communication of the p-list and t-list of a particular assertion between the matching functions themselves and are not meaningful to any other part of the system. CURRULE holds the name of the current RULE under consideration and is set up in apply or eval whenever a RULE is encountered. The other register, ALIST, is used not only for communication between the matching functions, but also to transmit the created a-list back to apply in the event of a successful match being found. (See Appendix C for further details about CURRULE and ALIST.)

```
m1 [f;e] =
   atom[f] → m2[f;e];
   atom[car[f]] ∧ sassoc[car[f];tlist;NIL]
                  → m3[f;e; assoc[car[f];tlist]];
   T → m2[f;e]].
```

m1 is merely a switching function, which is used to decide whether the match should continue through m2 or m3. If the form f has, as its governing car, an atomic symbol with an associated transformation list in the TLIST element, then the match proceeds through m3, otherwise it continues through m2.

```
m2 [f;e] = [
 null[f] → null[e];
 atom[f] → [np[f] V funp[f] → eq[f;e];
           null[sassoc[f;alist;NIL]]
                  → [null[sassoc[f;plist;NIL]]
                        V eval[assoc[f;plist];cons[cons[f;e];alist]]
                              → prog2[alist:=cons[cons[f;e];alist];T];
                     T → NIL];
           T → equal[cdr[assoc[f;alist]];e]];
 eq[car[f];QUOTE] → [equal[cadr[f];e] → m2[cadr[f];e];
                     T → NIL];
 T → m1[car[f];car[e]] ∧ m2[cdr[f];cdr[e]]].
```

This function is a direct implementation of the matching algorithm, described in §2.3, with two additions. These involve, firstly, the creation of a-list bindings, and secondly, the search for and subsequent testing of any predicate expressions in the PLIST element which are associated with the form variable being matched.

```
m3 [f;e;trfs] = [prog[v]
   v: = e;
   trfs: = cdr[trfs];
L1:m2[f;v] → return[T];
L2:null[trfs] → return[F];
   v: = trans[e;car[trfs]];
   trfs: = cdr[trfs];
   null[v] → go[L2];
   go[L1] .
```

The third argument of m3, trfs, should be a list of the form

(name L1 L2 ... Ln)

that is, the first element is the atomic symbol which governs the form f and the other elements L1, ..., Ln are the labels of the transformations which may be utilised in the attempt to find a suitable match. This function uses m2 to perform the matching trials, and trans to reconstitute the argument expression according to the supplied transformation list.

```
trans [e;label] = [prog [v]
      v: = fetch [TRFS;label];
      alist: = NIL;
      plist: = NIL;
      tlist: = NIL;
      null [cddr [v]] → go [L2];
      eq [caddr [v];T] → go [L1];
      plist: = caddr [v];
L1:   null [cdddr [v]] → go [L2];
      tlist: = cadddr [v];
L2:   m2 [car [v];e] → return [replace [copy [cadr [v]];alist]];
      return [NIL]].
```

The arguments of trans are an expression, e, and a label of a transformation (which should exist on the property list of the special atom TRFS). The function tries to reconstitute the expression e according to the indicated transformation. The function m2 is used in an attempt to match the form of the transformation with e, and replace is used to evaluate the substitute element in the event of a successful match.

APPENDIX C

The Modified LISP Interpreter

This appendix describes the operational characteristics of the programming system and presents the definitions of those functions which are involved in the modified LISP interpreter.

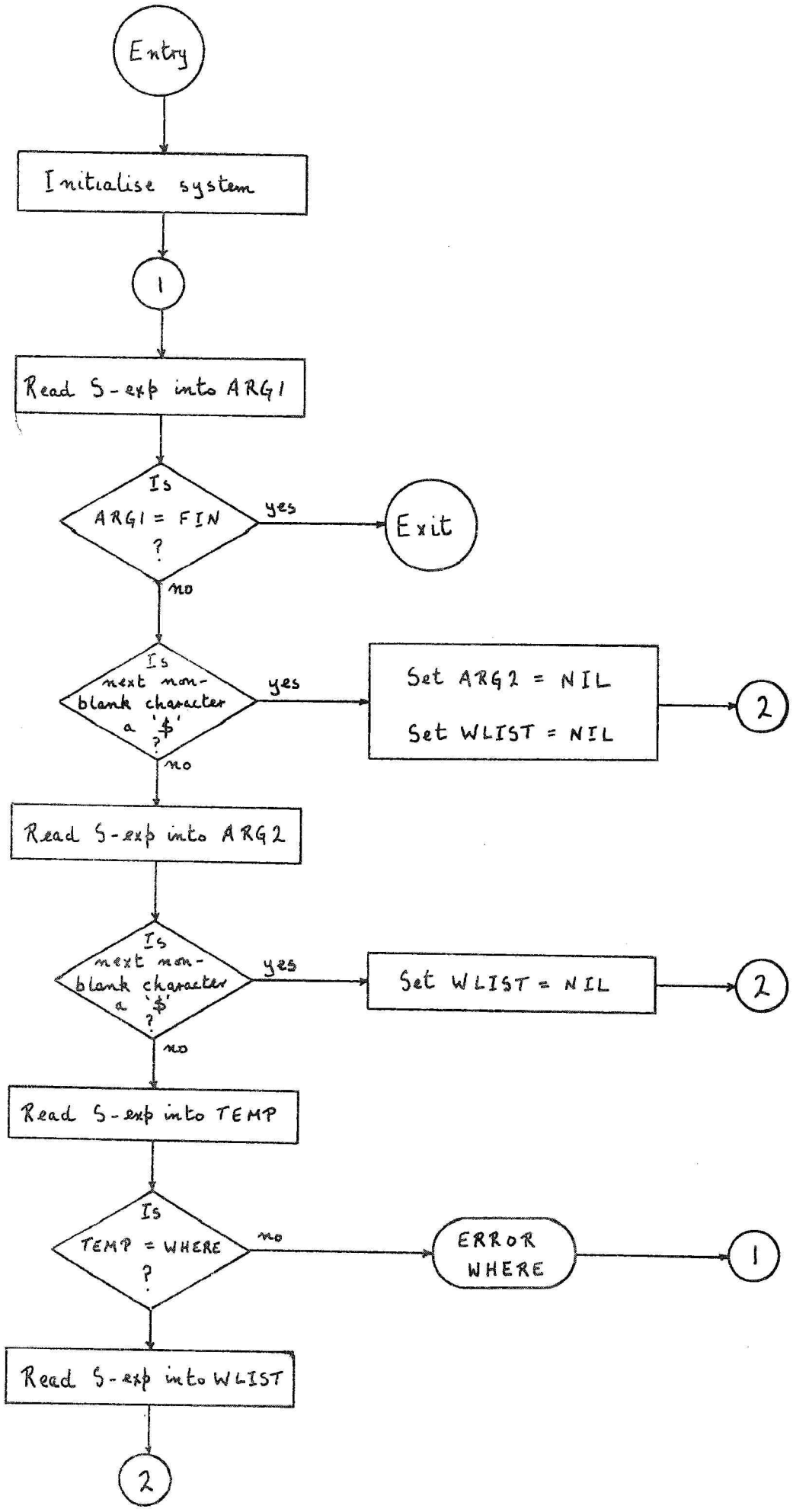
C.1 System's control structure

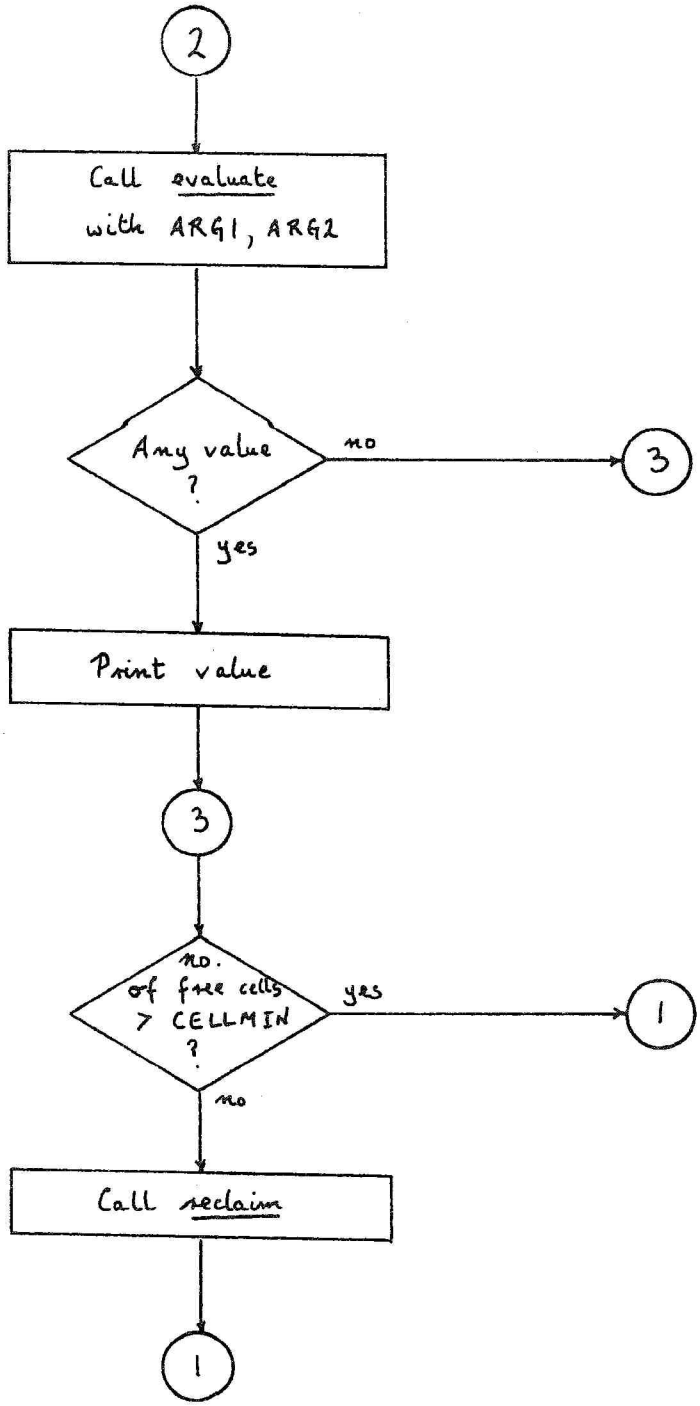
This section describes the overall flow of control during operation in terms of a flow-chart. In essence, a loop with three components is involved, that is, read a command, execute the issued command, display the value and then back to read the next command and so on. This loop may be terminated by issuing a FIN command, which causes a branch out of the loop and the system to come to a halt.

A command must take one of three formats - either a single S-expression or a doublet (i.e. a pair of S-expressions) or a pair of doublets. In the third instance, the first S-expression of the second doublet must be the atom WHERE, otherwise an error message will be output and control passed back to the start of the loop. If WHERE is encountered, then the fourth S-expression is set into the special register WLIST. WLIST is set to NIL for single S-expression or doublet commands. The symbol '$' is used as a terminator for each command.

The special registers ARG1, ARG2, ... are used for transmitting arguments to functions.

The register CELLMIN holds an integer number (set to zero during initialisation, with subsequent resettings through the function setgarb). At the end of every loop, if the number of cells still available in the working space is discovered to be less than the number held in CELLMIN, then garbage collection is invoked through reclaim before control passes to the next input phase, otherwise control passes directly to the next read.


Entry
Initialise system
1
Read S-exp into ARG1
Is ARG1 = FIN ?
yes
Exit
no
Is next non-blank character a '$' ?
yes
Set ARG2 = NIL
Set WLIST = NIL
2
no
Read S-exp into ARG2
Is next non-blank character a '$' ?
yes
Set WLIST = NIL
2
no
Read S-exp into TEMP
Is TEMP = WHERE ?
no
ERROR WHERE
1
Read S-exp into WLIST
2


2
Call evaluate
with ARG1, ARG2
Any value
?
no
3
yes
Print value
3
no.
of free cells
> CELLMIN
?
yes
1
no
Call reclaim
1

C.2 Functions of the interpreter

In this section, the functions evaluate, evalquote, apply, eval, evcon and replace are defined in a language that follows the M-expression notation as closely as possible and contains some insertions in English. The purpose is to describe as accurately as possible the actual working of the modified interpreter.

```
evaluate [fn;args] = [
marked [fn] → evalquote [fn; replace [args; WLIST]];
T → evalquote [fn;args]].
```

If the first argument of evaluate is a "marked" function, then the second argument is replaced before control passes to evalquote, otherwise evalquote is entered with the original arguments.

```
evalquote [fn;args] = [
get [fn; FEXPR] V get [fn; FSUBR] → eval [cons [fn;args]; NIL];
T → apply [fn;args; NIL]].
```

This exhibits no change from the standard version of evalquote, described in the LISP 1.5 Programmer's Manual, [12].

```
apply [fn;args;a] = [
null [fn] → NIL;
atom [fn] → [get [fn;EXPR] → apply [expr[1];args;a];
        get [fn;RULE] → { CURRULE: = fn;
                          apply [rule[1];args;a]; } ;
        get [fn;SUBR] → { spread [args];
                          ALIST: = a;
                          obey (subr[1]); } ;
```

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.

```
        null [sassoc [fn;a;NIL]] → pause [NO DEFINITION FOR fn];
        T → apply [cdr [assoc' [fn;a]];args;a]];
eq [car [fn];LABEL] → apply [caddr [fn];args;cons [cons [cadr [fn];caddr [fn]];a]];
eq [car [fn];FUNARG] → apply [cadr [fn];args;caddr [fn]];
eq [car [fn];LAMBDA] → eval [caddr [fn];nconc [pair [cadr [fn];args];a]];
eq [car [fn];DARG] → eval [match [cddr [fn];args];ALIST];
T → apply [eval [fn;a];args;a]].
```

In this description of apply (and in that of eval which is to follow), spread can be regarded as a pseudo-function of one argument, which should be a list. spread puts the individual elements of this list into ARG1, ARG2, ARG3, ... the standard registers for transmitting arguments to functions.

```
                    eval [form;a] = [
null [form] → NIL;
numberp [form] → form;
atom [form] →   [get [form;APVAL] → car [apval[1]];
                 T → cdr [sassoc [form;a;error [A8]]]];
eq [car [form];COND] → evcon [cdr [form];a];
atom [car [form]] → [get [car [form];EXPR] → apply [expr[1];evlis [cdr [form];a];a];
                     get [car [form];RULE] → { CURRULE: = car [form];
                                               apply [rule[1];evlis [cdr [form];a];a]; };
                     get [car [form];SUBR] → { spread [evlis [cdr [form];a]];
                                               ALIST: = a;
                                               obey(subr[1]); };
                     get [car [form];FEXPR] → apply [fexpr[1];list [cdr [form];a];a];
                     get [car [form];FSUBR] → { ARG1: = cdr [form];
                                                ARG2: = ALIST: = a;
                                                obey(fsubr[1]); };
```

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.

```
null [sassoc [car [form];a;NIL]]
        V eq [car [form];assoc [car [form];a]]
        → pause [NO DEFINITION FOR eval [car [form]]];
query [IS eval [assoc [car [form];a]] A VALID BINDING ?]
                → eval [cons [cdr [assoc [car [form];a]];cdr [form]];a]
        T → pause [NEED DEFINITIONS FOR eval [assoc [car [form];a]]]];
T → apply [car [form];evlis [cdr [form];a];a]].

                evcon [c;a] = [
null [c] → error [A3];
eval [caar [c];a] → eval [cadar [a];a];
T → evcon [cdr [c];a]].

                replace [e;a] = [
atom [e] → [null [sassoc [e;a;NIL]] → [get [e;IDEN] → car [iden[1]];
                                        T → e];
        T → cdr [assoc [e;a]]];
eq [car [e];EVAL] → eval [cadr [e];a];
eq [car [e];QUOTE] → cadr [e];
T → prog 2 [{rplaca [e;replace [car [e];a]];
             rplacd [e;replace [cdr [e];a]];} ; e ]]
```

The basic differences between this interpreter and the standard LISP 1.5 interpreter arise from the embedding of matching processes and the introduction of system pauses and queries.

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.

APPENDIX D

Listings of Examples

This appendix contains listings of all the examples mentioned in the text. Some extra examples are included in §D.3 to illustrate more fully the use of the provided identification and editing facilities.

The on-line operating system,† which acts as host for this programming system, uses a colon symbol to invite user input, and the user's program name (CAMS in this case) followed by a colon to indicate output.

D.1 Examples from Chapter II

D.1.1 Differentiation with allied simplification

```
: DERIVES ((
: (+(DARG(A B)
: LAST ( (A B)  (LIST("+) A B) ) ))
: (-(DARG(A B)
: LAST  ( (A B)  (LIST("-) A B) ) ))
: (*(DARG(A B)
: LAST ( (A B)  (LIST("*) A B) ) ))
: (/(DARG(A B)
: LAST ( (A B)  (LIST("/) A B) ) ))
: (↑(DARG(A B)
: LAST ( (A B)  (LIST("↑) A B) ) ))
: (#(DARG(A)
: LAST ( (A)  (LIST("#) A) ) ))  )) $
CAMS: (+ - * / ↑ #)
```

†The on-line operational environment is provided by the Multi-plexed Console System developed at O.U.B. by J. Warne and J. Hall - no publications.

```
      : DEFRULES ((
      : (D(DARG(Y X)
      : D1 ( (Y X)  0  (NP Y) )
      : D2 ( (X X)  1 )
      : D3 ( ((+ U V)X)  (+(D U X)(D V X)) )
      : D4 ( ((* U V)X)  (+(* U(D V X))(* V(D U X))) )
      : D5 ( ((/ U V)X)  (/(-(* V(D U X))(* U(D V X)))(↑ V 2)) )
      : D6 ( ((↑ U V)X)  (*(* V(↑U(- V 1)))(D U X))  (NP V) )
      : ))  )) $
CAMS: (D)

    : D((+(#(* 7 Z))3)Z) $
CAMS:  !! NO MATCH FOR
CAMS: (D(#(* 7 Z))Z)
CAMS: (+ NIL 0)
    : ADDRULE(D NIL (
    : D7 ( ((# Y)X)  (#(D Y X)) ) )) $
    : ADDRULE(+ LAST (
    : A1 ( (A 0)  A) )) $
    : D((+(#(* 7 Z))3)Z) $
CAMS: (#(+(* 7 1)(* Z 0)))
    : ADDRULE(* LAST (
    : A1 ( (A 1)  A) )) $
    : ADDRULE(* LAST (
    : A2 ( (A 0)  0) )) $
    : D((+(#(* 7 Z))3)Z) $
CAMS: (# 7)
```

```
      : D((+(* 6(↑ Z 3))(* 2 Z)) Z) $
CAMS: (+(* 6(* 3(↑ Z(- 3 1)))) 2)
      : ADDRULE(- LAST (
      : A1 ( (A B)  (DIFFERENCE A B)  (AND(NP A)(NP B)) ) )) $
      : ADDRULE(* LAST (
      : A3 ( (A(* B C))  (*(TIMES A B)C)  (AND(NP A)(NP B)) ) )) $
      : D((+(* 6(↑ Z 3))(* 2 Z)) Z) $
CAMS: (+(* 18(↑ Z 2)) 2)

      : D((/ T(- T 1)) T) $
CAMS:  !! NO MATCH FOR
CAMS: (D(- T 1) T)
CAMS: (/(-(- T 1)(* T NIL))(↑(- T 1) 2))
      : ADDRULE(D NIL (
      : D8 ( ((- U V)X)  (-(D U X)(D V X)) ) )) $
      : D((/ T(- T 1)) T) $
CAMS: (/(-(- T 1) T)(↑(- T 1) 2))
      : ADDRULE( - LAST (
      : A2 ( ((- A B) A)  (# B) ) )) $
      : D((/ T(- T 1)) T) $
CAMS: (/(# 1)(↑(- T 1) 2))

      : D((+(* 8(↑ X 2))(/ 8(↑ X 2))) X) $
CAMS: (+(* 16(↑ X 1))(/(- 0(* 16(↑ X 1)))(↑(↑ X 2) 2)))
      : ADDRULE(↑ LAST (
      : A1 ( (A 1)  A) )) $
      : ADDRULE(- LAST (
      : A3 ( (0 B)  (# B) ) )) $
      : ADDRULE(* LAST (
      : A2 ( ((↑ A B) C)  (↑ A(* B C)) ) )) $
```

```
      : D((+(* 8(↑ X 2))(/ 8(↑ X 2)))X) $
CAMS: (+(* 16 X)(/(#(* 16 X))(↑ X(* 2 2))))
      : ADDRULE(* LAST (
      : A1 ( (A B)  (TIMES A B)  (AND(NP A)(NP B)) ) )) $
      : ADDRULE(/ LAST (
      : A1 ( ((# A)B)  (#(/ 1 B)) ) )) $
      : ADDRULE(/ LAST (
      : A2 ( ((* A B)(↑ B C))  (/ A(↑ B(- C 1))) ) )) $
      : D((+(* 8(↑ X 2))(/ 8(↑ X 2)))X) $
CAMS: (+(* 16 X)(#(/ 16(↑ X 3))))
      : ADDRULE(+ LAST (
      : A2 ( (A(# B))  (- A B) ) )) $
      : D((+(* 8(↑ X 2))(/ 8(↑ X 2)))X) $
CAMS: (-(* 16 X)(/ 16(↑ X 3)))

      : DEFRULES((
      : (SIN(DARG(A)
      : LAST ( (A)  (LIST (" SIN) A) ) ))
      : (COS(DARG(A)
      : LAST ( (A)  (LIST (" COS) A) ) )) ) $
CAMS: (SIN COS)

      : ADDRULE(D NIL (
      : D9 ( ((SIN U)X)  (*(D U X)(COS U)) ) )) $
      : ADDRULE(D NIL (
      : D10 ( ((COS U)X)  (*(D U X)(#(SIN U))) ) )) $

      : D((+(#(COS X))(SIN X))X) $
CAMS: (+(#(* 1(#(SIN X))))(* 1(COS X)))
      : ADDRULE(* LAST (
      : A5 ( (1 A)  A) )) $
```

```
: ADDRULE(# LAST (
: A1 ( ((# A))  A) )) $
: D((+(#(COS X))(SIN X))X) $
CAMS: (+(SIN X)(COS X))

: D((SIN(* 4 Y))Y) $
CAMS: (* 4(COS(* 4 Y)))

: D((*(SIN X)(COS X))X) $
CAMS: (+(*(SIN X)(#(SIN X)))(*(COS X)(COS X)))
: ADDRULE(* LAST (
: A6 ( (A(# B))  (#(* A B)) ) )) $
: ADDRULE(* LAST (
: A1 ( (A A)  (↑ A 2) ) )) $
: D((*(SIN X)(COS X))X) $
CAMS: (+(#(↑(SIN X)2))(↑(COS X)2)) $
: ADDRULE(+ LAST (
: A3 ( ((# A)B)  (- B A) ) )) $
: D((*(SIN X)(COS X))X) $
CAMS: (-(↑(COS X)2)(↑(SIN X)2))

: D((/(SIN T)(COS T))T) $
CAMS: (/(-(↑(COS T)2)(#(↑(SIN T)2)))(↑(COS T)2))
: ADDRULE(- LAST (
: A4 ( (A(# B))  (+ A B) ) )) $
: ADDRULE(+ LAST (
: A4 ( ((↑(COS A)2)(↑(SIN A)2))  1 ) )) $
: D((/(SIN T)(COS T))T) $
CAMS: (/ 1(↑(COS T)2))
```

```
: DISPLAY(D) $
CAMS: ## ASSERTIONS
CAMS: D1 ( (Y X)  0  (NP Y) )
CAMS: D2 ( (X X)  1 )
CAMS: D3 ( ((+ U V)X)  (+(D U X)(D V X)) )
CAMS: D4 ( ((* U V)X)  (+(* U(D V X))(* V(D U X))) )
CAMS: D5 ( ((/ U V)X)  (/(-(* V(D U X))(* U(D V X)))(↑ V 2)) )
CAMS: D6 ( ((↑ U V)X)  (*(* V(↑ U(- V 1)))(D U X))  (NP V) )
CAMS: D7 ( ((# Y)X)  (#(D Y X)) )
CAMS: D8 ( ((- U V)X)  (-(D U X)(D V X)) )
CAMS: D9 ( ((SIN U)X)  (*(D U X)(COS U)) )
CAMS: D10 ( ((COS U)X)  (*(D U X)(#(SIN U))) )

: DISPLAY(+) $
CAMS: ## ASSERTIONS
CAMS: A1 ( (A 0)  A )
CAMS: A2 ( (A(# B))  (- A B) )
CAMS: A3 ( ((# A)B)  (- B A) )
CAMS: A4 ( ((↑(COS A)2)(↑(SIN A)2))  1 )
CAMS: LAST ( (A B)  (LIST(QUOTE +) A B) )

: DISPLAY(-) $
CAMS: ## ASSERTIONS
CAMS: A1 ( (A B)  (DIFFERENCE A B)  (AND(NP A)(NP B)) )
CAMS: A2 ( ((- A B)A)  (# B) )
CAMS: A3 ( (0 B)  (# B) )
CAMS: A4 ( (A(# B))  (+ A B) )
CAMS: LAST ( (A B)  (LIST(QUOTE -) A B) )
```

```
      : DISPLAY(*) $
    CAMS:  ## ASSERTIONS
    CAMS: A1 ( (A 1)  A )
    CAMS: A2 ( (A 0)  0 )
    CAMS: A3 ( (A(* B C))  (*(TIMES A B) C)  (AND(NP A)(NP B)) )
    CAMS: A4 ( (A B)  (TIMES A B)  (AND(NP A)(NP B)) )
    CAMS: A5 ( (1 A)  A )
    CAMS: A6 ( (A(# B))  (#(* A B)) )
    CAMS: A7 ( (A A)  (↑ A 2) )
    CAMS: LAST ( (A B)  (LIST(QUOTE *) A B) )

       : DISPLAY(/) $
    CAMS:  ## ASSERTIONS
    CAMS: A1 ( ((# A) B)  (#(/ A B)) )
    CAMS: A2 ( ((* A B)(↑ B C))  (/ A(↑ B(- C 1))) )
    CAMS: LAST( (A B)  (LIST(QUOTE /) A B) )

       : DISPLAY(↑) $
    CAMS:  ## ASSERTIONS
    CAMS: A1 ( (A 1)  A )
    CAMS: A2 ( ((↑ A B) C)  (↑ A(* B C)) )
    CAMS: LAST( (A B)  (LIST(QUOTE ↑) A B) )

       : DISPLAY(#) $
    CAMS:  ## ASSERTIONS
    CAMS: A1 ( ((# A))  A )
    CAMS: LAST ( (A)  (LIST(QUOTE #) A) )
       :
       :
       :
       :
```

D.1.2 The Wang algorithm for the propositional calculus

D.1.2.1 Without print out of steps

```
: DEFINE((
: (JOINT(LAMBDA(X Y)
: (COND((NULL X)NIL)
:       ((MEMBER(CAR X)Y)T)
:       (T(JOINT(CDR X)Y)) )))
: (TEST(LAMBDA(S)(ARROW NIL NIL NIL (LIST S NIL)) )) )) $
CAMS: (JOINT TEST)

: DEFRULES((
: (ARROW(DARG(L1 L2 R1 R2)
:       (L1 L2 R1 (X R2)) (ARROW L1 L2(CONS X R1) R2) (ATOM X) )
: STKLHS ( (L1 (X L2) R1 R2) (ARROW(CONS X L1)L2 R1 R2) (ATOM X) )
: P2A ( (L1 L2 R1((NOT P)R2)) (ARROW L1(LIST P L2)R1 R2) )
: P2B ( (L1((NOT P)L2)R1 R2) (ARROW L1 L2 R1 (LIST P R2)) )
: P3A ( (L1 L2 R1((AND A B)R2)) (AND(ARROW L1 L2 R1(LIST A R2))
:                                 (ARROW L1 L2 R1(LIST B R2))) )
: P3B ( (L1((AND A B)L2)R1 R2) (ARROW L1(LIST A(LIST B L2))R1 R2) )
: P4A ( (L1 L2 R1((OR A B)R2)) (ARROW L1 L2 R1(LIST A(LIST B R2))) )
: P4B ( (L1((OR A B)L2)R1 R2) (AND(ARROW L1(LIST A L2)R1 R2)
:                                 (ARROW L1(LIST B L2)R1 R2)) )
: P5A ( (L1 L2 R1((IMPLIES A B)R2)) (ARROW L1(LIST A L2)R1(LIST B R2)) )
: P5B ( (L1((IMPLIES A B)L2)R1 R2) (AND(ARROW L1 (LIST B L2) R1 R2)
:                                 (ARROW L1 L2 R1(LIST A R2))) )
: P6A ( (L1 L2 R1((EQUIV A B)R2)) (AND(ARROW L1(LIST A L2)R1(LIST B R2))
:                                 (ARROW L1(LIST B L2)R1(LIST A R2))) )
```

```
      : P6B ( (L1((EQUIV A B)L2)R1 R2) (AND(ARROW L1(LIST A(LIST B L2))R1 R2)
                                              (ARROW L1 L2 R1(LIST A(LIST B R2)))) )
      : TRUEORFALSE ( (L1 L2 R1 R2) (JOINT L1 R1) ) )) )) $
CAMS: (ARROW)

      : TEST((IMPLIES P (OR P Q))) $
CAMS: *T*

      : TEST((IMPLIES (NOT(OR P Q)) (NOT P))) $
CAMS: *T*

      : TEST((IMPLIES P (AND P Q))) $
CAMS: NIL

      : TEST((IMPLIES (AND(NOT P)(NOT Q)) (EQUIV P Q))) $
CAMS: *T*
```

D.1.2.2 With print out of steps

```
      : DEFINE((
      : (TEST(LAMBDA(S)(ARROW(" START)NIL NIL NIL(LIST S NIL)) )) )) $
CAMS: (TEST)

      : DEFRULES((
      : (ARROW(DARG(LAB L1 L2 R1 R2)
      : STKRHS ( (LAB L1 L2 R1(X R2)) (ARROW LAB L1 L2(CONS X R1)R2) (ATOM X) )
      : STKLHS ( (LAB L1(X L2)R1 R2) (ARROW LAB(CONS X L1)L2 R1 R2) (ATOM X) )
      : OUT ( (LAB L1 L2 R1 R2) (PROG2
      :                  (PRINT(LIST LAB(":)L1(";)L2("=>)R2(";)R1))
      :                  (ARR(L1 L2 R1 R2)) ) ))
      : (ARR(DARG(L1 L2 R1 R2)
      : P2A ( (L1 L2 R1((NOT P)R2)) (ARROW("P2A)L1(LIST P L2)R1 R2) )
      : P2B ( (L1((NOT P)L2)R1 R2) (ARROW("P2B)L1 L2 R1(LIST P R2)) )
```

```
        : P3A ( (L1 L2 R1 ((AND A B)R2))
        :
                              (AND(ARROW("P3A1)L1 L2 R1(LIST A R2))
        :
                              (ARROW("P3A2)L1 L2 R1(LIST B R2))) )
        : P3B ( (L1 ((AND A B)L2)R1 R2)
                              (ARROW("P3B)L1(LIST A(LIST B L2))R1 R2) )
        : P4A ( (L1 L2 R1((OR A B)R2))
        :
                              (ARROW("P4A)L1 L2 R1(LIST A(LIST B R2))) )
        : P4B ( (L1 ((OR A B)L2)R1 R2)
        :
                              (AND(ARROW("P4B1)L1(LIST A L2)R1 R2)
        :
                              (ARROW("P4B2)L1(LIST B L2)R1 R2)) )
        : P5A ( (L1 L2 R1 ((IMPLIES A B)R2))
        :
                              (ARROW("P5A)L1(LIST A L2)R1(LIST B R2)) )
        : P5B ( (L1 ((IMPLIES A B)L2) R1 R2)
        :
                              (AND(ARROW("P5B1)L1(LIST B L2)R1 R2)
        :
                              (ARROW("P5B2)L1 L2 R1(LIST A R2))) )
        : P6A ( (L1 L2 R1 ((EQUIV A B)R2))
        :
                              (AND(ARROW("P6A1)L1(LIST A L2)R1(LIST B R2))
        :
                              (ARROW("P6A2)L1(LIST B L2)R1(LIST A R2))) )
        : P6B ( (L1 ((EQUIV A B)L2)R1 R2)
        :
                              (AND(ARROW("P6B1)L1(LIST A(LIST B L2))R1 R2)
        :
                              (ARROW("P6B2)L1 L2 R1(LIST A(LIST B R2)))) )
        : TRUE ( (L1 L2 R1 R2)  (PROG2(PRINT("VALID))T)  (JOINT L1 R1) )
        : FALSE ( (L1 L2 R1 R2)  (PROG2(PRINT("INVALID))F) ) )) )) $
    CAMS: (ARROW ARR)

        : TEST((IMPLIES P (OR P Q))) $
    CAMS: (START: NIL; NIL=>((IMPLIES P (OR P Q)) NIL);NIL)
    CAMS: (P5A: (P); NIL=>((OR P Q) NIL); NIL)
    CAMS: (P4A: (P); NIL=> NIL;(Q P))
    CAMS: VALID
    CAMS: *T*
```

```
: TEST((IMPLIES(NOT (OR P Q))(NOT P))) $
CAMS: (START: NIL; NIL=>((IMPLIES(NOT(OR P Q))(NOT P))NIL); NIL)
CAMS: (P5A: NIL;((NOT(OR P Q))NIL) => ((NOT P)NIL); NIL)
CAMS: (P2A: (P); ((NOT(OR P Q))NIL) => NIL; NIL)
CAMS: (P2B: (P); NIL=>((OR P Q)NIL); NIL)
CAMS: (P4A: (P); NIL=> NIL;(Q P))
CAMS: VALID
CAMS: *T*

: TEST((IMPLIES P (AND P Q))) $
CAMS: (START: NIL; NIL=>((IMPLIES P (AND P Q))NIL); NIL)
CAMS: (P5A: (P); NIL=>((AND P Q)NIL); NIL)
CAMS: (P3A1: (P); NIL=> NIL; (P))
CAMS: VALID
CAMS: (P3A2: (P); NIL=> NIL; (Q))
CAMS: INVALID
CAMS: NIL

: TEST((IMPLIES(AND(NOT P(NOT Q))(EQUIV P Q))) $
CAMS: (START: NIL; NIL=>((IMPLIES(AND(NOT P)(NOT Q))(EQUIV P Q))NIL; NIL)
CAMS: (P5A: NIL;((AND(NOT P)(NOT Q))NIL)=>((EQUIV P Q)NIL); NIL)
CAMS: (P3B: NIL;((NOT P)((NOT Q)NIL))=>((EQUIV P Q)NIL); NIL)
CAMS: (P2B: NIL;((NOT Q)NIL)=>((EQUIV P Q)NIL); (P))
CAMS: (P2B: NIL; NIL=>((EQUIV P Q)NIL); (Q P))
CAMS: (P6A1: (P); NIL=> NIL; (Q Q P))
CAMS: VALID
CAMS: (P6A2: (Q); NIL=> NIL; (P Q P))
CAMS: VALID
CAMS: *T*
```

D.2 Examples from Chapter III

D.2.1 linear using left-to-right match

```
: DEFRULES((
: (LINEAR(DARG(X Y)
: L1 ( (X X)  ("L1) )
: L2 ( (X(* A X))  ("L2)  (FREE A X) )
: L3 ( (X(* X A))  ("L3)  (FREE A X) )
: L4 ( (X(+ X B))  ("L4)  (FREE B X) )
: L5 ( (X(+ B X))  ("L5) (FREE B X) )
: L6 ( (X(+(* A X)B))  ("L6)  (AND(FREE A X)(FREE B X)) )
: L7 ( (X(+(* X A)B))  ("L7)  (AND(FREE A X)(FREE B X)) )
: L8 ( (X(+ B(* A X)))  ("L8)  (AND(FREE A X)(FREE B X)) )
: L9 ( (X(+ B(* X A)))  ("L9)  (AND(FREE A X)(FREE B X)) )
: ))  )) $
CAMS: (LINEAR)
```

In the event of a successful match being obtained, the value returned by this definition of linear is the label of the successful assertion.

```
: LINEAR(Z Z) $
CAMS: L1

: LINEAR(Z (* 6 Z)) $
CAMS: L2

: LINEAR(X (* X 3)) $
CAMS: L3

: LINEAR(Y (+ Y Z)) $
CAMS: L4
```

```
      : LINEAR(Y (+ 4 Y)) $
CAMS: L5

      : LINEAR(X (+(* 3 X)(* 2 Z))) $
CAMS: L6

      : LINEAR(Y (+(* Y(* 3 Z))(* 4 Z))) $
CAMS: L7

      : LINEAR(X (+(* 3 Z)(* 2 X))) $
CAMS: L8

      : LINEAR(X (+ 4(* X(+ Y 2)))) $
CAMS: L9
```

D.2.2 linear using transformational match

```
      : DEFRULES((
      : (LINEAR(DARG(X E)
      : L1 ( (X(+(* A X)B))  (LIST ("A=) A (";X=) X (";B=) B)
      :          ((A FREE A X)(B FREE B X))  ((+ T1 T2)(* T4 T5)) ) ))
      : (TRFS(DARG NIL
      : T1 ( (+ A B)  (+ B A) )
      : T2 ( A  (+ A 0) )
      : T4 ( (* A B)  (* B A) )
      : T5 ( A  (* 1 A) ) ))  )) $
CAMS: (LINEAR TRFS)
```

This definition of linear uses $ax + b$ as the standard linear form in x, and if a successful match is found, then the value returned is a list like

$$(A = a \;;\; X = x \;;\; B = b).$$

The examples performed in §D.2.1 using the left-to-right version of linear are now repeated.

```
: LINEAR(Z Z) $
CAMS: (A = 1; X = Z; B = 0)

: LINEAR(Z (* 6 Z)) $
CAMS: (A = 6; X = Z; B = 0)

: LINEAR(X (* X 3)) $
CAMS: (A = 3; X = X; B = 0)

: LINEAR(Y (+ Y Z)) $
CAMS: (A = 1; X = Y; B = Z)

: LINEAR(Y (* 4 Y)) $
CAMS: (A = 1; X = Y; B = 4)

: LINEAR(X (+(* 3 X)(* 2 Z))) $
CAMS: (A = 3; X = X; B = (* 2 Z))

: LINEAR(Y (+(* Y(* 3 Z))(* 4 Z))) $
CAMS: (A = (* 3 Z); X = Y; B = (* 4 Z))

: LINEAR(X (+(* 3 Z)(* 2 X))) $
CAMS: (A = 2; X = X; B = (* 3 Z))

: LINEAR(X (+ 4(* X(+ Y 2)))) $
CAMS: (A = (+ Y 2); X = X; B = 4)
```

D.3 Examples from Chapter V

D.3.1 Defining, editing and display facilities

```
: DEFRULES((
:(NAME(DANG(D1 D2 D3)
: L1 (F1 S1 P1 T1)
: L2 (F2 S2 P2 T2)
: L3 (F3 S3 P3 T3) )) )) $
CAMS: (NAME)

: ADDRULE(NAME L3 (L2A (F2A S2A P2A T2A))) $
: ADDRULE(NAME NIL (L4 (F4 S4 P4 T4))) $
: DELRULE(NAME L1) $
: FETCH(NAME L3) $
CAMS: (F3 S3 P3 T3)
: CHANGE(NAME L3 FORM F3B) $
CAMS: L3 (F3B S3 P3 T3)

: DISPLAY(NAME) $
CAMS: ## ASSERTIONS
CAMS: L2 (F2 S2 P2 T2)
CAMS: L2A (F2A S2A P2A T2A)
CAMS: L3 (F3B S3 P3 T3)
CAMS: L4 (F4 S4 P4 T4)

: DELRULE(NAME L3) $
: DELRULE(NAME L2A) $
: ADDRULE(NAME L4 (NEW (FNEW SNEW))) $
: CHANGE(NAME NEW TRES TNEW) $
CAMS: NEW (FNEW SNEW T TNEW)
```

```
      : DES LAX(NAME) $
CAMS: ## ASSERTIONS
CAMS: L2 (F2 S2 P2 T2)
CAMS: NEW (FUEN SNEW T TNEW)
CAMS: L4 (F4 S4 P4 T4)
```

D.3.2 Identification facilities

```
      : IDENT(Y (*(↑ X 3)(COS X))) $
      : IDENT?(DYDX (D Y X)) $
CAMS: (-(*(COS X)(* 3(↑ X 2)))(*(↑ X 3)(SIN X)))
      : IDENT?(Y (D Y X)) $
CAMS: (-(*(COS X)(* 3(↑ X 2)))(*(↑ X 3)(SIN X)))

      : IDENT(Y (+(↑ X 2)(SIN X))) $
      : D(Y X) $
CAMS: (+(* 2 X)(COS X))
      : D(LEE X) $
CAMS: (- 2 (SIN X))
      : IDENT?(D2Y LEE) $
CAMS: (- 2 (SIN X))

      : D(Y X) $
CAMS: (+(* 2 X)(COS X))
      : IDENT?(D2Y (D LEE X)) $
CAMS: (- 2 (SIN X))

      : IDENT(A (* 2 Z)) $
      : IDENT(B (↑ Z 2)) $
      : D((+(SIN A)(EXP B))X)
```

```
:                    D(((A.(* 3(↑ X 2)))(B.(↑ X 3))) $
CAMS: (+(*(* 6 X)(COS(* 3(↑ X 2))))(*(* 3(↑ X 2))(EXP(↑ X 3))))

: D((+(SIN A)(EXP B))Z) $
CAMS: (+(* 2(COS(* 2 Z)))(*(* 2 Z)(EXP(↑ Z 2))))
```

D.4 Examples from Chapter VI

This section contains the listings of two separate runs. During the first, the RULE's int and edge were created and subsequently stored on a private disk file. Also, certain PACK properties were created and stored. For the second session, all needed definitions were restored from the disk file, and the examples given in Chapter VI were performed. Following these examples, the transformations and simplification functions were displayed.

```
: DEFRULES((
: (INT(DARG(X T)
: I1 ( (X T)  (* E X)  ((E FREE T X)) )
: I2 ( (X(↑ X N))  (/(↑ X(+ N 1))(+ N 1))
:                  ((N AND(FREE N X)(NEQ N -1))) ((↑ T9)) )
: I3 ( (X(LOG X))  (* X(-(LOG X)1)) )
: I4 ( (X(SIN X))  (#(COS X)) )
: I5 ( (X(COS X))  (SIN X) )
: I6 ( (X(TAN X))  (#(LOG(COS X))) )
: I7 ( (X(EXP X))  (EXP X) )
: I8 ( (X(+ A B))  (+(INT X A)(INT X B)) )
: I9 ( (X(- A B))  (-(INT X A)(INT X B)) )
: I10 ( (X(# A))  (#(INT X A)) )
: I11 ( ( (X(* A B))  (* A(INT X B))
:                  ((A FREE A X))  ((* T4)) )
: I12 ( (X(*(OP A)B))  (* DEFACT(SUBST A X(INT X(OP X))))
:            ((OP OP= OP)(B NDRV X A B))  ((* T4 T6)) )
```

```
: I13 ( (X(/ B A))  (* DFACT(LOG A))  ((A MDRV X A B)) )
: I14 ( (X(*(↑ A N)B))  (*(/ DFACT(+ N 1))(↑ A(+ N 1)))
:              ((N AND(FREE N X)(NEQ N -1))(B MDRV X A B))
:                    ((* T4)(↑ T9)) )
: I15 ( (X(*(↑ C A)B))  (*(/ DFACT(LOG C))(↑ C A))
:              ((C FREE C X)(B MDRV X A B))  ((* T4)) )
: I16 ( (X I)  (EDGE X I)  ((I QUERY TRY EDGE(X I) ?)) )
: LAST ( (X I)  (LIST("INT) X I) ) ))
: (EDGE(DARG(X I)
: E1 ( (X(* H(EXP G)))  (PROG(A)
:        (SETQ A(/ H(DRV G X)))
:        (RETURN(-(* A(EXP G))(INT X(*(DRV A X)(EXP G))))))
:                  T  ((* T4)) )
: E2 ( (X(* H(LOG G)))  (PROG(A B C)
:           (SETQ C(INT X H))
:           (SETQ A(SIMP(SUBST 0 (LOG G) C)))
:           (SETQ C(/(/(- C A)(LOG G))2))
:           (SETQ B(INT X(/(* A(DRV G X))G)))
:           (RETURN(+(* C(↑(LOG G)2))(+(* A(LOG G))B))))
:                  T  ((* T4)) )
: E3 ( (X(* H(/ 1(+ 1(↑ G 2))))) (* DFACT(LIST("ARCTAN)G))
:           ((G MDRV X H G))  ((+ T1)(* T4)(↑ T11)) )
: E4 ((X(* H(SIN G)))  (PROG(A B)
:           (SETQ A(#(/ H(DRV G X))))
:           (SETQ B(INT X(#(*(DRV A X)(COS G)))))
:           (RETURN(+(* A(COS G))B)))
:                  T  ((* T4)) )
: E5 ( (X(* H(COS G)))  (PROG(A B)
```

```
      :              (SETQ A(/ H(DRV G X)))
      :              (SETQ B(INT X(#(*(DRV A X)(SIN G)))))
      :              (RETURN(+(* A(SIN G))B)))
      :                    T  ((* T4)) ) )) )) $
CAMS: (INT EDGE)

    : GETFILE(LDF) $
    : STORE((INT EDGE)LDF) $

    : DEFLIST((
    : (INTEGRATOR(INT EDGE MDRV DRV OPP + - * / ↑ #
    :                HCF LOG SIN COS EXP TAN TRFS SIMP))
    : (OPERATORS(+ - * / ↑ # HCF))
    : ) PACK) $
CAMS: (INTEGRATOR OPERATORS)

    : STORE((INTEGRATOR OPERATORS)LDF) $
```

This completed the first run. Notice that the list of all those functions which may be needed during an integration is now stored as a PACK property of the atomic symbol INTEGRATOR.

```
    : GETFILE(LDF) $
    : RESTORE(INTEGRATOR LDF) $

    : INT(X(↑ X(/ 3 2))) $
CAMS: (/(* 2(↑ X(/ 5 2)))5)

    : INT(X(+(SIN X)(EXP X))) $
CAMS: (-(EXP X)(COS X))

    : INT(X(COS(+(* 2 X)3))) $
CAMS: ## IS THE FOLLOWING A VALID BINDING ?
```

```
CAMS: (OP.EXP)

    : YES $

CAMS: (*(/ 1 2)(EXP(↑ X 2)))

    : INT(X(* X(↑(+ 1(↑ X 2))(/ 1 2)))) $

CAMS: (*(/ 1 3)(↑(+ 1(↑ X 2))(/ 3 2)))

    : INT(X(/(EXP X)(+ 1(EXP X)))) $

CAMS: (LOG(+ 1(EXP X)))

    : INT(X(*(* X(COS(↑ X 2)))(EXP(SIN(↑ X 2))))) $

CAMS: ## IS THE FOLLOWING A VALID BINDING ?

CAMS: (OP.EXP)

    : YES $

CAMS: (*(/ 1 2)(EXP(SIN(↑ X 2))))

    : INT(X(* X(EXP X))) $

CAMS: ## QUERY?

CAMS: (TRY EDGE(X(* X(EXP X))) ? )

    : YES $

CAMS: (*(- X 1)(EXP X))

    : INT(X(*(+ X(/ 1 X))(LOG X)))$$

CAMS: ## QUERY ?

CAMS: (TRY EDGE(X(*(+ X(/ 1 X))(LOG X))) ? )

    : YES $

CAMS: (+(*(/ 1 2)(↑(LOG X)2))(*(+(LOG X)(/1 2))(/(↑ X 2)2)))

    : INT(X(/ X(+ 1(↑ X 4)))) $

CAMS: ## QUERY ?

CAMS: (TRY EDGE(X(* X(/ 1(+ 1(↑ X 4))))) ? )

    : YES $

CAMS: (*(/ 1 2)(ARCTAN(↑ X 2)))
```

```
      : INT(X(*(↑ X 2)(SIN X))) $
CAMS:  ## QUERY ?
CAMS: (TRY EDGE(X(* (↑ X 2)(SIN X))) ? )
      : YES $
CAMS:  ## QUERY ?
CAMS: (TRY EDGE(X(* X(COS X))) ? )
      : YES $
CAMS: (-(*(* 2 X)(SIN X))(*(-(↑ X 2)2)(COS X)))

      : INT(X(*( EXP X)(SIN X))) $
CAMS:  ## QUERY ?
CAMS: (TRY EDGE(X(*(EXP X)(SIN X))) ? )
      : YES $
CAMS:  ## QUERY ?
CAMS: (TRY EDGE(X(*(EXP X)(COS X))) ? )
      : YES $
CAMS:  ## QUERY ?
CAMS: (TRY EDGE(X(*(EXP X)(SIN X))) ? )
      : NO $
CAMS: (-(*(SIN X)(EXP X))(+(*(COS X)(EXP X))
CAMS: (INT X(*(EXP X)(SIN X))))


      : DISPLAY(TRFS) $
CAMS:  ## TRANSFORMATIONS
CAMS: T1 ( (+ A B)  (+ B A) )
CAMS: T2 ( A  (+ A 0) )
CAMS: T3 ( A  (+ 0 A) )
CAMS: T4 ( (* A B)  (* B A) )
CAMS: T5 ( A  (* 1 A) )
```

```
CAMS: T6 ( A  (* A 1) )
CAMS: T7 ( A  (- A 0) )
CAMS: T8 ( A  (/ A 1) )
CAMS: T9 ( A  (↑ A 1) )
CAMS: T10 ( A  (#(# A)) )
CAMS: T11 ( (↑ A N)  (↑(↑ A(VAL(QUOTIENT N 2)))2)
CAMS: ((N AND(NP N)(ZTROP(REMAINDER N 2)))) )
CAMS: T12 ( (* A (* B C))  (* A (* C B)) )

    : DISPLAY(+) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A 0)  A  T  ((+ T1)) )
CAMS: A2 ( ((/ A B)(/ C D))  (/(PLUS(TIMES A D)(TIMES B C))
CAMS: (TIMES B D))  ((A NP A)(B NP B)(C NP C)(D NP D))  ((/ T8)) )
CAMS: A3 ( ((* A B)(* C B))  (*(+ A C)B)  T  ((* T4 T5)) )
CAMS: A4 ( (B(# A))  (- B A)  T  ((+ T1)) )
CAMS: LAST ( (A B)  (LIST(QUOTE +) A B) )

    : DISPLAY(-) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A 0)  A )
CAMS: A2 ( (0 A)  (# A) )
CAMS: A3 ( ((+ A B)(+ A C))  (- B C)  T  ((+ T1 T2)) )
CAMS: A4 ( ((/ A B)(/ C D))  (/(DIFFERENCE(TIMES A D)
CAMS: (TIMES B C))(TIMES B D))  ((A NP A)(B NP B)(C NP C)
CAMS: (D NP D))  ((/ T8)) )
CAMS: A5 ( ((* A B)(* C B))  (*(- A C)B)  T  ((* T4 T5)) )
CAMS: A6 ( (B(# A))  (+ B A) )
CAMS: A7 ( ((# A)B)  (#(+ A B)) )
CAMS: LAST ( (A B)  (LIST(QUOTE -) A B) )
```

```
: DISPLAY(/) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A 0)  (LIST(QUOTE UNDEFINED) A (QUOTE / 0)) )
CAMS: A2 ( (A 1) A )
CAMS: A3 ( (A B)  (QUOTIENT A B)  ((A NP A)(B AND(NP B)
CAMS: (ZEROP(REMAINDER A B)))) )
CAMS: A4 ( (A B) (LIST(QUOTE /)(QUOTIENT A(HCF A B))
CAMS: (QUOTIENT B HCFACT))  ((A FIXP A)(B FIXP B)) )
CAMS: A5 ( ((/ A B)(/ C D))  (/(* A D)(* B C))
CAMS: ((C NEQ C 1)(D OR(NEQ B 1)(NEQ D 1)))  ((/ T8)) )
CAMS: A6 ( (A(# B))  (#(/ A B)) )
CAMS: A7 ( ((# A)B)  (#(/ A B)) )
CAMS: A8 ( ((* D(↑ A B))(* E( A C)))  (*(/ D E)
CAMS: (↑ A(- B C)))  T  ((* T4 T5 T12)(↑ T9)) )
CAMS: LAST ( (A B)  (LIST(QUOTE /) A B) )

: DISPLAY(*) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A 0)  0  T  ((* T4)) )
CAMS: A2 ( (A 1)  A  T  ((* T4)) )
CAMS: A3 ( (A -1)  (# A)  T  ((* T4)) )
CAMS: A4 ( ((/ A B)(/ C D))  /(TIMES A C)(TIMES B D))
CAMS: ((A NP A)(B NP B)(C NP C)(D NP D))  ((/ T8)) )
CAMS: A5 ( ((↑ A B)( A C))  (↑ A(+ B C))  T  ((↑ T9)) )
CAMS: A6 ( (A(* B C))(*(TIMES A B)C)  ((A NP A)(B NP B))
CAMS: ((* T4)) )
CAMS: A7 ( (A(* B C))  (* B(* A C))  ((B NP B))  ((* T4)) )
CAMS: A8 ( (A(# B))  (#(* A B))  T  ((* T4)) )
CAMS: A9 ( (A B)  (LIST(QUOTE *) B A)  ((B NP B)) )
CAMS: LAST ( (A B)  (LIST(QUOTE *) A B) )
```

```
    : DISPLAY(↑) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A B)  (EXPT A B)  ((A NP A)(B NP B)) )
CAMS: A2 ( (0 0)  (QUOTE(UNDEFINED 0 ↑ 0)) )
CAMS: A3 ( (0 A)  0 )
CAMS: A4 ( (A 0)  1 )
CAMS: A5 ( (1 1)  1 )
CAMS: A6 ( (A 1)  A )
CAMS: LAST ( (A B)  (LIST(QUOTE ↑) A B) )

    : DISPLAY (#) $
CAMS:  ## ASSERTIONS
CAMS: A1 ( (A)  (MINUS A)  ((A NP A)) )
CAMS: A2 ( ((# A))  A )
CAMS: A3 ( ((- B A))  (- A B) )
CAMS: LAST ( (A)  (LIST(QUOTE #) A) )
    :
    :
```

REFERENCES

1. Bond, E., et al., "FORMAC - An experimental Formula Manipulation Compiler", Proc. ACM Nat. Conf., Philadelphia, 1964.

2. Desautels, E.J., and Smith, D.J., "An Introduction to the String Processing Language SNOBOL", in Programming Systems and Languages, McGraw-Hill, New York, 1967.

3. Engeli, M.A., "Achievements and Problems in Formula Manipulation", Proc. IFIPS Congress, Edinburgh, 1968.

4. Engleman, C., "MATHLAB: A Program for On-Line Machine Assistance in Symbolic Computations", Proc. FJCC, 1965.

5. Engleman, C., "Rational Functions in MATHLAB", Proc. of IFIPS Working Conference on Symbol Manipulation Languages, Pisa, 1966.

6. Farber, D.J., et al., "SNOBOL, a String Manipulation Language", J. ACM, Vol. 11, 1964.

7. Fenichel, R.R., "An On-Line System for Algebraic Manipulations", Ph.D. Thesis, Harvard University, 1966.

8. Guzman, A., and McIntosh, H.V., "CONVERT", Comm. ACM, Vol. 9, No. 8, 1966.

9. Hearn, A.C., "REDUCE User's Manual", Stanford A.I. Project Memo No. 50, 1967.

10. Knowlton, K.C., "A Programmer's Description of L6", Comm. ACM, Vol. 9, No. 8, 1966.

11. McBride, F.V., "A LISP Programming System for the ICT 1900", M.Sc. Dissertation, The Queen's University of Belfast, 1968.

12. McCarthy, J., et al., "LISP 1.5 Programmer's Manual", MIT Press, Cambridge, Massachusetts, 1963.

13. McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1", in Programming Systems and Languages, McGraw-Hill, New York, 1967.

14. Martin, W.A., "Symbolic Mathematical Laboratory", Ph.D. Thesis, MIT, Cambridge, Massachusetts, 1967.

15. Moses, J., "Symbolic Integration", Ph.D. Thesis, MIT, Cambridge, Massachusetts, 1967.

16. Newell, A., "Documentation of IPL-V", Comm. ACM, Vol. 6, No. 1, 1963.

17. Newell, A., Shaw, J.C., and Simon, H.A., "Empirical Explorations of the Logic Theory Machine", in Computers and Thought, McGraw-Hill, New York, 1963.

18. Risch, R.H., "The Problem of Integration in Finite Terms", Report SP-2801, Systems Development Corp., Santa Monica, California, 1967.

19. Ritt, J.F., "Integration in Finite Terms", Columbia Univ. Press, New York, 1947.

20. Sammett, J.E., "An Annotated Description Based Bibliography on the use of Computers for Non-Numerical Mathematics", Computing Review, Vol. 7, No. 4, 1966.

21. Sammett, J.E., "Survey of Formula Manipulation", Comm. ACM, Vol. 9, No. 8, 1966.

22. Slagle, J.R., "A Heuristic Program that solves Symbolic Integration Problems in Freshman Calculus", Ph.D. Thesis, MIT, Cambridge, Massachusetts, 1961.

23. Weizenbaum, J., "ELIZA - A Computer Program for the Study of Natural Language Communication between Man and Machine", Comm. ACM, Vol. 9, No. 1, 1966.

24. Weizenbaum, J., "Symmetric List Processor", Comm. ACM, Vol. 6, 1963.

25. Wooldridge, D., "An Algebraic Simplify Program in LISP", Stanford A.I. Project Memo. No. 11, 1963.

26. Yngve, V.H., "COMIT Programmer's Reference Manual", MIT Press, Cambridge Massachusetts, 1962.