A Study of Function Evaluation

Thesis

How to cite:

For guidance on citations see FAQs.

© 1973 The Author

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Version of Record

Link(s) to article on publisher’s website:
http://dx.doi.org/doi:10.21954/ou.ro.0000f249

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.

oro.open.ac.uk
A STUDY OF FUNCTION EVALUATION

A

THESIS

submitted in partial fulfilment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in the

FACULTY OF MATHEMATICS

of

THE OPEN UNIVERSITY

by

SOHRAB SAADAT, M.Sc.

February 1973

Date of submission: March 73
Date of award: June 73
This research was carried out under the supervision of Professor R.M. Pengelly for whose guidance and advice I am deeply grateful.

I would like to thank Dr. F.V. McBride for many constructive discussions, Mr. C.I. Johnston and Miss M.M. Gorman for reading the script, Dr. S.H. Reza for preparing most of the graphs and flow-charts and Mrs. J. Warne and Miss L. Montgomery for typing this thesis.

I would also like to thank the Computer Science Department of The Queen's University of Belfast, headed by Professor C.A.R. Hoare, for providing accommodation and encouragement, and the staff of the Computer Centre of The Queen's University of Belfast for the service they provided for the duration of the project.

This work was supported by a postgraduate research studentship from the Open University.
This thesis is concerned with an investigation into the techniques of function evaluation as used in symbol manipulation. The overall objective of this study is to investigate the possibility of constructing a programming system in which the performance of function evaluations improves in efficiency as a result of information provided by the user and gathered by the system as a result of previous evaluations. The approach to this problem is to construct a mechanism which allows the user and the system to work together to build up "useful properties of functions". The properties which are expressed in the form of relations between the function, its arguments and its corresponding function values, are used to improve the performance of function evaluations.

Various programming systems, each incorporating a particular set of facilities designed to improve the performance of function evaluation, are constructed and their performance relative to each other and relative to the underlying host system analysed. From the analysis of the experimental results, conclusions are drawn about those facilities which should be included in the design of a general purpose symbol manipulation system.

The present work is based on the assumption that by providing all the facilities within an interactive environment, the user can guide the course of an evaluation and thus improve the performance of the evaluation. This assumption, which is
based on the assertion that no programming system can
currently match the power of its human user to recognise
"significant" and "useful" items of information, leads to an
interactive symbol manipulation system being chosen as the
host system for the investigation. A number of the results
obtained are applicable to the design of other types of
programming systems.
# LIST OF CONTENTS

## CHAPTER 1 - INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>The purpose of this project</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Concepts</td>
<td>4</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Rules of correspondence</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2</td>
<td>The human approach</td>
<td>5</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Function evaluation</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Previous work</td>
<td>11</td>
</tr>
<tr>
<td>1.5</td>
<td>The thesis</td>
<td>13</td>
</tr>
</tbody>
</table>

## CHAPTER 2 - BACKGROUND MATERIAL

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>The host system</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Introduction</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Description</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Some typical problems</td>
<td>25</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Multiple differentiation</td>
<td>28</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Third order differentiation</td>
<td>28</td>
</tr>
<tr>
<td>2.3.3</td>
<td>A function with numerical arguments</td>
<td>29</td>
</tr>
</tbody>
</table>

## CHAPTER 3 - SYSTEMS WITH PERMANENT ATTRIBUTES

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Basic concepts</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>The Expression-Driven System</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Basic features</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The object-stack</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Operation of the interpreter</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>The Function-Driven System</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Basic features</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Operation of the interpreter</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>Repertoire of commands</td>
<td>48</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Repertoire of commands: EDS</td>
<td>49</td>
</tr>
</tbody>
</table>
3.5.2. Repertoire of commands: FDS 55
3.6. Analysis of practical results 57
  3.6.1. Introduction 57
  3.6.2. Problem 1 60
  3.6.3. Problem 2 64
  3.6.4. Problem 3 67
  3.6.5. Summary 70
3.7. Conclusion 72

CHAPTER 4 - FUNCTION DRIVEN SYSTEM WITH FIXED SIZE ATTRIBUTE-LISTS

4.1. Introduction 74
4.2. Basic concepts 75
4.3. Expression Driven System 77
4.4. Function Driven System 78
  4.4.1. Basic features 78
  4.4.2. Attribute house-keeping operations 80
  4.4.3. Operation of the interpreter 82
4.5. Repertoire of commands 85
4.6. Analysis of practical results 87
  4.6.1. Problem 1 88
  4.6.2. Problem 2 91
  4.6.3. Problem 3 94
4.7. Critical factors 95
  4.7.1. Size of system attribute-list 95
  4.7.2. Time of direct evaluation versus look-up time 96
  4.7.3. A choice of memo function 98
  4.7.4. Rate of promotion 100
4.8. Conclusion 101

CHAPTER 5 - SYSTEMS WITH DYNAMIC ATTRIBUTE-LISTS

5.1. Introduction 104
5.2. Basic concepts 105
CHAPTER 7 - CLASSIFICATION

7.1. Introduction

7.2. Basic concepts

7.3. Classification in expression driven system
   7.3.1. Basic features
   7.3.2. Attribute house-keeping operations
   7.3.3. Operation of the interpreter

7.4. Automatic classification
   7.4.1. Operation of the interpreter

7.5. Classification in the function driven system

7.6. Repertoire of commands

7.7. Analysis of practical results
   7.7.1. Discussion 1
   7.7.2. Discussion 2

7.8. Conclusion

CHAPTER 8 - SUMMARY, CONCLUSION AND POSSIBLE DEVELOPMENT

8.1. Overview and conclusion
   8.1.1. Systems with permanent attributes
   8.1.2. Function driven system with fixed-size attribute-lists
   8.1.3. Systems with dynamic attribute-lists
   8.1.4. Selectivity
   8.1.5. Classification
   8.1.6. Conclusion

8.2. Future developments
   8.2.1. A symbol manipulation system
   8.2.2. Further investigation

8.3. Epilogue
1.1. Introduction

The research reported in this thesis is an investigation into the possibility of constructing a programming system in which the mechanism of function evaluation improves its performance during the course of an evaluation. Such a system must be able to collect and to organise the results of the previous evaluations and utilize these results implicitly or explicitly on subsequent occasions. If the system is to be of real value, then the method of gathering and organising information must impose a minimum burden on the user, and the use of this information must improve the performance of the function evaluation mechanism.

The various programming systems developed as part of this investigation are all based upon modifications of a programming system developed by F.V. McBride, (4), a brief description of which appears in Chapter 2. This thesis contains a brief description of the various systems which have been developed and a comparative study of the performances of each of these systems.

The remainder of this chapter introduces the objectives of the project, describes the concepts underlying the project, and reviews the previous work which has been done in this area.
The first use of computers in any area of study is to mechanize the routine parts of tasks in essentially the same manner as they were formerly done by hand. Later, with the development of both new hardware facilities and new software techniques, the methods used for solving problems are altered to make better use of the resources provided by a computer system. So far, in most fields of computation, man has maintained the dominant role and the machine a subordinate one. J.C.R. Licklider, {22}, has described the "man computer symbiosis" which would result from the optimum utilization of the abilities of man and currently available computing facilities in a problem solving situation. He states - "Men will make approximate and fallible, but leading contributions, and they will define criteria and serve as evaluators, judging the contributions of the equipment and guiding the general line of thought. In addition, men will handle the very low probability situations when such situations arise. ...The information processing equipment, for its part, will convert hypotheses into testable models and then test the models against data." Licklider states further that the equipment will simulate, transfer, interpolate, extrapolate, carry out routinizable clerical operations, and remember precise values and exact details. Part of what is prophesised in these quotations has already been achieved, and the current investigation may be taken as a further small step in the direction of exploiting the stated principles in order to improve the performance of function evaluations.

The one facet of computation which hindered progress in
many scientific fields for decades and led to the development of automatic computers is the monotony and tedium of repetitive calculations. As a consequence, all contemporary programming languages are designed to deal with problems involving repetitive calculations. However, most of the facilities provided are very unsophisticated so that, for example, time is wasted by re-evaluating a result on each occasion it is needed. Any attempt to overcome this difficulty by saving "useful" values in the form of a table of values can lead to storage space being wasted by the unused entries of the tables, and to a burden on the user who has to construct and maintain the tables. These aspects of modern computing systems have been described by Michie, {19}, in the following terms. "If computers could learn from experience, their usefulness would be increased. When I write a clumsy program for a contemporary computer a thousand runs on the machine do not re-educate my handiwork. On every execution, each time-wasting blemish and crudity, each needless test and redundant evaluation, is meticulously reproduced".

Where possible, at the expense of the time of professional programmers, the programming techniques available and the capabilities of the programming languages can be combined in order to reduce the inefficiencies occurring in a program. With casual users and beginners, on the other hand, their lack of experience and their unfamiliarity with the intricacies of the programming languages and programming techniques prevent them from making the best use of the resources available to them. Consequently, the ability of the proposed system to collect, organise, supervise and use the results of function
evaluations almost automatically during a computation is likely to benefit both experienced users and the beginners.

Although the methods of evaluation proposed are investigated in the area of symbol manipulation, the aim is not to construct an efficient general purpose symbol manipulation system containing all the features proposed. Instead, the effect that the use of each proposed feature has on the efficiency of performing a number of specific manipulations is carefully examined.

The aspects of performance which are considered throughout this thesis are the time of evaluation, the use of storage space, and the burden that the use of any new feature imposes on the user. All the comparisons of performance of the systems implemented are made in terms of the components just mentioned. The aim of the comparative study is to provide some ideas and guidelines for the design of general purpose symbol manipulation systems.

1.3. Concepts

1.3.1. Rules of correspondence

A function is nothing more than a rule of correspondence designed to map the elements of an argument set onto the corresponding elements of a value set. There are many ways of expressing the rules of correspondence used to map the argument set onto the value set. When a function is to be evaluated using a computer, two rules of correspondence are of primary importance - table look-up and function definition in algorithmic form.
Given an argument set and a value set, several different rules of correspondence can be used to define the same mapping between these sets. The only programming system known to the author which takes account of this fact explicitly is POP-2, (14). The present investigation aims to exploit this situation in order to improve the efficiency of function evaluation.

Pictorially, an arbitrary function, \( f \), may be represented as:

\[
\begin{align*}
R_1 & \\
R_2 & \\
R_i & \\
R_{n-1} & \\
R_n &
\end{align*}
\]

where \( R_i \) for \( i = 1, 2, \ldots, n \) represents the various rules of correspondence which can represent the function \( f \). Notice that the input \( P \) will invariably produce the same output \( f(p) \), in spite of the fact that different rules of correspondence may be used on different occasions. A user can decide on the appropriate rule of correspondence by analysing the characteristics of the rules and the evaluation which is to be attempted.

1.3.2. The human approach

Let us interpret the diagram above by looking at how a
man using a pencil and paper evaluates a function. The various activities involved are:

(a) if the evaluation has not been carried out before, the function value must be calculated using a formal algorithmic definition of the function,

(b) if the evaluation was performed and recorded previously then the value is obtained from some standard table, for example, logarithm table or a table of standard integrals,

(c) if the evaluation has already been carried out during the present calculation, the value is retrieved from the record of the present computation,

(d) if a similar evaluation has been made before, the value may be deduced from a previous result. For example, having calculated the value of:

$$\int_0^\infty e^{-sx} \cos(ax) \, dx = \frac{s}{s^2 - a^2}$$

one can then use the result to obtain the value of:

$$\int_0^\infty \cos(b - 1)p e^{-pt} \, dp$$

This gives us the following picture:
Arising from these activities are house-keeping tasks that must be performed. These involve the organisation of the information used in an evaluation in such a way that:

(i) frequently required items of information are obtained with as little extra effort as possible,

(ii) less frequently required values can be obtained in some systematic fashion,
(iii) results which no longer serve a useful purpose are filed away for future reference.

Although some effort is needed to perform the house-keeping tasks, in the long run they result in an increase in the efficiency with which evaluations are performed.

The important aspects of this approach from our point of view are, first, the use of previous results eliminates re-evaluation and the process of obtaining the value is made more efficient. Second, at least two and often more different rules of correspondence, representing the same function, are available and the appropriate one is selected for each evaluation.

By incorporating in a computer system features capable of performing the tasks described above, the computer based evaluations can be made more efficient for the reasons cited. There is also an added advantage, that is, the system is solving a problem in a similar manner to that used by man. This similarity in approach increases the user's understanding of the manner in which the system is solving the problem, and in turn this demands less awareness on the part of the user.

1.3.3. Function evaluation

The material in this section is concerned with a computer based approach to functions whose values can be calculated by using more than one rule of correspondence. In fact, we restrict our consideration to two particular forms of correspondence, table look-up and algorithmic definition. A typical evaluation of a function using these techniques may be represented as follows:
As the diagram shows, given an argument of a function we can choose to attempt evaluation either by look-up or from the definition. In the former case the table is searched for the corresponding argument-result pair. If the search is successful, the corresponding value is returned. If the search is unsuccessful, the argument can be passed over to the definition of the function in order to attempt the evaluation that way. The system can be set up to make new entries in the table every time the definition of the function is used. Using this method of evaluation, a table containing no entries looks like a normal algorithmic definition while a function with no algorithmic definition looks like a table of values.

If the ideas described are implemented in a computer system, measures must be taken to ensure that most of the
entries in the table correspond to evaluations required subsequently in a computation. Unlike a man using pencil and paper, a computer system is not capable of recognising values which will subsequently be useful unless the necessary criteria are specified by the user. Notice, however, that the table can also be used to hold values which the system is not capable of computing because the algorithmic definition of the function is not sufficiently general.

Any function can be evaluated using this sort of mechanism. The table in the diagram should not be regarded as having an independent existence: rather, it constitutes part of the function definition. Once a function definition has associated with it a table of values, then the procedure shown in the diagram is used for evaluations involving this function.

If a particular function evaluation is performed many times, the proper use of a table of values has a great advantage, especially if it avoids the use of a rather inefficient algorithmic definition. If this sort of facility is provided, the user can opt to use inefficient function definitions if it save him time and effort because the definitions are used infrequently.

Each entry in a table of values consists of an argument result pair. In future discussion, an argument-result pair and a table of such pairs are referred to as an "attribute" and an "attribute-list", respectively. This terminology enables us to talk about the entries in the table as attributes of the function, thus emphasising the fact that they are properties of the function itself. Below, we define a number
of related terms which are used throughout this thesis.

permanent attribute This is an attribute defined by the user.
temporary attribute This is an attribute generated by the system.
user attribute-list This is a list of permanent attributes.
system attribute-list This is a list of temporary attributes.

1.4. Previous Work

Functions which have tables of values associated with their definitions were proposed by D. Michie, (13,19) and called by him "memo functions". This concept was implemented by R.J. Popplestone, (14,20), and D. Marsh, (15,16), for the POP-2 programming language, (14).

Originally, the idea behind memo functions came from the investigation by A.L. Samuel, (18), into machine learning using the game of checkers. He implemented a rote learning scheme where values associated with the board states encountered during the games were stored on magnetic tape, and then utilized in the evaluation of subsequent states so as to reduce the length of the search required to achieve the same result. States encountered most frequently were credited with extra life - a process called "refreshing" - while those hardly used at all were discarded - "forgetting".

The only published investigation, known to the author, of function evaluation using these concepts is, (20,23). These reports relate to calculations with the following characteristics:
(a) The table of a memo function can only have a fixed number of entries. This limit on the number of entries is specified by the user before the evaluation.

(b) All the function evaluations reported in the investigation have numerical arguments, \{17\}.

(c) According to D. Marsh, \{23\}, the addition of argument result pairs to the table was made at the top level of recursion but the tables were searched at all levels of recursion.

According to these reports, the use of tables of values in connection with the evaluation of functions with numerical arguments leads in most cases to an increase in the speed of evaluation.

There are many differences between the present investigation and those reported above, among them:

(a) the idea of a "memo expression", except that tables of values are associated with expressions which form the arguments of functions. Consequently, the value of a function can be obtained either from a table associated with the definition of the function, or from tables associated with the arguments of the function. This facility is particularly useful in evaluations which are rich in common expressions and sub-expressions,

(b) the investigation is not restricted to functions with numerical arguments; evaluations involving both numerical and structured data arguments have been attempted,

(c) the investigation is not limited to the use of tables of
values with a fixed number of entries. Instead, we begin with a single situation where the user exercises total control over an evaluation and then go on to explore situations in which the user and the system can co-operate in solving a problem, and the sizes of the tables of values vary dynamically depending on the characteristics of the functions and the expressions involved in the evaluation.

1.5. The Thesis

The work reported in this thesis is broken down into seven chapters as follows. A brief description of the programming system used as a basis for each of the programming systems created, and a description of three of the problems used to compare and contrast the performance of the systems implemented, are provided in Chapter 2. Chapter 3 describes and evaluates attribute systems where the user has to create and control attributes. Chapters 4 and 5 describe and evaluate the behaviour of systems where attributes are created and controlled by the system. The provision of facilities for selecting relevant attributes and for classifying attributes is dealt with in Chapters 6 and 7. Chapter 8 summarizes the essence of what has been achieved and makes some suggestions regarding future work.
CHAPTER 2

Background Material

2.1. Introduction

The concepts put forward in Chapter 1 are to be implemented within the domain of symbol manipulation. The term "symbol manipulation" can be interpreted in a variety of ways. Some authors take it as a synonym for "string processing", that is, for processing involving the manipulation of sequences of characters, where each character is normally treated as an individual element. Other authors relate the term to the concept of "list processing", that is to the techniques of processing information stored in a structured form with pointers linking each data item to its neighbours. "Non-numerical computation" and "any manipulations, other than numerical computations, performed on a computer" are yet other meanings given to this term.

Without attempting to define the term rigorously, symbol manipulation is taken throughout this thesis to mean the branch of computing science that is concerned with the processing of data items which exhibit a variable structure. In many commercial and most scientific computer applications, the data to be processed are of known length and format before the computation is performed. In contrast, the extent and the format of the data involved in symbol manipulation are not known in advance and normally vary very significantly during program execution. This point of
view does not contradict the other definitions of symbol manipulation for, clearly, data which are not predetermined in either extent or format require facilities such as those provided in list or string processing systems for their description.

One particular field of application for symbol manipulation systems, which has aroused interest in recent years, is that of algebraic manipulation. Following Sammet, (9), algebraic manipulation is defined as "the computer processing of formal mathematical expressions without any particular concern for their numeric values....". Thus algebraic manipulation is just a particular field of application for symbol manipulation systems where the internal lists or strings are taken to represent formal mathematical expressions.

The host system used for this project, which is described in Section (2.2), shows a distinct bias towards the field of symbol manipulation, and was designed with algebraic manipulation particularly in mind.

The remainder of this Chapter falls into two parts. First, we give a brief description of the host system upon the basis of which the trial systems described in later chapters have been implemented. Some knowledge both of the underlying concepts and of the working of the host system is essential to understanding the work described in this thesis. The account given below provides adequate knowledge of the host system for our present purpose. A reader interested in acquiring a detailed knowledge of the host system is referred
Second, we describe the three problems which are used as a basis to compare and contrast the performances of the systems described in the later chapters.

2.2. The Host System
2.2.1. Introduction

The command-orientated, conversational programming systems used in this project are based upon an extension to the LISP 1.5 programming language, \{1,5\}, developed by F.V. McBride, \{2,3,4\}. The host system was implemented by adding on-line operations and a matching algorithm to a standard LISP 1.5 interpreter, and by introducing a slightly altered syntax of the LISP 1.5 language.

"Minimum assumptions about the manipulations to be performed, and maximum flexibility in the construction of processing facilities" were the basic concepts underlying the design of the host system. This type of system provides only basic processing functions from which the user is expected to build up the capabilities he requires. Such flexibility can not be achieved without some penalty for the user who must pay the price of an increased burden of "awareness". The increase in the user's awareness is justified by the relative ease with which he can create

‡"awareness" - term introduced by Weizenbaum to indicate the attention to detail which must be maintained in any given situation.
facilities which match his particular problem.

The approach adopted, however, contrasts with higher level languages such as FORMAC (28), MATHLAB (29,30), REDUCE (31), and Martin's symbolic mathematical laboratory (21), which provide useful specific capabilities, but do not make allowance for either the definition of radically new capabilities, or the alteration of existing ones in any simple fashion. Indeed, a change to one of these systems which might appear relatively trivial on the surface, will often involve a chain of detailed modifications which only someone who is totally familiar with the system's design features could hope to perform.

There are several reasons for using this particular host system.

1. The time required to construct an alternative symbol manipulation system from scratch is considerable and, if implemented, the alternative may be criticized as being biased toward this project.

2. Many symbol manipulation problems are best solved by a dialogue between the user and the system. The host system which is a command-orientated, conversational programming system satisfies this requirement.

3. The host system is the only significant symbol manipulation system available at The Queen's University of Belfast. Its design has been tailored to the requirements of the local on-line environment.

4. The host system has already been used in various research projects. Its continual use has the
advantage that many of the functions used in this investigation are taken from previous projects.

5. Unlike many programming languages, the host system is capable of processing both structured and unstructured data, of handling both iterative and recursive function definitions, and of performing pattern matching.

The description which follows together with Appendix A adequately describes the essential features of the host system as far as they are relevant to this project. Nearly all of the remainder of this section is devoted to a description of the pattern matching algorithm in the host system. Some illustrative examples and a description of some of the built-in functions appear in Appendix A. The reader is assumed to be familiar with the LISP 1.5 programming language (5).

2.2.2. Description

The host system incorporates three types of capabilities; first, facilities for the definition, and subsequent amendment, of functions needed for a particular evaluation; second, facilities for describing the environment in which the actual evaluation of these functions is to be performed once they are defined; and finally, interactive facilities which permit the user to exert control over the evaluation and its environment while evaluation is in progress. These three components of the programming system will be described in Appendix A without considering details of the implementation of the interpreter for the host language.
The Pattern Matching System

The evaluation process of standard LISP is associated with the functional form:

\[ G(e) = \left( \text{P}_1(e) + \text{S}_1(e); \text{P}_2(e) + \text{S}_2(e); \ldots; \text{P}_n(e) + \text{S}_n(e) \right) \]

where the \( \text{P}_i \) (\( i = 1 \) to \( n \)) are predicates, the \( \text{S}_i \) (\( i = 1 \) to \( n \)) are general expressions and \( e \) is an argument expression. The value of such a function is the value of the substitute \( \text{S}_i \) corresponding to the leftmost true predicate \( \text{P}_i \), encountered by the interpreter (if such a \( \text{P}_i \) exists).

The functional form of the pattern matching system is based on an idea similar to the LISP conditional expressions. In this case the standard form is:

\[ G(e) = \left( \text{f}_1 \text{ matches } e \land \text{P}_1(e) + \text{S}_1(e); \right. \\
\left. \text{f}_2 \text{ matches } e \land \text{P}_2(e) + \text{S}_2(e); \right. \\
\left. \ldots \ldots \ldots \ldots \ldots \ldots \right. \\
\left. \text{f}_n \text{ matches } e \land \text{P}_n(e) + \text{S}_n(e) \right) \]

where the \( \text{f}_i \) (\( i = 1 \) to \( n \)) are dummy forms. The value of such a function is the value of the substitute \( \text{S}_i \) corresponding to the leftmost "successful match" (between the form \( \text{f}_i \) and the argument expression \( e \)) with an associated true predicate.

A Matching Algorithm

In the previous section the term "successful match" was used without giving a formal definition; in this section the concept of "matching" is described.
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 \( f \).

\[ f = 2 \text{ matches only } e = 2 \]
\[ f = + \text{ matches only } e = + \text{ (assuming } + \text{ is a defined function)} \]
\[ f = T \text{ matches only } e = T \]

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

\[ f = x \text{ matches } e = x, y, x + y \text{ and all others} \]

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

\[ f = "x \text{ matches only } e = x \]

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

1) \( e \) must consist of a list of elements \( e_1, e_2, \ldots, e_n \)

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

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

\[ f = (u \ v \ w) \text{ matches } e = (x \ y \ z) \]

but \[ f = (u \ u) \text{ does not match } e = (x \ y) \]

however \[ f = (u \ v \ w) \text{ matches } e = (x \ x \ x) \]

RULEs and EXPRs

LISP functions which are defined by the pseudo function
define have an EXPR indicator on their property lists. To facilitate the use of the matching algorithm, a new type of function has been introduced into the LISP interpreter, characterized by the indicator RULE on the property list of an atom. Its S-expression definition has the form:

\[(DARG \ (D1 \ D2 \ ... \ DN))\]

\[\text{ASSERTION1}\]
\[\text{ASSERTION2}\]
\[\ldots\]
\[\text{ASSERTIONM}\)

where the atom DARG serves a similar purpose for RULE's as LAMBDA for EXPR's and \((D1 \ ... \ DN)\) is a list of dummy arguments. An "assertion" consists of two parts, a label and a body. Assertion labels are present primarily to provide a mechanism for amendment. The format of the body of an assertion is:

\[(\text{form}; \ \text{substitute}; \ \text{predicate})\]

that is a three element list. The predicate has been positioned at the end of the list so that a two element list can be taken to denote an implicitly true predicate.

The matching process consists of comparing the argument expression with the form of each assertion taken in sequence, according to the algorithm described earlier. If a successful match is obtained the form elements are bound to corresponding elements in the argument expression and the bindings will be appended to an association list (a-list).
\((\ast\ A\ B)\) 
\((+(\ast\ A\ C)(\ast\ T\ Z))\) match? 

If not, then why is binding \((+.+)\) produced?
e.g.: a form \((+ A B)\) matches an expression
\[ (+(* 4 C)(+ T 2)) \]
and produces bindings
\[ (+A, (+ T 2)) \]
and \((B.(* 4 C))\).

Then, if the associated predicate evaluated with respect to the newly created a-list is true, the value of the rule is the corresponding substitute evaluated with respect to the a-list. Either of these evaluations may involve re-entry into the matching system.

For an illustration of the matching system see Section \(\lambda 2\).

An Extended Matching System

The matching system described above employs a left-to-right, one-to-one, matching algorithm. In certain contexts, predominantly those involving arithmetic operations, this system requires the definition of a substantial set of assertions for solving a simple matching problem. In order to overcome this difficulty facilities are provided to inform the system of the axiomatic properties of operators, for example, the commutativity of the operators \(+\) and \(*\), or the equivalence of \(A, (+ A 0), (* A 1)\). This has been achieved by the introduction of a mechanism to reconstitute the argument expression in an equivalent form.

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.
Transformations

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

\[ \text{label(form; substitute; predicate)} \]

but the evaluation process is somewhat different. The object of applying a transformation to an argument expression is to obtain an equivalent expression; thus after the form-match and the predicate testing have been performed as they were 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 can be written as:

\[ t_1 : +(a;b)\rightarrow +(b;a) \]

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

\[ t_2 : a\rightarrow +(a;0) \]

then \(t_1\) applied to an S-expression \((+ x y)\) would yield the reconstitution \((+ y x)\) and \(t_2\) applied to the same expression would give \((+(+ x y)0)\).

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 Section (A2).

Modified Form of Assertions

A fourth element called a transformation list or t-list, is added to the assertion 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 formats for assertions.

**Operation of Extended Matching System**

The general strategy which the matching system employs for processing "transformational" information is as follows: as before the left-to-right matching algorithm described in Section (A.2.1) is used; if the match fails, however, an attempt is made to reconstitute the argument sub-expression at the level of failure by applying from the t-list element of the assertion under consideration those transformations whose labels are associated with the RULE governing* the corresponding sub-form. If no suitable reconstitution can be found, the process returns to the previous higher structural level and searches for an alternative match using those transformations 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. Section (A.2.2) presents an illustration of the extended matching system.

* A RULE with the name r is said to "govern" a form f, if f is a list whose first element is r.
Facilities of the Programming System

The programming system provides the user with the following capabilities:

(a) to define and amend functions,
(b) to define local and global variables,
(c) to interact with the system,
(d) to communicate with the backing store.

Section (A.3) is devoted to the description of these facilities.

2.3. Some Typical Problems

Two methods can be used for testing the systems implemented in this project and both of them are open to criticism. The first is to select one or two complete problems of real significance and to compare system performance for these problems. The choice of one or two realistic problems can be criticised for its lack of flexibility and the cost, in terms of computer resources, of providing meaningful practical results on which to base the comparison.

The second approach is to select a number of small, artificial problems and to compare system performance for these problems. The choice of a number of small problems, although it corrects the defects in the first method, may be criticized for its artificial nature. In fact, the second method was adopted and some of the reasons for this choice are given below.

In order to be able to attempt a completely realistic
problem in the area of symbol manipulation, the system developed must be equipped with all the particular facilities required by the specific problem. For the purpose of this project the problem selected must utilize the various new features that have been developed. Selection of the problem to match these requirements is likely to create the impression that the results of the practical work apply only to this particular problem. It is also possible that the details involved in the statement, and the solution, of an actual problem may obscure the nature of the features which are under investigation. Another difficulty which limits the choice of realistic problems is to find problems whose degree of complexity can be varied for the purpose of comparing the performance of systems under a variety of conditions. With an artificial problem the degree of complexity can be varied by adjusting the parameters of the problem.

The limitations in the resources provided by the computer installation which is used cannot be ignored. For example, under the operating system available the free word list can only contain a maximum of 10,000 cells when programs are run off-line and 5,000 cells when programs are run on-line. This storage space must hold all the function definitions and provide an adequate working space. The small size of the storage space available has also influenced the choice of problems.

A real problem can be viewed as a collection of simpler sub-problems. According to this view the overall gain in efficiency is related to the individual gains in the efficiency of the constituent sub-problems. In this way, although the
sub-problems chosen to demonstrate the performance of the systems described may seem artificial, in most cases they can be regarded simply as constituent of a more complex problem. For example, differentiation and integration of expressions can be regarded as sub-problems in a system designed to solve differential equations (6), to perform symbolic integration (7), or to carry out manipulations in the fields of theoretical physics (8) and general relativity (11).

Below we describe those problems whose solutions are used in testing the systems developed in later chapters. Two of the problems described below make use of the differentiation function whose LISP M- expression definition is given in Appendix E. Use of this example has a number of advantages, among them:

(a) the reader is likely to be familiar with the working of the function and the complexity of a problem can be varied widely by selecting suitable argument expressions,
(b) higher order differentiation, which involves the recursive use of the same function, provides us with evaluations of varying degrees of complexity,
(c) this function can be coded in all contemporary symbol manipulation languages,
(d) it is a compact and a well-defined function which does not involve the definition of other subsidiary functions.

The problems described are not adequate for testing all features of the systems implemented. Other problems are introduced and used as examples where the need arises.
\[
\left( \frac{\text{something}}{} \right) \quad \text{and} \quad \left( \frac{\text{something else}}{\text{something more}} \right)
\]
2.3.1. Multiple Differentiation

The first problem involves computing

\[ \frac{d^p}{dx^p} (a + (x - b)^n)^m \]

where, \( a \) and \( b \) are constants; \( n \) and \( m \) are integers and \( p \) takes the values 1, 2, ..., 9. The evaluation of a range of the higher order derivatives is useful in that it creates evaluations of increasing complexity.

In our subsequent analysis we need to know the the variation of the evaluation time and the number of garbage collections, with respect to the order of differentiation. The curves provided in Figs. 2.1 and 2.2 supply this information for the host system with a free word list consisting of 7,000 free cells. In the later chapters references are made to these curves.

2.3.2. Third Order Differentiation

The second problem is the computation of

\[ \frac{d^3}{dx^3} \{ (x^2-x+1)^i (2+\log(x^2+4x-1))^j (3+\exp(3x^2-x-1))^k \}^l \]

where \( i, j, k, l, m \) and \( n \) are all integers. This third order differentiation is considerably more complex than the first problem, and is used throughout to compare and contrast systems including attributes with each other, and with the host system. From the form of the expression differentiated it is possible to identify some of the repetitive evaluations which have to be performed, namely:
Fig. 2.1. Curve shows the variation of the number of garbage collections required with respect to the order of differentiation for the differentiation I problem in the host system.
Fig. 2.2. Curve shows the variation of evaluation time with respect to the order of differentiation for the differentiation I problem in the host system.
This problem is used to test almost all the features which are described in the later chapters, and from this point of view it is the most significant problem considered.

The two curves given in Figs. 2.3 and 2.4 represent the variation of evaluation time and the number of garbage collections required, with respect to the storage space available when this problem is solved using the host system. These two curves are referred to in subsequent discussions.

2.3.3. A Function with Numerical Arguments

prs(n) is a function which takes numerical arguments. Given a positive integer n, this function produces a list of all the prime numbers in the range from 0 up to n as its value. All the tests were carried out using 500 random numbers in the range 0<n<100, whose values are listed in section (B.4).

The M-expression definitions of prs, which is defined both recursively and iteratively, are given in Section (E.8).
Fig. 2.3. Curve shows the variation of the number of garbage collections required with respect to the storage space available for the differentiation 2 problem in the host system.
Fig. 2.4. Curve shows the variation of time of evaluation with respect to the storage space available for the differentiation 2 problem in the host system.
The two families of curves which are provided in Figs. 2.5 and 2.6 show how the evaluation time and the number of garbage collections required vary with respect to the storage space available for the evaluation of the recursive and the iterative definitions of prs by the host system. These families of curves are referred to in later chapters.
Fig. 2.5. Curves show the variation of the number of garbage collections required with respect to the storage space available for the iterative and recursive definitions of prs in the host system.
Fig. 2.6. Curves show the variation of evaluation time with respect to the storage space available for the iterative and recursive definitions of prs in the host system.
3.1. Introduction

This chapter describes two symbol manipulation systems that incorporate the idea of an attribute-list as described in Section (1.3.3). Attribute-lists can be associated either with functions or with the expressions which make up their arguments. The first system described below is an "Expression Driven System", EDS, in which all attribute-lists are associated with expressions. The other system described in this chapter is a "Function Driven System", FDS, in which all attribute-lists are associated with functions. Apart from this difference, the two systems provide similar facilities. While it would have been possible to construct one system incorporating all of the facilities, the two separate systems are a more convenient basis for making comparative studies of performance of the type described later in the chapter.

The expressions and functions which are to possess attribute-lists are selected by the user, who is also responsible for providing the attributes and performing a variety of "house-keeping tasks", such as organising the entries in the attribute-lists so that they can be accessed efficiently. Subsequently, the use of these permanent attributes makes the process of evaluation more efficient.

After introducing the basic terminology, used to describe the facilities provided, a brief description of the most
important features of each system is given. This is followed by a detailed specification of the facilities implemented in each system.

An examination of some typical evaluations made using these systems enables us to identify the effects which the provision of such facilities has on evaluations performed in symbol manipulation. Use of an attribute-list leads to some reduction in the time of evaluation, and the number of garbage collections required to perform an evaluation. Further, under the same conditions, these systems may make possible the solution of problems which could not be solved in the host system. The reasons for this increase in efficiency of evaluation are investigated.

3.2. Basic Concepts

In Chapter 1 we put forward the set of ideas which underline the entire thesis (Section 1.3). The purpose of this section is to elaborate on those parts of these ideas which are concerned with the two programming systems presented in this chapter.

In a computation, when a user knows that a particular function is evaluated a large number of times with each of a small number of arguments, he may decide to use a table of function values to save the time involved in re-evaluating the function. In order to make use of this technique, appropriate changes are made to the definition of the function. Any subsequent alterations to the range of arguments used by the function may mean that the definition of the function has to be modified and re-tested.
The two programming systems which are described in this chapter provide a mechanism for dealing with repetitive function evaluations by allowing the user to designate functions and arguments whose values are to be saved so as to avoid repetitive evaluation. An advantage of these systems is that they allow a general function definition to be constructed, which is capable of being evaluated with a full range of arguments. Later, when the actual data which forms the arguments is specified, then the user can identify to the system those functions and arguments whose values are to be saved. The system constructs appropriate tables of values and modifies the evaluation mechanism for these functions so as to take advantage of any tabulated values in order to reduce the labour of re-evaluation. The user can provide the system with the additional information throughout an evaluation, thus altering the mechanism of function evaluation without altering the function definition. This last facility is particularly useful for situations in which the occurrence of repetitive function evaluations depends on the nature of the data being processed, and consequently can only be identified when evaluation is in progress.

3.3. The expression driven system

3.3.1. Basic features

The concept of an attribute-list was described in Chapter 1 (section 1.3.3). In the system described in this section attribute-lists are associated with particular expressions. The system leaves the user to exert total control over the generation of attribute-lists, and, as a result, places a considerable burden on him. This is because
the user must identify all the expressions which are to have attribute-lists associated with them, ensure the creation of all the attributes which are needed, and look after attribute house-keeping operations.

Before describing the system we must define the terminology used. The appropriate terms and their corresponding definitions are listed below:

**Object** An atomic symbol with the property indicator IDEN appearing on its property list is called an object. The two functions which append the property indicator IDEN to the property list of an atomic symbol are `ident` and `identq` whose descriptions appear in section (3.5.1). An atomic symbol with the property IDEN is also referred to as an object name.

**Natural-value** The expression assigned as the value of an atomic symbol, (which is to act as the object name), using one of the two functions `ident` and `identq` is referred to as the natural-value of the object. Natural-values are also called "declared expressions".

**Object-value** An object-value is the address of the cell which contains as its **car** a pointer to the natural-value of the object, and as its **cdr** a pointer to the attribute-list of the object. A null attribute-list is assumed if **cdr** is **NIL**.

**Attribute-indicator** An atomic symbol which occurs on an attribute-list, and specifies that the associated item on the list is an attribute-list corresponding to the specified property, for example, differentiation denoted by D is a typical attribute-indicator.
**Derived-value** This is the value of a function corresponding to a specified argument.

Having introduced the terminology we can describe the internal and external representations of attributes and attribute-lists. Since the host programming language used is LISP 1.5, all non-atomic quantities are internally represented by binary tree structures. In order to achieve a natural written form of the external representations, we decided against the use of the M-expression and S-expression notations, (which are commonly used in LISP), and adopt the external and internal representations described below by means of an example.

An object EXP has natural-value \( ax + b \) so that

\[
\text{EXP} = ax + b
\]

If EXP has the following attributes:

\[
\frac{d}{dx} (ax + b) = a
\]

where \( a \) and \( b \) are constants*

\[
\frac{d}{dk} (ax + b) = 0
\]

where \( a, x \) and \( b \) are independent of variable \( k \).

* In the host system \( a \) and \( b \) cannot be declared as constants using a type statement. Instead, during the process of evaluation the predicate function \texttt{free}, whose description appears in Section (C.5), examines the implicit and explicit dependencies of its first argument on its second argument. If the predicate function returns \( T \) as the value, its first argument is regarded as a constant term.
Then the attribute-list associated with $ax + b$ can be written in the following external representation:

$$(ax + b) \dfrac{\text{d}}{\text{d}x} (ax + b) = a; \dfrac{\text{d}}{\text{d}k} (ax + b) = 0;$$

$$f(ax + b) \text{ d}x = \frac{a}{2} x^2 + bx ;$$

$$\text{linear (x; } ax + b) = T \}$$

The internal representation of this symbolic form is shown in the diagram below.

The meanings of the symbols used in this diagram and in the corresponding external representation above are given in Section (B.1).

---

* constant of integration not included.
3.3.2. The object-stack

One of the basic difficulties to be surmounted in attaching an attribute-list to an expression is to maintain the connection between the natural-value and the attribute-list. This difficulty arises because in the host language (LISP 1.5) the basic data structure is a binary tree represented as a unidirectional list. This means that it is not possible to access the main list from one of its sub-lists. The interpretation of this fact in terms of the diagram:

When access is here

\[ \text{object-value} \]

\[ \text{attribute-list} \]

\[ \text{natural-value} \]

is that if, for example, we are currently accessing the natural-value of an object, it is not possible to recover its object-value and, as a result, its attribute-list.

To avoid the complications inherent in symmetric structures, the systems implemented maintain the link between natural-values and their attribute-lists by using an auxiliary stack called the "object-stack". As new objects are defined, or as objects are brought into core from backing store, (Section A.3.5), an entry is made in the object-stack. This entry is a reference to the natural-value of the object, and this system of back referencing enables the system to
maintain the link between the natural-value and its attribute-list at all times. A diagrammatic representation of this scheme is given below:

As long as we have access either to the natural-value or to the object name of a particular object, we can gain access to its attribute-list.

As mentioned above, entries in the object-stack are built up in sequence as objects are defined by the user or as they are brought into core from the backing store. The entries in the stack are never deleted by the system. This is to allow for the fact that at some later stage in the evaluation, the user may use the name of a previously defined object as part of another manipulation. The only time entries are removed from the object-stack is when an expression is deleted from a group of declared expressions by the use of the command remprop, (Section 3.5.1); the space released in this way can be reused for subsequent entries.
A problem which had to be resolved was the choice between a variable size or a fixed size object-stack. Although the use of a static stack is in a sense contrary to the principle of dynamic storage allocation which is so basic to symbol manipulation systems, its use does have one advantage - direct access to the object-value can be obtained without processing a linked structure. The disadvantage is that the system cannot make use of all the available entries which are not being used in order to extend its working space.

Even with only a few entries in a variable size object-stack, the processing of the linked structure creates an overhead on search time. Further, since in our case, two words of storage space are required for every entry in the variable size object-stack, it is questionable whether there would have been any saving in storage space. So we choose a fixed size object-stack.

3.3.3. Operation of the interpreter

In this section we concentrate on those modifications to the interpreter which are concerned with the mechanism of function evaluation by searching an attribute-list, and the subsequent actions depending on the outcome of this search, (see Section C.1 for changes in definitions of eval and apply). The appropriate sequence of events is summarized in the flow-chart in Fig. 3.1.

As long as the function being evaluated contains only one object among its argument expressions, that is, one
Fig. 3.1 Flow-chart shows the modifications which had to be made to the interpreter of the host system in order to make it workable for the expression driven system with permanent attributes.

Fig. 3.2 Flow-chart shows the modifications which had to be made to the interpreter of the host system in order to make it workable for the expression driven system with permanent attributes.
expression which is referencing the object-stack, the evaluation is straight-forward: the appropriate attribute-list is searched and if this search is successful the derived-value is returned directly from the attribute-list, otherwise the function is evaluated from its definition. If there is more than one object in an argument expression then evaluation through the function definition is only attempted after an unsuccessful search of each of the attribute-lists involved. This process is inefficient because of the long look-up times involved. A simple constraint which eliminates this draw-back is to search the attribute-list of only one of the objects in the argument list. In the system implemented this constraint is applied by searching only the attribute-list of the last element of the argument-list which must be an object. The choice of the last element in the argument-list permits efficient evaluation of functions of the type rule, which are fully described in Section (2.2.2). This restriction does not seem to present any practical problems.

Essentially, the flow-chart fragment given in Fig. 3.1 replaces the dashed rectangular box given in the flow-chart of apply provided in Section (A.1). It should be pointed out that the majority of the changes to the LISP interpreter concern minor modifications to many of the service functions in the interpreter, and not only to the function apply. These small changes are made at machine code level, and a description of the details is not essential to an understanding of this thesis. The labour involved in the implementation of the system should not be under-estimated.
due to the omission of a detailed account of the implementation of the modifications which were made to the LISP interpreter.

In comparison with the host system, the complete set of modifications increases the storage space required by the interpreter by 1632 words.

Example

The best way to demonstrate the operation of evaluation in the presence of attributes is to go through the process of evaluation step by step. The example chosen for this purpose is:

$$\frac{d^2}{dx^2} (a + (x - b)^n)^m$$

where, $a$ and $b$ are constants and $m$ and $n$ are integers.

It is clear that during the evaluation, among the expressions which will be differentiated more than once with respect to $x$ are:

$$(a + (x - b)^n); (x - b)$$

Consequently, we create two objects $u$ and $v$ with the natural-values $(a + (x - b)^n)$ and $(x - b)$ respectively. To avoid these two expressions being repeatedly differentiated with respect to $x$ we create attributes:

$$\frac{d}{dx} (a + (x - b)^n) = n(x - b)^{n-1}$$

$$\frac{d}{dx} (x - b) = 1$$
At this stage the object-values of the two objects u and v have the form:

\[ u = \{ (a + (x - b)^n)^m \} = n (x - b)^{n-1} \]

and

\[ v = \{ (x - b) + \frac{d}{dx} (x - b) = 1 \} \]

respectively. A reader interested in the set of commands needed to create these attribute-lists is referred to Section (B.3).

Having created these objects, the process of differentiation is carried out according to the set of rules listed in Section (B.2.1). The steps of evaluation are annotated using the notation described in Section (B.2.2). The steps in the evaluation are as follows:

1. \[
\frac{d}{dx} (a + (x - b)^n)^m = m (a + (x - b)^n)^{m-1} \quad \text{(D6)}
\]

2. \[
\frac{d}{dx} (a + (x - b)^n)^m = mn (a + (x - b)^n)^{m-1} (x - b)^{n-1} \quad \text{(SUBS)}
\]

- 42 -
E2. \[ \frac{d}{dx}(mn(a + (x - b)^n)^{m-1}(x - b)^{n-1}) = \] (D4) 

\[ mn \frac{d}{dx}(a + (x - b)^n)^{m-1}(x - b)^{n-1} \] 

E3 

\[ + (a + (x - b)^n)^{m-1}(x - b)^{n-1} \frac{d}{dx}(mn) \] 

E4 

E5 

E3' \[ 55{(a + <x \_ b)^n)^{m-1}(x - b)^{n-1} = \] (D4) 

(a + (x - b)^n)^{m-1} \frac{d}{dx}(x - b)^{n-1} 

E5 

\[ + (x - b)^{n-1} \frac{d}{dx}((a + (x - b)^n)^{m-1}) \] 

E6 

E5 \[ \frac{d}{dx}(x - b)^{n-1} = (n - l)(x - b)^{n-2} \frac{d}{dx}(x - b) \] (D6) 

\[ \frac{d}{dx}(x - b) + 1 \] (AL) 

E5. \[ \frac{d}{dx}(x - b)^{n-1} = (n - l)(x - b)^{n-2} \] (SUBS) 

E6. \[ \frac{d}{dx}((a + (x - b)^n)^{m-1}) = (m - l)(a + (x - b)^n)^{m-2} \] (D6) 

\[ \frac{d}{dx}((a + (x - b)^n)) \] 

\[ \frac{d}{dx}((a + (x - b)^n)) + n(x - b)^{n-1} \] (AL) 

E6. \[ \frac{d}{dx}((a + (x - b)^n)^{m-1}) = n(m - l)(a + (x - b)^n)^{m-2} \] (SUBS) 

(x - b)^{n-1}
Notice that three times in the course of evaluation the required values are obtained from attribute-lists. The trace of a second order differentiation:

\[
\frac{d^2}{dx^2} (a + (x - b)^n)^m
\]

as evaluated by the host system and expression driven system with permanent attributes appears in section (B.3). A comparison of these two traces shows that the presence of the three attributes reduces the total number of entries to the differentiation function by 12.

3.4. The function driven system

3.4.1. Basic features

In this system expressions are still named using the two functions \texttt{ident} and \texttt{identq}, but they do not have attribute-lists associated with them. Instead, in this system, the user selects the functions which are to have
attribute-lists associated with them.

The function driven system is likely to place less burden on the user than the expression driven system because the user is more likely to be aware of some (or all) of the functions in common use during an evaluation (Section 5.2.2).

The association of an attribute-list with a function means that a link must be established between functions and their attribute-lists, so that by having access to a function we can obtain access to its attributes. This is a simple matter due to the way in which the data are structured and used within LISP 1.5. In this case the attribute-list is attached to the property list of the function as the value of the property indicator ROTE. To demonstrate this organisation consider memo functions $d$, $f$ and linear with attributes:

$$\frac{d}{dx}(ax + b) = a; \frac{d}{dk}(ax + b) = 0$$

where $a$, $b$ are constants and $x$ and $k$ are independent variables;

$$f(ax + b)dx = \frac{a}{2}x^2 + bx$$

where the constant of integration is not included and

$$\text{linear} \{x; (ax + b)\} = T$$

The external representations of the attribute-lists associated with $d$, $f$ and linear are:

$$d \left\{ \frac{d}{dx}(ax + b) = a; \frac{d}{dk}(ax + b) = 0 \right\}$$

$$f \left\{ f(ax + b)dx = \frac{a}{2}x^2 + bx \right\}$$

$$\text{linear} \left\{ \text{linear} \{x; (ax + b)\} = T \right\}$$

respectively. The corresponding symbolic representations of
the internal representation are given below.

The meaning of the symbols used in these diagrams are given in Section (B.1). It is clear from the above diagrams that the expression \((ax + b)\) is stored on four separate occasions compared to its single occurrence if the same attributes are created in an expression driven system, as shown clearly by the diagram given in Section (3.3.1).

3.4.2. Operation of the interpreter

As the flow-chart (Fig. 3.2) shows, the modifications that are made to the interpreter of the host system in order
to implement the function driven system with permanent attributes, closely resemble those needed to implement the expression driven system with permanent attributes (Section 3.3.3).

The flow-chart fragment in Fig. 3.2 replaces the dashed rectangular box in the flow-chart of apply provided in Section (A.1). Once again, we have chosen not to describe the many small modifications made to the service functions of the interpreter, since they are not essential to understanding the process of evaluation. The modifications increased the size of the interpreter of the host system by 1412 words. Further, since the process of evaluation is identical to that given in an expression driven system (Section 3.3.3), no illustrative example is given at this point.

An attribute is an argument result-pair. The internal representations of attributes in the expression and the function driven systems, (given in Sections 3.3.1 and 3.4.1), mean that the argument component of an attribute is more complex in the function driven system than in the expression driven system. This is because in the expression driven system, the argument component of an attribute does not include the natural-value of the argument, while in the function driven system it does.

When searching an attribute-list, the current argument of the function being evaluated is compared against the argument component of the attributes. The search is successful if an attribute with an argument component identical to the current argument list is found. Since the argument component of the
attributes can be a complex expression, to attempt a direct expression match every time is inefficient. In view of the remarks above, this effect is a more serious problem in the function driven system where the argument component of attributes is more complex.

To make the searching of attribute-lists more efficient, we arrange for the search to proceed on two levels. Each argument component of an attribute has associated with it a value to specify the number of atomic symbols it contains. At the start of the match, the number of atomic symbols in the current argument expression of the function is evaluated. The search mechanism subsequently matches expressions only when both contain the same number of atomic symbols.

In cases where all the argument components of functions and their attributes have the same number of atomic symbols, for example, a function with an integer argument, the first stage of the search leads to a further increase in look-up time.

3.5. Repertoire of Commands

This section describes the set of commands that a user can use to define the process of evaluation in the expression and function driven systems with permanent attributes. As indicated in Sections (3.3.1 and 3.4.1), due to the different internal structures adopted in organising the attribute-lists in the expression and function driven systems, a command performing the similar task has different definitions in the two systems. All the commands which are described below are hand coded into a LISP 1.5 interpreter. Where indicated
below, the M-expression definitions of these commands together with examples of their use appear in Appendices B and C.

3.5.1. Repertoire of commands : EDS

ident {argl; arg2} :SUBR pseudo function

In the expression driven system this command is used to identify those expressions which are to have attribute-lists associated with them. The function ident is a pseudo function in the LISP 1.5 sense, that is, it is a function whose execution affects the state of the system in addition to producing a value for use in subsequent evaluations. argl which is referred to as the object name, must be an atomic symbol. arg2 referred to as the natural-value of the object identified by argl in the expression which is associated with an attribute-list. The effect of executing this command is to place the property IDEN on the property list of argl and to insert an entry on the object-stack (Section 3.3.2).

An expression which has appeared as arg2 in an ident evaluation is called a "declared expression", and is represented diagrammatically as follows:

![Diagram of ident command]

argl

ident

attribute-list currently null

arg2
(natural-value)
where the meaning of the symbols used in the diagram is
given in Section (B.1). No error messages are generated and
no value is returned. Examples of the use of this function
appear in Section (B.6), and its M-expression definition
appears in Section (C.2).

identq \{arg1; arg2\} :SUBR pseudo function

The action of identq is identical to that of ident
except that the second argument is evaluated before the IDEN
property is set up. The result of evaluating the expression
arg2 is used as the natural-value of the object, and is
returned as the value of this function. No error messages
are generated. An example of its use and its M-expression
definition appear in Section (C.2).

remprop \{x\} :SUBR pseudo function

Occasionally, the need for removing a named expression
from the class of declared expressions arises. remprop is
used for this purpose. It takes a list of pair-lists as its
only argument. The first element of each pair-list is the
name of a declared expression which is to be deleted, while
the second element is the property indicator IDEN. The user
has to specify the property involved because this function
can be used to delete other properties. No error messages
are generated and no value is returned.

attrib \{x\} :SUBR pseudo function

This function is used to create permanent attributes.
Its argument is a list of pair-lists, that is:
These pair-lists are processed from left to right and each of them will give rise to one attribute. $U_i$ is also a list, the first element of which is the name of a function and its remaining elements are its arguments, for example

$$U_i = (f \, x_1 \, x_2 \, \ldots \, x_{n-1} \, \text{obj})$$

As specified in Section (3.3.3), the last element in the argument list of $f$ must be the name of an object which has already been created. As each pair-list is processed two conditions are checked:

(a) if the last element in the argument list of $f$ is not the name of an existing object, the user is informed by an appropriate message,

(b) if the attribute which is to be created already exists, the system proceeds to process the next pair-list.

The provision of $V_i$, which is normally the value of the attribute as given by evaluating $U_i$, is optional. If $V_i$ is NIL then the system evaluates $U_i$ to obtain the derived-value before constructing the attribute. The facility to specify the derived-value directly is particularly useful where the system is not capable of handling the definition of the function $f$, or the user decides not to include its definition in the system. If the user provides neither the derived-value nor the definition of $f$ then an error message is generated.

When creating attributes, two possibilities arise. If the name of the function $f$, already exists as an attribute-
indicator on the attribute-list of the object \texttt{obj}, then the structure below

\[
\left( x_1 x_2 \ldots x_{n-1}.f(x_1 x_2 \ldots x_{n-1} \texttt{obj}) \right)
\]

is created and appended to the attribute-list of \texttt{obj}. Otherwise an attribute-indicator \( f \) is created and the structure:

\[
\left( (x_1 x_2 \ldots x_{n-1}).f(x_1 x_2 \ldots x_{n-1} \texttt{obj}) \right)
\]

is appended to the attribute-list of \texttt{obj}. No value is returned. Examples of its use appear in section (B.6) and its M-expression definition appears in Section (C.3).

\texttt{destatts \{x\} :SUBR \ pseudo function}

This function is used to remove unwanted attributes from attribute-lists. \texttt{destatts} takes only one argument which is a list. Each element of this list, (which is processed from left to right), falls into one of the three categories specified below,

(a) Object name

In this case the entire attribute-list of the
specified object is replaced by a null attribute-list, thus destroying all the attributes.

(b) \((\text{Object name}; \text{attribute-indicator})\)

In this instance the attribute-indicator and its corresponding list of attributes are deleted from the attribute-list of the given object.

(c) \((f \ x_1 \ x_2 \ldots \ x_{n-1} \ \text{obj})\)

The particular attribute specified is deleted from the attribute-list of the object \text{obj}. An error message is generated if the particular attribute, attribute-indicator or object does not exist. No value is returned. An illustrative example and the M-expression definition of \text{destatts} appear in Section (C.3).

\[
\text{dispatts} \ (x) \quad :\text{SUBR pseudo function}
\]

This command is used to display part or all of an attribute-list. The argument, \(x\), has the same meaning as for \text{destatts}. The three possible actions are as follows:

(a) Object name

This causes the natural-value and all the attributes of the given object to be printed.

(b) \((\text{Object name}; \text{attribute-indicator})\)

This causes the natural-value of the object, together with all the attributes associated with the given attribute-indicator, to be printed.

(c) \((f \ x_1 \ x_2 \ldots \ x_{n-1} \ \text{obj})\)

The natural-value of the specified object and the specified attribute are printed. When an invalid object-name or
attribute-indicator is specified an error message is generated. An example of its use and its M-expression definition appear in Section (C.3).

swopatts {x} :SUBR pseudo function

This function is used to alter the position of an attribute or a group of attributes in an attribute-list. In this way a user can promote or demote the attributes according to their frequency of use. The elements of the argument list, x, are evaluated from left to right and fall into one of the two categories specified below.

(a) (object-name; attribute-indicator 1, attribute-indicator 2)

In this case the positions of the two attribute-indicators, and their associated values, are interchanged on the attribute-list. The user is informed of the absence of either the given object or of the specified attribute-indicators by an error message.

(b) ((f x_1 x_2 x_{n-1} obj) (f x'_1 x'_2 x'_{n-1} obj))

In this case the positions of the two attributes specified are interchanged. In the given pair-list the attribute-indicators (function names) and the object names must be the same. A user will be informed by an error message if:

- either the two attribute-indicators are different,
- or the two object names are different,
- or the specified obj does not exist.

No value is returned. An illustrative example and the M-expression definition of this function appear in Section (C.3).
3.5.2. Repertoire of commands : FDS

memo \{x\} :SUBR pseudo function

memo takes as its argument a set of function names which are to be associated with attribute-lists. Its effect is to identify memo functions by putting the property indicator ROTE and a null attribute-list on the property lists of the specified functions. No error messages are generated and no value is returned. Examples of the use of this function appear in Section (B.6) and its M-expression definition appears in Section (C.2).

remprop \{x\} :SUBR pseudo function

Occasionally the need for removing a function name from the class of memo functions may arise, remprop is used for this purpose. It takes a list of pair-lists as its only argument. The first element of each pair-list is the name of the memo function which is to be taken out of the class of memo functions, while the second element specifies the property to be deleted, that is, it is the property indicator ROTE. No error message is generated and no value is returned.

attrib \{x\} :SUBR pseudo function

This function is used to create attributes. Its only argument is a pair-list of the form:

\[((U_1 \ V_1)(U_2 \ V_2) \quad (U_i \ V_i) \quad (U_n \ V_n))\]

where \(V_i\) has the same meaning as specified for the command with the same name in the expression driven system, and \(U_i\) has the form

\[U_i = (f \ x_1 x_2 \ldots x_n)\]
The user will be informed if \( f \) is not a memo function and no attribute is created. Otherwise, in the absence of the required attribute on the attribute-list of \( f \), the structure

\[ ((x_1 x_2 x_n).f(x_1 x_2 x_n)) \]

is created and appended to the attribute-list of the memo function \( f \). No value is returned. Some instances of the use of this function and its M-expression definition appear in Sections (B.6) and (C.3), respectively.

\textbf{destatts} \( \{x\} \) :SUBR pseudo function

This function enables the user to delete unwanted attributes. Each element of the list \( x \) has the form

\[ (f x_1 x_2 x_n) \]

where \( f \) is a memo function. If the specified attribute exists, it will be deleted from the attribute-list of the specified memo function. The user is informed if the specified attribute does not exist. No value is returned. An example of the use of this function and its M-expression definition appear in Section (C.3).

\textbf{dispatts} \( \{x\} \) :SUBR pseudo function

This command is used to display either individual attributes or the entire attribute-list of memo functions. For those elements of the list \( x \) which are the names of memo
functions, the entire attribute-list is displayed. Otherwise the specified attribute, for example,

\[(f \ x_1 \ x_2 \ldots \ x_n)\]

is displayed. The user is informed if \( f \) is not a memo function. An example of the use of this function and its M-expression definition appear in Section (C.3).

\texttt{swopatts \{x\}} \\
\texttt{:SUBR pseudo function}

This function enables the user to rearrange the attributes on an attribute-list. The argument expression \( x \) is a pair-list of the form

\[((f \ x_1 \ x_2 \ldots \ x_n)(f \ x'_1 \ x'_2 \ldots \ x'_n))\]

If either the names of the two memo functions are different, or if either of the specified attributes is not defined or the function is not a memo function, the user is informed by an error message. Otherwise the positions of the attributes specified are interchanged on the attribute-list. No value is returned. An example of the use of this function and its M-expression definition appear in Section (C.3).

3.6. Analysis of Practical Results

3.6.1. Introduction

In this section we use three problems, (whose descriptions and a measure of their performance in the host system appear in Section 2.3), to demonstrate some properties of expression and function driven systems with permanent attributes. As pointed out in Chapter 1, we concentrate on three aspects of the operation of these systems. They are: the speed of evaluation, the use of storage space during
evaluation, and the burden which the use of a system with attributes imposes on the user.

Our estimates of the speed of evaluation and the use of storage space are based on comparisons between the time taken and the number of garbage collections, respectively, as performed in order to carry out a particular manipulation in the host system and in the system with permanent attributes under the same conditions, (that is with a free word list consisting of the same number of free cells).

A meaningful way of measuring the burden on the user in using the systems with permanent attributes is not easy to find. In this case, our comparisons are based on the awareness that the user must have of his problem, the effort in learning to use the systems with attributes, and the commands which must be explicitly written by the user in order to perform the evaluations.

Our comparisons of the speed of evaluation and the use of storage space are based on performing the same evaluation with varying amounts of free storage space. Our findings are reported in the form of graphs. The graphs appearing in this section are often based upon a small number of points in order to reduce to a minimum the cost of this comparative study. The data used in plotting the necessary graphs and for comparing the performance of the various systems are taken from the tables of Section (B.5).

Among these examples there are instances where the use of a system with attributes produces a substantial effect on the components of efficiency mentioned above. An attempt has been
made to explain these effects by relating them to the characteristics of the system involved.

Before looking at particular examples, let us consider the relation between evaluation time and the storage space available on the free list. Normally, if the free working space of a problem is restricted to \( M \) cells, then these \( M \) cells serve as the working space throughout the process of evaluation. In a system with permanent attributes, a portion of this working space will be permanently occupied by attribute-lists, \( m \) cells say, and this reduces the effective size of the free working space for evaluation to \((M-m)\) free cells. For all evaluations there is a critical storage size \( V \), such that if there are less than \( V \) cells of free work space available then the system spends a large part of its time collecting garbage and almost no evaluations are performed. While the difference \((M-m)\) is greater than \( V \), the system with attributes always performs more efficiently than the host system.

If a problem is attempted in both systems, (systems with and without attributes), and the evaluation time used is plotted against the available storage space, then we have curves of the type given in Fig. 3.3.

Two points should be noted from these graphs. First, \((t_H - t_A)\), which is the gap between the steady state of the two curves, is a measure of the saving in central processor time due to the use of attributes. The steady state of the two curves refers to the portions of the curves above the two
Fig. 3.3. Curves show that as the storage space is reduced a point is reached beyond which the host system performs more efficiently than the systems with attributes.
points A and H. If the storage space available is larger than these values no garbage collections are performed, and the time taken is that for the actual evaluation. Second, A and H are two critical points below which the system spends a substantial amount of its time collecting garbage rather than doing useful work. As the graphs show, the presence of attributes shifts the point downwards from K to Kl. The extent of this shift is important because if it is large, problems which proved uneconomic to solve under the system without attributes can be solved satisfactorily using a system with attributes.

An example of this behaviour is provided by the solution of the differentiation problem (described in Section 2.3.2). Using the host system, the solution had to be abandoned as the size of the free word list was reduced to about 4000 free cells. The same problem was successfully solved in both the expression and the function driven systems with permanent attributes, with a free word list of less than 3000 free cells in size. Moreover, the number of garbage collections required to solve the problem in the expression and the function driven systems with permanent attributes, with the free word list consisting of 3000 free cells, proved to be nearly equal to the number of garbage collections required to solve the same problem in the host system with the free word list consisting of 7000 free cells.

3.6.2. Problem 1 (Differentiation 1: Section 2.3.1)

Burden on the user

The user has two tasks to perform. The first is to
recognise the useful attributes, that is, attributes which will help speed up the evaluation, and the second is to use the appropriate commands to establish them. For example, in the evaluation of

\[ \frac{d^p}{dx^p} (a + (x - b)^n)^m \quad 1 \leq p \leq 7 \]

where, \( a \) and \( b \) are constants and \( m, n \) and \( p \) are integer variables; a user who is familiar with the working of the differentiation function, (Section B.2.1), can recognise that the expressions

\[ (a + (x - b)^n); (x - b) \]

are the most significant among the expressions which are repeatedly differentiated. Notice that in the case of the sixth order differentiation, \( (p = 6) \), most of the expressions which are differentiated more than once arise in the intermediate stages, and the user has no direct knowledge of how often each intermediate expression is used. Hence in using the expression and function driven systems, we choose to create the attributes for two expressions whose differentiation occurs in every step:

\[ \frac{d}{dx} (a + (x - b)^n) = n(x - b)^{n-1} \]

\[ \frac{d}{dx} (x - b) = 1 \]

In this particular case the graph shows that the number of occasions on which values were obtained from the attribute-lists increases very rapidly as the order of differentiation increases, Fig. 3.6.

The set of commands needed to create the appropriate attribute-lists for the expression and function driven systems
Fig. 3.6. This curve shows variation in the number of attributes used with respect to the order of differentiation in the differentiation I problem.
are listed in Section (B.6.1).

**Storage Space**

The two attributes created occupy 37 and 35 words in the expression and function driven systems, respectively. This difference in the storage space occupied by the attributes is so small that the curves showing the number of garbage collections performed for the expression and function driven systems, plotted against the order of differentiation, (Fig. 3.4), coincide. In examples where the difference in the storage space occupied by the attribute-lists in the two systems is substantial, the garbage collection curves do not coincide (Section D.1).

We notice that in comparison with the host system, the use of systems with attributes does not change the number of garbage collections for differentiation up to fourth order (see tables of Section B.5.1). As we go on to higher orders of differentiation the ratio:

\[
\frac{\text{number of garbage collections required in a system with attributes}}{\text{number of garbage collections required in the host system}}
\]

decreases. It is the prevention of the repetitive evaluations which saves storage, and this causes a reduction in the number of garbage collections.

**Speed of Evaluation**

The time required to carry out the same set of evaluations in the expression and function driven systems differs from the fourth order of differentiation onward. This difference is clearly shown in the curves of the evaluation time plotted
Fig. 3.4. The curve shows the variation of the number of garbage collections required with respect to the order of differentiation for the differentiation problem in the expression and function driven systems with permanent attributes.
against the order of differentiation, given in Fig. 3.5. Little, if any, of this time difference arises because of the time difference in the garbage collection cycles between the two systems, since exactly the same evaluation is attempted with a free word list consisting of 7000 cells. According to the tables of Section (B.5.1), we see that the number of unsuccessful searches of the attribute-list in the function driven system must be a major contributory factor to this time difference between the two systems. There is also a component of this difference arising from differences in the look-up time, (Section 3.6.3), for searching the attribute-lists.

Since evaluation time cannot be measured very accurately, the difference in the evaluation time between the two systems does not show until fourth order differentiation is reached.

In comparison with the host system, the expression and function driven systems perform 1.1 times faster at the third order of differentiation and 1.3 and 1.2 times faster, respectively, at the sixth order of differentiation. Although, according to the curves of Fig. 3.6, the number of attributes used increases sharply with the order of differentiation, a correspondingly rapid increase does not occur in the speed of evaluation. This is because the two attributes which are used are only encountered at lower levels of recursion, so they only reduce the number of calls to the differentiation function by a relatively small amount. Further attributes could be created and these would give further improvements.
Fig. 3.5. The two curves indicate how the evaluation time varies with the order of differentiation in the expression and function driven systems for the differentiation problem.
Burden on User

Some of the repetitive operations which can be removed from the evaluation by using an attribute system are described in Section (2.3.2). The number of commands needed to create the appropriate attribute-lists are 13 and 2 in the expression and function driven systems, respectively, and they are listed in Section (B.6.2). If, in the function driven system, we choose to name expressions prior to their subsequent use in other commands, then the labour of using the function of expression driven system is equivalent. This is clearly indicated in the set of commands listed in the section just referenced. From this example, it is clear that in evaluations where a large number of attributes is required, the clerical burden that the use of either expression or function driven system imposes upon the user is considerable.

Storage Space

The effect of attributes on the use of storage space is demonstrated by the curves given in Fig. 3.7, where the number of garbage collections is plotted against storage space available. In this particular case the two curves coincide. But this is not normally the case.

As indicated in the diagrams of the internal representation given in Sections (3.3.1) and (3.4.1), the expression \((ax + b)\) which forms part of the argument expressions is preserved only once as a natural-value in the expression driven system, and 4 times as the constituent part of the argument expressions in the function driven system. In
Fig. 3.7. Curve shows the variation of the number of garbage collections invoked with respect to the storage space available to solve the differentiation problem in the expression and function driven systems with permanent attributes.
general, let us assume that $Q$ denotes an expression which occurs as part of $m$ argument expressions, and that storing a copy of $Q$ uses $n$ units of storage space. In the expression driven system, therefore, the expression, $Q$, is the natural-value of an attribute-list and is stored only once. The storage space required, $SE$, for $Q$ itself is given by:

$$SE = n \text{ units}$$

In the function driven system, on the other hand, the expression, $Q$, is preserved $m$ times, (once with every attribute). In this case the storage space, $SF$, required to save copies of $Q$ is given by:

$$SF = mn \text{ units}$$

The difference between the storage space requirements of the two systems is therefore given by the relation:

$$SF - SE = n(m - 1) \text{ units}$$

The difference between the number of garbage collections required to solve the same problem in the expression and function driven systems arises largely as a result of this difference, $(SF - SE)$, in the storage space requirement of the two systems. Other differences in attribute-list structure contribute little.

In this instance, the attribute-lists created occupy 476 and 540 free cells in the expression and function driven systems, respectively. The difference in the storage space requirement of the attribute-lists in the two systems is not due to the explanation given above, since each attribute is preserved only once ($m = 1$). This difference (64 cells) is therefore solely due to the different internal structures used
in the expression and function driven systems.

The two garbage collection curves, given in Fig. 3.7, for the expression and function driven systems coincide because, for the reasons given above, the space occupied by the attribute-lists is approximately equal. An instance where the solution of the same problem in the expression and function driven systems is likely to give rise to different numbers of garbage collections is discussed in Section (D.1).

Time of Evaluation

According to data in the tables in Section (B.5.2), 100 times in the course of this evaluation the value is retrieved from the attribute-list. This method of evaluation reduces the number of entries to the routine apply from 14387 in the host system to 5520 in both the expression and function driven systems.

It was found that, on average, the expression and function driven systems perform 2.5 and 2.4 times faster than the host system respectively.

In spite of the fact that the processes of evaluation performed by the expression and function driven systems with permanent attributes are the same, and their garbage collection curves as we have seen coincide, their central processor time curves as shown in Fig. 3.8 do not. Since the same number of garbage collection cycles was invoked in the expression and function driven systems, this gap between the two curves is largely due to the difference in the look-up time of attributes. When the garbage collection curves for the expression and the function driven systems do not coincide, the gap between the
Fig. 3.8. Curves show the variation of the central processor time with respect to the storage space available to solve the differentiation 2 problem in the expression and function driven systems with permanent attributes.
central processor time curves accounts both for the extra look-up time of attributes and for the time taken by the extra garbage collections.

The following argument shows how this difference arises. An attribute is a pair-list of the form:

(argument; result)

In a successful search, the current argument expression must be identical to the first element of one of the entries in the attribute-list being searched. If the argument expression occupies $t$ cells, on average there are $(t + 1)/2$ comparisons in each unsuccessful match in the function driven system. The internal representation in the expression driven system does not include the natural-value as a constituent of the argument in the attributes. An unsuccessful match in the expression driven system therefore involves $(t - n + 1)/2$ comparisons, where $n$ is the number of cells taken up by the natural-value. So, the difference in the argument part of the attributes in the expression and function driven systems gives rise to a difference in their look-up time. This fact makes the function driven system less suitable for problem areas where the argument expressions are complex. The supporting evidence for this appears in Section (D.1).

3.6.4. Problem 3 (Prime numbers: Section 2.3.3)

Burden on User

$\text{Prs}$, which was described in Section (2.3.3), takes a positive integer as its argument. If the arguments of the function $\text{prs}$ are being generated as part of a larger evaluation,
a user may have some idea of the range of the arguments, but no idea of the frequency of use of each individual integer argument. To avoid defining an evaluation dependent on \texttt{prs}, this function has been evaluated using a set of 500 randomly chosen positive integers which are in the range $0 < n < 100$; these random integers are listed in Section (B.4).

If an expression driven system with permanent attributes is used for an evaluation, then an extra burden is placed on the user. This is because each object, as well as each attribute, must be explicitly created, that is, up to 100 objects and attributes for the example evaluation. In circumstances like this, the use of the function driven system with permanent attributes is to be preferred because it imposes much less burden on the user. Accordingly, the evaluation was performed only with the function driven system. An important fact that arises here is the choice of a suitable function definition. If a recursive definition of \texttt{prs} is selected, then it is possible to avoid a lot of extra work. For example, if we create the attributes of all the integers in the range $0 < n < 100$ which are divisible by 5

\begin{align*}
\texttt{prs}(5); \texttt{prs}(10); \ldots \ldots \texttt{prs}(95); \texttt{prs}(100)
\end{align*}

then with the recursive definition of \texttt{prs} (Section E.8), one of the following situations can arise. In the evaluation of \texttt{prs}(n), if the corresponding value exists on the attribute-list then the value of the function is retrieved directly from the attribute-list. If the value of \texttt{prs}(n) does not exist on the attribute-list then a combination of recursive evaluation plus attribute look-up is used to obtain the required value. For example, in the evaluation of \texttt{prs}(47); \texttt{prs}(47) requires the
value of \( \text{prs}(46) \) and \( \text{prs}(46) \) requires the value of \( \text{prs}(45) \). The value of \( \text{prs}(45) \) can now be obtained from the attribute-list and, as a result, a considerable amount of evaluation is prevented.

With the iterative definition of \( \text{prs} \), (Section E.8.), unless the value is found on the attribute-list it must be calculated directly. That is, the presence of the attribute for \( \text{prs}(45) \) does not facilitate the evaluation of \( \text{prs}(47) \).

The complete set of commands needed to create the attribute-list is given in Section (B.6.3).

Storage Space

Fig. 3.9 shows the garbage collection curves for \( \text{prs} \) in the function driven system. In comparison with the host system, (Section 2.3.3), the use of the function driven system in conjunction with the recursive definition of \( \text{prs} \) gives rise to an impressive reduction in the number of garbage collection cycles used. The considerable difference between the number of garbage collections under the recursive and iterative definitions is due to the manner in which attributes assist the evaluation.

Time of Evaluation

From the information in the tables given in Section (B.5.3), it can be seen that for the recursive definition of \( \text{prs} \) the function driven system performs approximately 20 times faster than the host system. Such a gain in the speed of evaluation arises as the result of the 488 values obtained from the attribute-list, which in turn reduces the number of
Fig. 3.9. Curves show the variation of the number of garbage collections required with respect to the storage space available for the recursive and iterative type definitions of pro in the function driven system with permanent attributes.
entries to the routine apply from 384,281 to 15,440. In addition, the reduction in the number of garbage collections gives a further saving in the time of evaluation.

With the iterative definition of prs, we have a somewhat different outcome. By making use of the data provided in the tables of Section (B.5.3), it can be shown that the function driven system performs only 1.2 times faster than the host system. This arises from the fact that only 99 values are obtained from the attribute-list, which reduces the number of entries to the routine apply from 384,281 to 301,083. The reduction in the number of garbage collection cycles also contributes to the saving in the evaluation time.

The variation of the evaluation time with respect to the storage space available for the recursive and iterative definitions is shown by the curves of Fig. 3.10.

3.6.5. Summary

As pointed out in Section (3.6.3), due to the internal representations adopted for attribute-lists in the expression and function driven systems, the storage space requirement of the function driven system is usually more than that of the expression driven system. This difference in the storage space requirements makes the expression driven system more suitable for problems with large common expressions (Section D.1). With manipulations where there are no common expressions, or where the argument expressions consist of a few atomic symbols, the function driven system is to be preferred. This is because the storage space requirements of attribute-lists are only slightly larger and the burden of creating the objects
Fig. 3.10. Curves show the variation of the central processor time with respect to the storage space available for the recursive and iterative type definitions of prs in the function driven system with permanent attributes.
before creating the necessary attributes is removed.

In the case of the differentiation problems attempted, the expression driven system performs slightly faster than the function driven system. Some of the reasons for this difference in the speed of evaluation are as follows. In the expression driven system there are no unsuccessful searches (see tables of Sections B.5.1 and B.5.2). This is because, if an argument-list does not contain a reference to an object there is no attribute-list to be searched, and if an argument-list contains a reference to an object, then its attribute-list is searched and the attribute required is obtained. With the function driven system, the attribute-list associated with the function is searched every time the function is used in an evaluation. This gives rise to a number of unsuccessful searches as shown in the tables referenced above. The other contributory factors are the slight difference in the look-up time between the expression and function driven systems, and a slight difference in the time spent on the garbage collection cycles in the two systems.

Normally, recursive definitions are regarded as being less efficient than iterative ones due to the house-keeping operations involved. In the case of the function this point of view is confirmed by using both types of definition on the same set of data in the host system, (see tables of Section B.5.3). Using the function driven system with attributes, the recursive and iterative definitions perform 20 and 1.2 times faster, respectively, than the host system. The recursive definition of a function is therefore preferable in this case since the attribute-list is being used to assist
the evaluation at all levels of recursion. With the iterative
definition, the attribute-list is only searched once when the
function is called, and the attribute-list is not used to
assist the evaluations within the function. In general, the
recursive definition of a function is likely to perform more
efficiently than its corresponding iterative definition in
systems with attributes, (Section 3.6.4).

3.7. Conclusion

The expression and function driven systems which have been
implemented are useful in problem areas where the number of
attributes required is small, and the user wishes to maintain
total control over the process of evaluation. The burden
that the use of these two systems imposes on the user comes
from two different sources. First, the determination of those
attributes which are likely to be useful during the evaluation.
This demands a detailed knowledge of the problem being solved,
and as a result imposes a significant burden of awareness on
the user. Second, the need to learn about the system in order
to create the appropriate commands necessary to use the attribute
systems.

The extent of physical labour that the use of these two
systems imposes on the user depends on the number and complexity
of attributes required (complexity is used to describe the
length of expressions which are submitted to the system). In
an on-line environment extensive input causes fatigue and is
therefore error prone.

It is dangerous to draw general conclusions from a small
number of trial examples. However, there is a reduction in
the number of garbage collections in each of the three problems attempted. For example, in case of the differentiation 2 problem, (Section 3.6.3), the total number of garbage collection cycles invoked when a free word list consisting of 3000 free cells is used with a system with attributes, is comparable to the number invoked for the same evaluation using the host system and a free word list of 7000 cells. Further, this problem could not be solved in the host system with a free word list of 4000 free cells, (because of the lack of work space), but can be solved with a free word list of 3000 free cells in both expression and function driven systems with attributes.

The use of the systems with permanent attributes described in this chapter enables the user to prevent only those repetitive operations in an evaluation that he is aware of. Consequently, many of the repetitive operations which arise in the intermediate stages of evaluation are not prevented, either because the user is not aware of them, or because the extent of the labour involved in setting up the attributes required is too great. Our aim therefore, is to investigate the possibility of designing a system to meet the following two objectives. First, a reduction in the burden on the user by automating some of the routine tasks he has to carry out. Second, the provision of a mechanism which can prevent repetitive evaluations at all stages during a computation, by the automatic generation of appropriate attributes.
4.1. Introduction

When using the simple symbol manipulation systems described in the previous chapter, the user has to create the attribute-lists and look after their house-keeping. In this chapter we describe a function driven system which is capable both of creating and using attributes, and of performing all the house-keeping operations on the attributes which it creates. This new system also includes the facilities for creating and using permanent attributes described in Chapter 3.

We begin by describing a set of criteria which are used to determine which system generated attributes are preserved and which are deleted. The system described is based upon the use of a "fixed-size attribute-list"; that is, the maximum number of system generated attributes that a memoized element can have is explicitly specified by the user. The criteria used for deciding which attributes to save, and the fixed-size of the attribute-lists used, control the number of attributes saved during an evaluation.

As before we provide a brief description of the most important features of the system which has been implemented. This is followed by a description of the user commands which are particular to this system.

The performance of the system implemented was evaluated by using it to solve a number of specific problems, among them those described in Chapter 2. The experimental results
show that the new system normally performs more efficiently than both the host system and the systems with permanent attributes, provided that a correct choice has been made for the maximum number of system generated attributes that an element can have. Further, the burden on the user is substantially reduced. This is because the user only has to intervene in situations where he finds it advantageous, or the system is not capable of handling the problem on its own.

4.2. Basic Concepts

In writing a program to solve a problem, a table is often allocated to store values which are frequently required. When these values are subsequently required during the evaluation, they are retrieved from the table. This has the following consequences: first, the user must analyse a problem in detail and identify those operations which are frequently performed and consequently lead to values which are used many times. Second, he must use his experience and skill of programming to construct a program which will make entries in a table and ensure that repetitive evaluations take advantage of the values stored in the table. These tasks place a substantial burden on the user.

But there is an additional consideration. While for many numerical problems the analysis to identify repetitive operations is straightforward for many symbol manipulation problems it is not. This is because in symbol manipulation it is the data which determine what operations are repetitive and what are not. For example, it is not easy to write a differentiation function in such a way that temporary locations
are used to prevent all the repetitive operations, unless the data which are to be processed by the function have been completely specified in advance.

In systems where attributes can be created and used automatically, each memo function or memo expression is associated with two attribute-lists, one holding attributes created by the user, the other handling attributes created by the system. No constraint is placed upon the size of the user defined attribute-lists, while the size of the system defined attribute-lists is controlled explicitly by the user. In addition, in the system implemented, a simple scheme described below operates so that during the evaluation, the unused system generated attributes are forgotten while the used ones are remembered.

Without some means of control, all the entries in the fixed-size system attribute-lists are filled by the first few attributes as they are generated, and that state of these attribute-lists will remain unchanged throughout the remainder of the evaluation. To prevent this, a number of rules are used to govern the entries made in the system attribute-lists and they are listed below.

The user identifies those functions and expressions which are to be associated with attribute-lists and also specifies the number of entries in each of the system attribute-lists.

(a) When an attribute is used, it is promoted towards the top of the attribute-list.

(b) If the attribute required does not exist, the value of
the function is calculated and its corresponding attribute is formed and added to the attribute-list of the appropriate element.

(c) If all the entries in a fixed-size system attribute-list are occupied and a new attribute is generated, then the last entry is deleted from the bottom of the attribute-list in order to make room for the insertion of the new attribute.

Consequences of the use of the above rules are:

(i) for this set of rules to operate successfully, the system attribute-list must have a linear structure,

(ii) frequently used attributes are close to the top of the attribute-list and this reduces the look-up time,

(iii) the attributes which are created and become redundant are eventually deleted from the bottom of the system attribute-list so that the corresponding storage space can be used again and the look-up time of useful attributes is not increased,

(iv) as long as the attributes maintain some specific frequency of use, they will remain within the system attribute-list,

(v) the insertion of a trivial but commonly used attribute can cause the deletion of a complex and useful attribute.

4.3. Expression driven system

If an expression is associated with a fixed-size attribute-list, each time the value of a function containing this memo expression as its argument, (or constituent of its argument), is evaluated the corresponding attribute is
constructed and added to the attribute-list. Hence in the course of evaluation the number of attributes in the attribute-list of a memo expression will depend on the number of functions which take the expression as their argument or part of their argument. Although the implementation of the expression driven system with a fixed-size attribute-list is straightforward and does not pose any practical difficulty, we decided against its implementation for two reasons. First, for the set of attribute housekeeping rules to work, the attribute-list must have a linear structure. As a result, the structure we adopted for attribute-lists in the expression driven system (Section 3.3.1) is not workable in this case. Of course attribute-lists with a linear structure could be created. Second, the size and complexity of problems in which many functions are repeatedly applied to a number of common expressions means that testing such a system is difficult.

4.4. Function driven system

4.4.1. Basic features

In the function driven system described below, with fixed-size attribute-lists, the task of selecting the functions used as carriers of attribute-lists is left to the user, but the subsequent creation of attributes and their house-keeping is under the control of the system. With the automation of these two tasks, a user can spend most of his time on the problem in hand rather than having to cope with the requirements of the system.

In order to keep permanent and temporary attributes
separate, the attribute-list structure had to be modified. As shown in the diagram below, the value of the property indicator \( \text{ROTE} \) is now a pointer to a cell whose \text{car} holds the start address of the list containing the temporary attributes, and whose \text{cdr} holds the start address of a list containing the permanent attributes.

To see how this structure is used consider:

\[ Q = x^2 + y^2 + z^2 + t^2 \]

with permanent attributes:

\[ \frac{\partial Q}{\partial x} = 2x; \quad \frac{\partial Q}{\partial y} = 2y \]

and temporary attributes:

\[ \frac{\partial Q}{\partial z} = 2z; \quad \frac{\partial Q}{\partial t} = 2t \]

The attribute-list which is attached to the property list of the \text{memo} function has the form
where the meaning of the symbols used in this diagram can be found in Section (B.1).

There are two advantages in keeping the permanent and temporary attributes of a memo function separate. First, the system is relieved of the task of permanently ensuring that all the user specified attributes are not interfered with. Second, the user is more likely to be aware of the positions of permanent attributes within the user attribute-lists, knowledge of which is essential to the attribute house-keeping of the permanent attributes which he must perform.

4.4.2. **Attribute house-keeping operations**

As in the function driven system with permanent attributes, the house-keeping for permanent attributes is the responsibility of the user. To minimize this burden, therefore, only those values of a function which the system cannot evaluate, or
which have a high frequency of use in an evaluation, are established as permanent attributes. The unnecessary creation of permanent attributes that the system can handle as temporary attributes will result in:

(a) imposition of the extra burden of the attribute housekeeping upon the user,

(b) an increase in the look-up time (Section 4.4.3), since the search of the user attribute-list takes priority over that of the system attribute-list,

(c) an insufficient use of the storage space if permanent attributes are preserved without considering their frequency of use.

As functions which are to be associated with attribute-lists are identified, the maximum number of temporary attributes that each function can possess is specified. The system carries out the attribute house-keeping operations according to the following rules:

(i) Every time an attribute is generated, it is placed at the top of the system attribute-list of the appropriate function.

(ii) Every time an attribute is used, it is promoted by one position towards the top of the system attribute-list.

(iii) If the space allocated in the system attribute-list is fully occupied, when a new attribute is added to the list, an attribute is deleted from the bottom of the list.

Clearly promotion not only reduces the look-up time but it also ensures preservation of attributes that are used, and
destruction of attributes that are not used.

4.4.3. Operation of the interpreter

With a memo function the sequence of operations which is carried out is presented in the flow-chart given in Fig. 4.1. Essentially, this flow-chart fragment replaces the dashed rectangular box in the flow-chart of apply provided in Section (A.1). The modifications made increased the size of the interpreter of the host system by 1525 words.

Notice that the search of the user attribute-list takes precedence over that of the system attribute-list. For this reason a careless use of permanent attributes creates an overhead on the look-up time of all temporary attributes. If, on the other hand, the most frequently used temporary attributes are placed in the class of permanent attributes, this reduces their look-up time and eliminates their re-construction time. Re-construction time is the time taken to recreate temporary attributes which have already been deleted from the attribute-list. For complex attributes, the total re-construction time may be considerable, and this is a factor which must be taken into account when considering the choice of permanent attributes.

Example

To illustrate the process of evaluation we use the first order differentiation:

\[ \frac{d}{dx} (A + (x - B)^n)^m \]

where A and B are constants and m and n are integers.
Fig. 4.1. Flow-chart shows the modifications which had to be made to the interpreter of the host system in order to make it workable for the function driven system with fixed-size attribute-lists.
Suppose that a fixed-size attribute-list associated with the
differentiation function can hold a maximum of five attributes.
No permanent attributes are used. The process of
differentiation is carried out according to the set of rules
listed in Section (B.2) and the steps of evaluation are
annotated using the notation described in the same section.
The steps in this evaluation are as follows:

1. \( \frac{d}{dx} (A + (X - B)^n)^m = m(A + (X - B)^n)^{m-1} \) (D6)

\[ \frac{d}{dx} (A + (X - B)^n)^m = m(A + (X - B)^n)^{m-1} \]

\[ E1 \]

\( E1 \quad \frac{d}{dx} (A + (X - B)^n) = \frac{d}{dx} (A) + \frac{d}{dx} (X - B)^n \) (D2)

\[ E2 \quad \frac{d}{dx} (A) = 0 \]

\[ E3 \quad \frac{d}{dx} (X - B)^n = n(X - B)^{n-1} \frac{d}{dx} (X - B) \] (D6)

\[ E4 \quad \frac{d}{dx} (X - B) = \frac{d}{dx} (X) - \frac{d}{dx} (B) \] (D3)

\[ E5 \quad \frac{d}{dx} (X) = 1 \] (D1)

\[ E6 \quad \frac{d}{dx} (B) = 0 \] (D0)
E4 \[ \frac{d}{dx} (X - B) = 1 \]  

system attribute-list:
\[ \{ \frac{d}{dx} (X - B) = 1 \ ; \ \frac{d}{dx} (B) = 0 \ ; \ \frac{d}{dx} (X) = 1 \ ; \ \frac{d}{dx} (A) = 0 \} \]

E3 \[ \frac{d}{dx} (X - B)^n = n(X - B)^{n-1} \]  

system attribute-list:
\[ \{ \frac{d}{dx} (X - B)^n = n(X - B)^{n-1} \ ; \ \frac{d}{dx} (X - B) = 1 \ ; \ \frac{d}{dx} (B) = 0 \ ; \ \frac{d}{dx} (X) = 1 \ ; \ \frac{d}{dx} (A) = 0 \} \]

We notice that at this point all the positions in the attribute-list of the differentiation functions are occupied.

E1 \[ \frac{d}{dx} (A + (X - B)^n) = n(X - B)^{n-1} \]  

system attribute-list:
\[ \{ \frac{d}{dx} (A + (X - B)^n) = n(X - B)^{n-1} \ ; \ \frac{d}{dx} (X - B)^n = n(X - B)^{n-1} \ ; \ \frac{d}{dx} (X - B) = 1 \ ; \ \frac{d}{dx} (B) = 0 \ ; \ \frac{d}{dx} (X) = 1 \} \]

According to the house-keeping rules given in Section (4.4.2), as the attribute \( \frac{d}{dx} (A + (X - B)^n) = n(X - B)^{n-1} \) is inserted at the top of the attribute-list, the attribute \( \frac{d}{dx} (A) = 0 \) is deleted from the bottom of the attribute-list.

1. \[ \frac{d}{dx} (A + (X - B)^n)^m = mn(A + (X - B)^n)^{m-1} (X - B)^{n-1} \]  

system attribute list:
\[ \{ \frac{d}{dx} (A + (X - B)^n)^m = mn(A + (X - B)^n)^{m-1} (X - B)^{n-1} \ ; \ \frac{d}{dx} (A + (X - B)^n) = n(X - B)^{n-1} \ ; \ \frac{d}{dx} (X - B)^n = n(X - B)^{n-1} \ ; \ \frac{d}{dx} (X - B) = 1 \ ; \ \frac{d}{dx} (B) = 0 \} \]
Notice that due to inadequate space for entries in the system attribute-list, the attribute $\frac{d}{dx}(A)$ and $\frac{d}{dx}(X)$ have been deleted from the end of the system attribute-list by the time the evaluation of the first order differentiation is completed. A trace of $\frac{d^2}{dx^2} (A + (X-B)^n)^m$ as evaluated by the system appears in Section (B.3).

4.5. Repertoire of commands

This section describes the three additional commands that are exclusive to the function driven system with fixed-size attribute-lists. All the commands described below are hand coded into the interpreter of the host system. Where indicated below, an M-expression definition of the command together with an example of its use appears in Appendices B and C.

memo \{x\} :SUBR pseudo function

memo takes one argument which is a list. Each element of this list is a pair-list of the form:

(function name; positive integer)

The effect of evaluating each pair-list is to append the structure:
to the property list of the specified function. The meanings of the symbols used in the above diagram appear in Section (B.1). The positive integer, n, is the upper bound limit imposed on the number of temporary attributes that the given function can have, and is associated with the node holding the property indicator ROTE. The command memo can be used subsequently to alter the value of this upper bound.

An increase in the bound leaves the state of the entries in the attribute-list unchanged. On the other hand, a reduction in the bound causes the number of entries in the attribute-list to be reduced, if necessary, to the newly specified limit by deleting some of the attributes from the bottom of the attribute-list. No error messages are generated and no value is returned. Examples of its use and its M-expression definition appear in Sections (B.6) and (C.2) respectively.

rotenos \{x\} :SUBR

This function takes as its argument a list of names of functions which are associated with fixed-size attribute-lists. For each memo function, it prints out the upper bound that has been specified by the user, and the number of attributes currently in the attribute-list. An error message is generated if the function used is not a memo function.

promote \{n\} :SUBR  pseudo function

In Section (4.4.2) it was pointed out that every time an attribute is used, it is promoted by one position towards the top of the attribute-list. This command enables the user
to alter this rate of promotion. An error message is generated if the rate of promotion is greater than the upper bound on the number of entries. No value is returned. Examples of use of this command appear in Section (B.6).

4.6. Analysis of practical results

We use the solution of the same problems as in Chapter 3 to identify some of the effects that the use of the function driven system with fixed-size attribute-list has on the various aspects of efficiency, that is, the burden on the user, the use of storage space and the time of evaluation.

As the description of these examples shows, the use of this system demands little extra work from the user and imposes only a small burden of awareness on him. In addition, its use produces a substantial reduction in the time of evaluation and the use of storage space.

The data used in the comparative study of the various systems, as well as those used in plotting the curves and constructing other tables, are taken from the appropriate tables provided in Section (B.5). Apart from the problems used in the discussion in this section, the experimental results of the two additional problems provided in Section (D.3) and (D.4) show how the use of attributes improves the performance of the host system. The choice of the additional problems just referenced is arbitrary and has no special significance in relation to the system described.
Burden on the user

So that the system can create and use attributes to aid evaluation, the differentiation function is identified as a memo function with a fixed-size attribute-list of 100 entries. The choice of the size of the attribute-list is an arbitrary one. It turns out to be large enough to hold all the attributes created up to the sixth order differentiation of the expression chosen. Once the differentiation function is declared as a memo function and the size of its attribute-list is fixed, the user has no further responsibility for the fact that attributes are being created, used and destroyed during the evaluation. The set of commands needed to carry out this evaluation appears in Section (B.6.1).

Storage space

During the evaluation, the storage space required by the attributes fluctuates, depending on their complexity. Fig.4.2 shows how the storage space required by the attribute-list varies with respect to the order of differentiation.

The table given below shows the number of garbage collections required to evaluate the differentiation 1, problem (Section 2.3.3), under the various systems. As indicated in the table, seventh order differentiation could not be carried out in the host system and systems with permanent attributes.
Fig. 4.2. This curve shows the variation in the number of cells required to house the attribute-list with respect to the order of differentiation.
In comparison with the host system, the function driven system with fixed-size attribute-lists does not affect the number of garbage collections required up to the second order of differentiation, but reduces it by a factor of 10 for the sixth order of differentiation. In comparison with the expression and function driven systems with the two permanent attributes, (Section 3.6.2), the use of this system reduces the number of garbage collections required by factors of 9 and 13 respectively for the sixth order differentiation. The main reason for these large reduction factors is the creation and use by the system of many attributes in intermediate stages. As expected, the magnitude of these reduction factors increases with the order of differentiation.

Time of evaluation

A major contributory factor in the saving of time of evaluation is the number of values which are obtained from the attribute-list. To give some idea of this effect we plot the curves of the number of times that the attribute-list is searched, and the number of times that the value is
obtained from the attribute-lists, against the order of differentiation, Fig. 4.3. The curves show how the number of repetitive operations increases rapidly with the complexity of the problem, (higher order of differentiation).

From the third order differentiation onward the total number of times that attribute-lists are searched is larger than in the expression driven system with permanent attributes, and less than in the function driven system with permanent attributes. This is because of the complex attributes that the system creates and uses during the evaluation. Note that the difference in the search time between the expression and function driven systems with permanent attributes is comparatively small, since the attribute-lists contain only one and two attributes respectively.

In comparison with the host system and systems with permanent attributes, a reduction in the evaluation time is apparent from the second order differentiation onward. This is indicated in the following table containing the evaluation time for the differentiation 1 problem.

<table>
<thead>
<tr>
<th>order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>47</td>
<td>139</td>
<td>528</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EDS: permanent attributes</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>34</td>
<td>109</td>
<td>408</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FDS: permanent attributes</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>38</td>
<td>121</td>
<td>455</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FDS: fixed-size attribute-list 100</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>23</td>
<td>40</td>
<td>77</td>
<td>343</td>
<td>X</td>
</tr>
</tbody>
</table>
Fig. 4.3. The curves show the variation of the "total number of searches" and the number of attributes available with respect to the order of differentiation.
Notice that this system performs third and sixth order differentiation 1.7 and 6.8 times faster, respectively, than the host system.

From the third order of differentiation onward, this system begins to perform faster than the expression and function driven system with permanent attributes. For example, with the sixth order differentiation this system performs 5.3 and 5.9 times faster than the expression and function driven systems with permanent attributes, respectively. Obviously, these factors will increase with the order of differentiation.

4.6.2. Problem 2 ( Differentiation 2 : Section 2.3.2)

Burden on the user

In this instance no permanent attributes are created. The only task for the user is to identify the differentiation function as a memo function, and to set the upper bound (100) on the number of temporary attributes that it can hold. This is simply achieved by one command as indicated in the Section (B.6.2).

The choice of size for the attribute-list is arbitrary. In fact in this case it turns out that the attribute-list is not large enough to hold all the attributes generated in the evaluation.

Storage space

Using the trace mechanism it was established that there are more than one hundred calls with different arguments to
the differentiation function and, as a result, there are more than 100 temporary attributes to be handled. The average storage space required to accommodate these attributes is about 1530 cells. With a free word list of 5000 cells, the rate of creation and destruction of attributes with respect to the garbage collection is diagrammatically represented below:

Two points to note:

(a) the two curves coincide once the upper bound on the number of attributes is reached. This is because as one attribute is added to the top of the attribute-list another attribute is deleted from the bottom.

(b) the storage space required by the attribute-list fluctuates depending on the complexity of the attributes on the list.
Fig. 4.4 shows the variation of the number of garbage collections required with respect to storage space available. Compared with the function driven system with permanent attributes, (see tables of Section B.5.2), the number of garbage collections involved is reduced by a factor of 2, and compared to the host system the number of garbage collections involved is reduced by a factor of 6 when using a free word list of 7000 cells. The tables just referenced also show that the problem, which could not be solved using a free word list consisting of 4000 cells in the host system, can now be solved with a free word list of 2000 cells. With a free word list of 2000 cells, however, the number of garbage collections involved is about 45 which is still less than the 46 garbage collection cycles required to solve the same problem using 7000 free cells in the host system.

**Time of evaluation**

According to tables in Section (B.5.2), the attribute-list is searched 298 times in the course of evaluation and on 102 occasions the attribute required was available. In comparison with the host system and using a free word list consisting of 7000 cells, evaluation is carried out 4.3 times faster. Fig. 4.5 shows the variation of the central processor time with respect to storage space. Since the amount of evaluation is constant irrespective of store space available, the variation arises from the extra time used by the additional garbage collection cycles involved.

This system performs approximately 1.7 and 1.8 times faster than the expression and function driven system with
Fig. 4.4. This curve shows the variation of the number of garbage collections required with respect to the storage space available for the differentiation 2 problem in the function driven system with a fixed-size attribute-list of 100 entries.
Fig. 4.5. This curve shows the variation of the central processor time with respect to the storage space available for the differentiation 2 problem in the function driven system with a fixed-size attribute-list of 100 entries.
permanent attributes, respectively, (using a free word list of 7000 cells). The reason for this improvement is the additional temporary attributes which are generated and used in the intermediate stages.

4.6.3. Problem 3 (Prime numbers : Section 2.3.3)

So that the system can create and use the attributes to advantage, we declare prs and prime as memo functions and limit the size of their attribute-lists to 100 attributes, (Section B.6.3). Due to the fact that the range of arguments is $1 < n < 100$, these attribute-lists are large enough to hold all the attributes which are generated in the course of evaluation. Consequently, when all the entries in the attribute-lists are established, the remainder of the evaluation is carried out by using the attributes and without reference to the definition of the function prs.

The table of results given in Section (B.5.3) shows that the use of the recursive definition of prs in this function driven system has the following consequences. First, the garbage collection cycle is invoked only once during the evaluation. Second, the same result was obtained 43 and 2 times faster, respectively, than in the host system and the function driven system with permanent attributes. This improvement is impressive in view of the fact that almost no extra burden is imposed on the user in declaring prs and prime as memo functions, and determining the maximum size of their attribute-lists.

The same evaluation was performed once only using an
iterative definition of prs and a free word list of 7000 cells. In comparison with the function driven system with 20 permanent attributes, (section 3.6.4.), the use of the present system reduced the number of garbage collections required from 104 to 12 and the time of evaluation from 2896 seconds to 244 seconds.

4.7. Critical Factors

In this section we examine how the number of entries in a system attribute-list and a choice of memo functions in a problem affect the evaluation time and the look-up time. We further investigate the variation of the evaluation time with respect to the rate of promotion of attributes within the attribute-lists. All the evaluations are carried out using a free word list of 7000 cells. With the exception of Section 4.7.1, all system attribute-lists contain 100 entries.

4.7.1. Size of system attribute-list

The determination of the upper bound on the number of temporary attributes that a memo function should have is not always an easy task. An over-estimation of this limit can result in all storage space available being used up, and bring the solution of the problem to a halt. An under-estimation of this limit, on the other hand, will dramatically increase the evaluation time. Consider the table in Fig. 4.6, where the data are obtained from the solution of the differentiation 2 problem, (see tables of Section B.5.2). With a system attribute-list consisting of 20 entries, however, the system with fixed-size attribute-list performs less efficiently than the expression and function driven system.
<table>
<thead>
<tr>
<th>Systems used</th>
<th>time in seconds</th>
<th>garbage collections</th>
<th>entries in the system attribute-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>417</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>EDS: 12 permanent attributes</td>
<td>167</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>FDS: 12 permanent attributes</td>
<td>178</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list</td>
<td>338</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>145</td>
<td>11</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>121</td>
<td>9</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>

Fig. 4.6. Table contains the evaluation time and the number of garbage collections required for the differentiation 2 problem in the indicated systems and under the conditions shown.

<table>
<thead>
<tr>
<th>Systems used</th>
<th>time in seconds</th>
<th>garbage collections</th>
<th>entries in the system attribute-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system: recursive definition</td>
<td>3620</td>
<td>130</td>
<td>0</td>
</tr>
<tr>
<td>host system: iterative definition</td>
<td>3429</td>
<td>204</td>
<td>0</td>
</tr>
<tr>
<td>FDS: 20 permanent attributes: recursive definition</td>
<td>178</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>FDS: 20 permanent attributes: iterative definition</td>
<td>2896</td>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list recursive definition</td>
<td>84</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>978</td>
<td>30</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>4350</td>
<td>150</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>4852</td>
<td>190</td>
<td>40</td>
</tr>
</tbody>
</table>

Fig. 4.7. Table contains the evaluation time and the number of garbage collections required to evaluate prs in the indicated systems and under the conditions shown.
with 12 permanent attributes. This is partly because, due to the inadequate number of entries in the system attribute-list, attributes are deleted before they are required again. Such deletions increase the amount of direct evaluation, which in turn results in an increase in the evaluation time and the number of times that attribute-lists are searched.

The table of Fig. 4.7 compares the evaluation time and the number of garbage collections required to evaluate \texttt{prs}, (using its recursive definition), in the specified systems. With the \texttt{prs} function, as the number of entries in the system attribute-list is reduced to the region of 40, the system performs less efficiently than the host system. This is again because the attributes are deleted from the system attribute-list before they are required again. Notice that the user, by making an intelligent choice of 20 permanent attributes, can make a function driven system perform far more efficiently than the function driven system with the number of entries in the system attribute-list as large as 80 - because of random arguments.

4.7.2. Time of direct evaluation versus look-up time

Occasions can arise where the time of direct evaluation of an attribute is less than its look-up time. The preservation of such attributes within the system attribute-list wastes the storage space available, and increases both the look-up time and the evaluation time. Below we give two examples to clarify this point.

**Greater common divisor (GCD: Section E.5)**

If a system attribute-list has a large number of entries,
it is reasonable to assume that the look-up time depends on the position of the entry relative to the top of the system attribute-list, and the complexity of the argument component of the entry being accessed. Since the complexity of all the argument components of all the entries in the attribute-list associated with \( \text{gcd} \) is the same, the look-up time depends only on the position of the entries within the system attribute-list.

The \( \text{gcd} \) function, on the other hand, is a simple function, and the time of its direct evaluation is small and does not vary substantially between different arguments of the function. Hence, as the number of entries in the system attribute-list is increased, a stage is reached where for entries beyond a certain point in the system attribute-list, the look-up time is always greater than the time of their direct evaluation. This is clearly demonstrated in the curves of Fig. 4.8, where the time of direct evaluation of the entries using the recursive definition of \( \text{gcd} \) and their look-up times are plotted against the position of the entries relative to the top of the system attribute-list. As the curve shows this is an instance where too large an attribute-list can prove harmful.

**Differentiation** (Problem 2: Section 2.3.2)

In contrast to the \( \text{gcd} \) example above, for the differentiation function the degree of complexity of the argument components of entries varies among the entries of the system attribute-list. Also, the time of the direct evaluation of most entries is usually much greater than their
**Fig. 4.8.** These curves are the result of plotting the direct evaluation time of the entries and their corresponding look-up time against the position of the entries relative to the top of the system attribute-list for GCD function.

An arbitrary evaluation is arranged so that the system attribute-list of the memo function \texttt{gcd} contains over 500 entries. The above two curves compare the time of direct evaluation and the look-up time of those function evaluations occupying the plotted entries.
look-up time. This is clearly demonstrated in the graph of Fig. 4.9 where the time of the direct evaluation and the look-up time of the same entries are plotted against the position of the entries relative to the top of the system attribute-list. With any reasonable size of system attribute-list, therefore, we are unlikely to encounter the situation where the look-up time is always greater than the time of direct evaluation.

4.7.3. A choice of memo function

If a large number of user-defined functions are involved in the solution of a problem, then the choice of appropriate memo functions is difficult. With a large number of memo functions, a larger portion of the free word list is taken up by the attribute-lists and the number of unsuccessful searches of the attribute-lists increases. It is hard to make any prediction about how these factors affect the evaluation time as measured relative to the host system.

From the result of the differentiation 2 problem:

<table>
<thead>
<tr>
<th>Memo functions</th>
<th>garbage collections</th>
<th>total searches</th>
<th>attributes found</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRV</td>
<td>8</td>
<td>298</td>
<td>102</td>
<td>95</td>
</tr>
<tr>
<td>DRV; +</td>
<td>8</td>
<td>364</td>
<td>109</td>
<td>99</td>
</tr>
<tr>
<td>DRV; +; -</td>
<td>8</td>
<td>408</td>
<td>124</td>
<td>103</td>
</tr>
<tr>
<td>DRV; +; -; *</td>
<td>8</td>
<td>626</td>
<td>132</td>
<td>113</td>
</tr>
<tr>
<td>DRV; +; -; *; +</td>
<td>8</td>
<td>657</td>
<td>139</td>
<td>113</td>
</tr>
</tbody>
</table>

We notice that the addition of the simplification functions +, -, * and +, (Section E.4) to the list of memo functions
Fig. 4.9. The above curves are the result of plotting the
look-up time of entries of the system attribute-list,
and the time of the direct evaluation of the
corresponding entries against the position of the
entries relative to the top of the system attribute-
list.

Once the differentiation 2 problem is complete
at least 100 entries of the system attribute-list
are occupied by the temporary attributes. The
above curves compare the look-up time and the time
of direct evaluation of those function evaluations
occupying the plotted entries.
one at a time brings about the following changes:

(a) The evaluation time is increased as the number of memo functions is increased.

(b) The number of successful searches as a percentage of all searches of the attribute-lists decreases as the number of memo functions is increased.

(c) The number of garbage collections required remains constant. Two of the reasons for this are as follows:
first, attributes may be pointers to constituent expressions and the expressions involved need not be copied, thus saving storage space. Second, the extra attributes are likely to save direct evaluations and so use less storage space.

Moreover, since the number of garbage collections required has remained almost constant, most of the increase in the evaluation time is due to the extra look-up time. As the above table shows, in this particular problem the best result is obtained if only the differentiation function is declared as a memo function.

Another problem which behaves somewhat differently is the function which, given an integer, returns a list of all its divisors as the value, (Section E.9). The first 250 random numbers listed in Section (B.4) form the argument set for this evaluation. As the table:
shows the evaluation time decreases as the number of memo functions in the problem increases. Further, the number of garbage collections required is reduced to a minimum when all the functions are memoized. This is an instance where a better result is obtained when all the user-defined functions are declared as memo functions. Notice that although the conversion of each function to a memo function does not bring about a striking improvement, a combination of them has lead to a significant improvement.

4.7.4. Rate of promotion

As stated in Section (4.4.2), the number of positions through which an attribute is promoted towards the top of the system attribute-list as it is used is significant, because it helps to reduce the look-up time and to prevent frequently used attributes from being deleted from the bottom of the system attribute-list.

The table:

<table>
<thead>
<tr>
<th>rate of promotion</th>
<th>0</th>
<th>1</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>time in seconds</td>
<td>99</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>garbage Collections</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
shows the variation of the evaluation time and the number of garbage collections with respect to the rate of promotion of attributes for the differentiation 2 problem. The above table indicates that with zero rate of promotion, some of the attributes required in the subsequent evaluation may have been deleted from the system attribute-list and the look-up time is also increased.

With the evaluation of:
\[ \frac{d^7}{dx^7} (a + (x - b)^n)^m \]
(Section 2.3.1), the rate of promotion of attributes produces the following results:

<table>
<thead>
<tr>
<th>order of promotion</th>
<th>0</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>time in seconds</td>
<td>527</td>
<td>343</td>
<td>256</td>
<td>255</td>
<td>253</td>
<td>252</td>
<td>251</td>
<td>250</td>
</tr>
<tr>
<td>garbage collections</td>
<td>139</td>
<td>88</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

The substantial increase in the evaluation time and the number of garbage collections at the zero rate of promotion is because many of the attributes are being deleted from the system attribute-list. The above table further shows that the evaluation time and the number of garbage collections decreases as the rate of promotion increases. For this specific problem and under the stated conditions, therefore, it may be preferable to promote an attribute which is used to the top of the system attribute-list.

4.8. Conclusion

By allowing the user to identify his memo function explicitly and by automatically creating entries in their system attribute-
lists, we achieve a reduction in the burden on the user. For many of the problems attempted, the experimental results show that this system performs the same manipulations in less time and with fewer garbage collections than either the host system or the system with permanent attributes. However, the optimum performance of the system depends on choosing the correct size for the system attribute-lists.

The main reason for the improved performance is that the system directly computes only those evaluations which either it has not previously encountered, or for which the attribute has already been deleted from the system attribute-list. Note that the simple promotion mechanism described in Section 4.4.2) ensures that a temporary attribute remains in the attribute-list as long as it is used frequently, (a process known as remembering).

The comparative study of performance shows that although an inappropriate choice of memo functions may result in a reduction in performance, (Section 4.7.3), a wrong choice for the size of the system attribute-list of a memo function has a much worse effect. Setting the size too large can bring the solution of the problem to a halt by exhausting the storage space available. Setting it too small can make the system perform less efficiently than the host system, (Section 4.7.1), because attributes are deleted before they are used.

The main drawback of this system is that the user has to specify appropriate limits on the sizes of the system attribute-lists. He may even have to alter these limits.
during the evaluation if the problem is a complex one. The object of our next chapter is, therefore, to try to eliminate this drawback by the introduction of the concept of a "dynamic system attribute-list".
5.1. Introduction

As pointed out in Chapter 4, there are two main drawbacks in the use of the system with fixed-size attribute-lists. First, the user must decide on the maximum number of entries in each system attribute-list. This task requires a detailed knowledge of the behaviour of the functions in an evaluation. As a result, a user who may be unfamiliar with the problem is still expected to make reasonable decisions regarding the number of entries in the system attribute-lists of the memo functions. Second, no matter how long a memo function remains inactive during an evaluation, its attributes are preserved, thus occupying storage space which could have otherwise been put to good use for other purposes.

The expression and function driven systems which are described in this chapter have dynamic attribute-lists. The system attribute-lists are dynamic in the sense that, first, there is no upper limit on the number of entries that they may contain and, second, a temporary attribute is preserved only as long as it is used in the evaluation. The criteria on which the life-time of the attributes depend are specified by the user and determined from the characteristics of the system and the evaluation.

We begin by studying the type of rules which are used to automatically control the size and the contents of the system attribute-lists. This is followed by a discussion on the two
different modes of memoization, (that is, the association of attribute-lists with functions or expressions). Although rather theoretical, this discussion is useful from two points of view. First, it emphasizes differences which arise when attribute-lists are associated with either functions or expressions. Second, it provides criteria by which to choose between an expression or function driven system by examining some of the relevant properties of a problem.

Subsequently, we give a brief description of the most important features of the expression and function driven systems implemented, and describe the commands which have been added to the user's repertoire of commands.

These systems are used to perform specific evaluations which have already been computed using the other attribute systems and the host system. In comparison with previous attribute systems, the use of dynamic attribute-lists further reduces the burden on the user, the evaluation time, and the number of garbage collections performed. The new systems are capable of performing complex manipulations which are not possible using the previous systems.

5.2. Basic Concepts

5.2.1. Dynamic attribute-list

The function driven system with fixed-size system attribute-lists described in Chapter 4 possesses the following drawbacks. First, the destruction of attributes of a given memo function begins once all the positions in the corresponding attribute-list are occupied and further attributes are to be created. The fact that many of the attributes of memo
functions which would have been used during a period of evaluation are destroyed leads to an increase in evaluation time. Second, whether or not a memo function is used often, the number of attributes that it can have is pre-determined and cannot be changed by the system. Third, preserving the attributes of those memo functions which are not used during a period of evaluation leads to inefficient use of storage space.

The use of dynamic attribute-lists removes these shortcomings. First, a temporary attribute is not destroyed unless conditions in the system demand it. Second, the attribute-lists of functions can expand or contract depending on the use of the attribute at different points in the evaluation. The set of rules used to relate the size of system attribute-lists to the storage space required by the evaluation are:

(a) Every time an attribute is created by the system, a life-time of \( m \) units is assigned to it:

\[
S = m
\]

Variable \( S \) is used to denote a typical life-time. The attribute created can be placed at any position in the attribute-list.

(b) Each time an attribute is used, its life-time is increased by \( q \) units:

\[
S = S + q
\]

and it is promoted towards the top of the attribute-list.

(c) When \( n \) units of storage space (or \( t \) units of time) have been used, the life-time of each temporary attribute is
reduced by $p$ units:

$$S \leftarrow S - p$$

If the life-time of an attribute fails to satisfy the inequality

$$S - e \geq 0$$

where $e$ is a pre-specified limit, then the attribute is deleted.

Unlike the system with fixed-size attribute-lists, (Section 4.4.2), promotion only affects the look-up time and does not influence the decision concerning the preservation or destruction of an attribute.

A fixed set of values for the parameters $m$, $q$, $p$, $n$ (or $t$) and $e$ may not be the best choice for all the different types of problems that the system is used to solve. Consequently, provision must be made so that, if the need arises, the user may re-set the values of any of these parameters.

5.2.2 Modes of memoization

Each of the attribute systems implemented for this project is either exclusively function driven or exclusively expression driven. This is to facilitate a comparative study of the performance of the systems. As a result, in the function driven system the user may only select the functions which are to be associated with attribute-lists, and in the expression driven system he may only select expressions which are to be associated with attribute-lists. With attribute systems containing all the features of the corresponding
expression and function driven systems, the user may choose to associate attribute-lists with both functions and expressions. In the remainder of this section we examine the consequences of each choice. The account given attempts to clarify the circumstances in which the use of an exclusively function driven system or an exclusively expression driven system is to be preferred.

Let us consider the evaluation of:

\[
\frac{d}{dx} (ax + b)
\]

where \(a\) and \(b\) are constants, according to the differentiation rules given in Section (B.2.1). If an attribute-list is associated with the differentiation function, according to the set of attribute house-keeping rules given in Section (4.4.2), the attributes generated in the course of the above evaluation are:

\[
\frac{d}{dx} (ax + b) = a; \frac{d}{dx} (ax) = a; \frac{d}{dx} (a) = 0; \frac{d}{dx} (b) = 0; \frac{d}{dx} (x) = 1
\]

If, instead, attribute-lists had been associated with the expressions \(ax + b\) and \(ax\), then during the above evaluation the attributes

\[
\frac{d}{dx} (ax + b) = a; \frac{d}{dx} (ax) = a
\]

would have been created and added to the appropriate attribute-lists. In other words, in the first method an attribute is generated every time the memoized function is evaluated, while in the second method an attribute is generated only when the argument contains a memoized expression.
In practical work, instances can easily arise where the choice of associating attribute-lists with functions or their argument expressions becomes a critical one. To explain this we use the following table:

<table>
<thead>
<tr>
<th></th>
<th>$f_1$</th>
<th>...</th>
<th>$f_j$</th>
<th>...</th>
<th>$f_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$a_{11}$</td>
<td>...</td>
<td>$a_{1j}$</td>
<td>...</td>
<td>$a_{1n}$</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$S_i$</td>
<td>$a_{i1}$</td>
<td>...</td>
<td>$a_{ij}$</td>
<td>...</td>
<td>$a_{in}$</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$S_m$</td>
<td>$a_{m1}$</td>
<td>...</td>
<td>$a_{mj}$</td>
<td>...</td>
<td>$a_{mn}$</td>
</tr>
</tbody>
</table>

where,

- $f_j$, $j = 1, 2, \ldots, n$ represent the functions in a problem,
- $S_i$, $i = 1, 2, \ldots, m$ represent all the argument expressions of the functions $f_j$ where $j = 1, 2, \ldots, n$,
- $a_{ij}$ is an integer variable, the value of which is zero if $S_i$ is not an argument expression of the function $f_j$ or with value equal to the number of times that the function $f_j$ is applied to the argument expression $S_i$ during the evaluation. The possibilities which are likely to arise can be described using this table.

1. If the function $f_j$ is used a large number of times during the evaluation, the sum of all non-zero $a_{ij}$ in the column of the table labelled $f_j$ is a large positive integer, that is,

$$
\sum_{i=1}^{m} a_{ij} >> C_j
$$  \hspace{1cm} (1)
where \( Q_j \) is the number of different arguments for the function \( f_j \). Furthermore, if such a function has a largely disjoint set of argument expressions, then, as far as the number of attributes in use is concerned, there is no difference in associating the attribute-lists with the function \( f_j \) or with each of its argument expressions. In either case, \( Q_j \) remains the maximum number of attributes in use with this function.

In situations where some of the argument expressions of the function \( f_j \), which has a high frequency of use, also appear as the argument expressions for other functions that may or may not satisfy relation (1), the consequences are as follows. If the attribute-list is associated with the function \( f_j \), the same number of attributes, \( Q_j \), are generated. If, instead, we decide to associate attribute-lists with one or more of the argument expressions of the function \( f_j \), this will increase the number of attributes created. This increase is caused by those memoized argument expressions of the function \( f_j \) that are also used by other functions.

In this particular case, if we are only interested in maintaining all the attributes of the function \( f_j \), then the function \( f_j \) is associated with an attribute-list. A specific choice of the attributes of \( f_j \) can be arrived at by associating attribute-lists with the appropriate arguments of \( f_j \). The expressions which are to have attributes must be carefully selected so that the system does not spend most of its time handling those attributes which arise from functions other than \( f_j \).
2. If the expression $S_j$ is used a large number of times as the argument of a large number of functions, then the sum of all the non-zero $a_{ij}$ in the row of the table labelled $S_i$ is a large positive integer, that is,

$$\sum_{j=1}^{n} a_{ij} \gg P_i$$  \hspace{1cm} (2)

where $P_i$ is the number of functions which take $S_j$ as their argument. In this case, associating an attribute-list with the argument expressions, $S_j$, means that there are at most $P_i$ attributes to be handled, (since there are $P_i$ functions which take $S_j$ as their argument expressions). However, generating the same attributes by memoizing the corresponding functions is likely to be unsatisfactory. This is because each of the memoized functions will give rise to all the attributes which can be associated with it, and in this way many attributes may be created which are not useful to the evaluation. Further, since extra storage space is required by attributes in the function driven system, (Section 3.6.3), this approach is likely to make poor use of the storage space available. In this case, therefore, the memoization with respect to common argument expressions is preferable.

3. The other possibilities which can arise in a problem are when nearly all the $a_{ij}$ have either large or small values. In this instance, the user must carefully select those functions and expressions which, when memoized, provide most of the appropriate attributes.

Although the structure of the table above was used in a general way to identify situations which are likely to arise in practice, for two reasons the actual use of such a table
for a real problem is not practicable. First the table can not be constructed until the problem has been solved because only then are the data available. Second, even for simple evaluations the collection and analysis of such data is a tedious and time consuming task.

5.3. Expression-driven system

5.3.1. Basic features

The main criticism of expression driven system with permanent attributes is that the user must do all the work. That is, he must decide on the expressions which are to be associated with attribute-lists, create the necessary attributes, and look after the house-keeping operations. Although this system is ideal for situations where there are few attributes, its use in a situation with a large number of attributes can place an unacceptable burden on the user.

In the expression driven system described in Section (3.3), objects with particular expressions as natural-values must be created by the user. Natural-values can only be chosen from those expressions which are made available by the user prior to the initiation of the evaluation cycle. In most practical situations, the user has inadequate knowledge of the evaluation process so that he cannot be expected to create those objects which prevent all repetitive evaluations.

An expression driven system with dynamic attribute-lists is an extension of the expression driven system with permanent attributes and includes all its capabilities. In the new system, either the user or the system, or both, can associate
expressions with attribute-lists. The system is responsible for the creation of temporary attributes and their housekeeping. The creation of permanent attributes and their house-keeping operation are left to the user.

The format of the attribute described in Section (3.3.1) was adequate for housing just one type of attribute, that is, permanent attributes. To keep permanent and temporary attributes separate, the structure had to be modified. As indicated in the diagram below, the value of the attribute-indicator is now a pointer to a cell whose car holds the start address of a list containing all the temporary attributes, and whose cdr holds the start address of a list containing all the permanent attributes.

To illustrate this structure consider the object, $Q$:

$$Q = x^2 + y^2 + z^2 + t^2$$

with the permanent attributes:

$$\frac{\partial Q}{\partial x} = 2x; \quad \frac{\partial Q}{\partial y} = 2y$$
and temporary attributes:
\[
\frac{\partial \Phi}{\partial z} = 2z; \quad \frac{\partial \Phi}{\partial t} = 2t
\]

The internal representation of the object \( \Phi \) then has the form:

![Diagram](image)

5.3.2. Dummy objects

It was pointed out in Section (3.3) that objects were created by selecting an expression and assigning a name to it. It was through these names, or their corresponding natural-values, that derived-values were obtained. Further, the user was able to perform the necessary attribute house-keeping operations by making use of the names of objects.

A "dummy object", on the other hand, is automatically created and has no name. A user, therefore, is neither aware of its existence nor has he access to its attribute-list. As in the case of a system with permanent attributes, an object-stack is used to maintain the connection between objects and
their attribute-lists. A symbolic description of the internal representation of a dummy object has the form:

The system can create and use dummy objects and their corresponding temporary attributes to assist evaluation. The user is not usually aware of such dummy objects which are operating behind the scenes, and rarely requires a knowledge of them. According to the commands described in Section (3.5.1), the user can only gain access to the attribute-list of an object through its name. With dummy objects, therefore, the user cannot gain access to their attribute-lists. The commands described in Section (5.5.2) enable the user both to obtain information about dummy objects and temporary attributes, and to transform dummy objects into permanent user defined objects.

5.3.3. Attribute house-keeping operations

In the expression driven system with dynamic attribute-lists, temporary attributes are being automatically created and discarded throughout the process of evaluation. Below,
we describe the way in which the sizes of the dynamic attribute-lists are related to the space required by the evaluation.

With each temporary attribute is associated an integer variable whose value depends on the space required by the evaluation and the rate of use of the attribute during the evaluation. The value of this variable is referred to as the life-time of an attribute, and the decision of the system concerning the preservation or removal of a temporary attribute is based on the value of its life-time. The set of rules, (Section 5.2.1), which are used by the system to determine the fate of each attribute is listed below.

(a) Once a temporary attribute is generated a life-time of four units is assigned to it.

(b) Every time a temporary attribute is used its life-time is incremented by two units, and the attribute, together with the attribute indicator and its associated value, is promoted one place towards the top of the system attribute-list.

(c) Every time the garbage collection cycle is invoked the life-time of all the temporary attributes is reduced by one unit, and those with a negative life-time are deleted from the system attribute-list.

The life-time parameters (i.e. 4, 2, 1, 1) which appear in these rules are built into the system. However, in circumstances where these life-time parameters are not suitable, a better choice can be arrived at by trial and error, and the parameters can be altered by using the commands initialize, increment, decrement and promote, a description of
which appears in Section (5.5.1).

Notice that the amount of information preserved, (i.e. - number of temporary attributes), has become a dynamic entity dependent on the storage space available for the evaluation. This is because as the size of the free word list is reduced the garbage collection cycle is invoked more frequently, thus reducing all life-times. According to rule C, this leads to a faster removal of temporary attributes. Hence the portions of storage space used to accommodate the attributes and to serve other purposes are closely interrelated throughout the evaluation.

5.3.4. New tasks for the garbage collection routine

Garbage collection in the host system is limited to marking all accessible cells which are to be preserved, returning all the unmarked cells, (inaccessible cells), to the free word list for future use, and removing the mark from the accessible cells. In the expression driven system with dynamic attribute-lists, two additional tasks are imposed upon the garbage collection routine:

(a) the deletion from the system attribute-list of temporary attributes whose life-time is exhausted,

(b) the release of those positions in the object-stack occupied by dummy objects which have been deleted.

Below we describe the way in which the garbage collection routine is modified to perform these two tasks.

The point at which temporary attributes are reviewed within the garbage collection cycle is of vital importance.
This is because unwanted attributes must be discarded so that their constituent cells can be collected during the later part of the same garbage collection cycle. Hence the review of the temporary attributes must precede the marking process. Diagrammatically we have:

Two types of objects are present in the system, those defined by the user, and the dummy objects which are created by the system. As pointed out in Sections (3.3.2) and (5.3.2), each object occupies a position in the object-stack. The names of user defined objects may be used in subsequent evaluations and so their corresponding entries in the object-stack can only be removed by an explicit command from the user. Since the user has no direct access to dummy objects created these objects are deleted by the system. This means that the system must be able to clear entries in the object-stack which are no longer needed so that they can be re-used.

It is possible that while the object-value of an object is occupying a position in the object-stack, all the expressions referencing this particular entry have ceased to exist. If a natural-value is accessed in the process of marking accessible cells, then the natural-value itself is marked. After marking is complete dummy objects which are
still in use are then determined by the mark on their natural-values.

Note that if the marking of the entries of the object-stack is left until the very end, the following situation can arise. Before the marking of the constituent cells of an object-value is initiated, the first cell of the natural-value is examined for the presence of the mark. If it is marked then expressions referencing the corresponding entry of the object-stack must still exist in the system, and no further action is taken. If the first constituent cell of the natural-value has not been marked, all the expressions referencing the corresponding entry of the object-stack have ceased to exist, and the position in the object-stack is released for future use. The flow-chart of the above operation is provided in Fig. 5.0.

In practice, the contents of the position released are set to zero, but positions containing zero and non-zero values are not collected together. Instead, the positions in the object-stack are used in a circular fashion. In order to create subsequent objects, therefore, the next position in the object-stack containing zero is used.

5.3.5. Operation of the interpreter

If the first argument of apply is a marked function, then the attribute-list of the memo element of the argument expression is searched for the required value. If the required attribute exists, the life-time of the attribute is incremented, the attribute is promoted by one place, and the derived-value is returned as the value. In the absence of the attribute, the
Fig. 5.0. Flow-chart shows the manner in which the positions in the object-stack are reviewed and the unused positions are released.
required value is calculated and the corresponding attribute is created before returning the value. The appropriate modifications to the host system interpreter appear in the form of the flow-chart given in Fig. 5.1. This flow-chart fragment replaces the dashed rectangular box in the flow-chart of apply given in Section (A.1). The modifications result in the size of the interpreter of the host system being increased by 1731 words.

Example

We use this example to show how dummy objects and temporary attributes are created during the evaluation. In this example no objects are created by the user. The process of differentiation is carried out according to the set of rules listed in Section (B.2), and the steps of the evaluation are annotated using the notations described in the same Section. Consider the evaluation of:

\[ \frac{d}{dx} (a + (x - b)^n)^m \]

where \(a\) and \(b\) are constants and \(m\) and \(n\) are integers. For the sake of brevity, we ignore the creation of dummy objects with atomic natural-values.

1. \[ \frac{d}{dx} (a + (x - b)^n)^m = m(a + (x - b)^n)^{m-1} \] (D6)

\[ \frac{d}{dx} (a + (x - b)^n) \]

\[ \frac{d}{dx} (a + (x - b)^n) \]

\[ \frac{d}{dx} (a + (x - b)^n) \]

At this point a dummy object with the natural-value:

\[ (a + (x - b)^n)^m \]

is created. However, its corresponding temporary attribute cannot be formed until the evaluation (1) is complete.
Fig. 5.1. Flow-chart shows the modifications which had to be made to the interpreter of the host system in order to make it workable for the expression driven system with dynamic attribute-lists.
\[ \frac{d}{dx}(a + (x - b)^n) = \frac{d}{dx}(a) + \frac{d}{dx}(x - b)^n \]  \hspace{1cm} (D2)

A dummy object with the natural-value \((a + (x - b)^n)\) is created.

\[ \frac{d}{dx}(a) = 0 \]  \hspace{1cm} (D0)

No dummy object is created since the natural-value, \(a\), is atomic.

\[ \frac{d}{dx}(x - b)^n = n(x - b)^{n-1} \frac{d}{dx}(x - b) \]  \hspace{1cm} (D6)

Dummy object with the natural-value \((x - b)^n\) is created.

\[ \frac{d}{dx}(x - b) = \frac{d}{dx}(x) - \frac{d}{dx}(b) \]  \hspace{1cm} (D3)

A dummy object with the natural-value \((x - b)\) is created.

\[ \frac{d}{dx}(x) = 1 \]  \hspace{1cm} (D1)

No dummy object is created since the natural-value, \(x\), is atomic.

\[ \frac{d}{dx}(b) = 0 \]  \hspace{1cm} (D0)

No dummy object is created since the natural-value, \(b\), is atomic.

\[ \frac{d}{dx}(x - b) = 1 \]  \hspace{1cm} (SUBE)

The above attribute is added to the system attribute-list of the dummy object with \((x - b)\) as the natural-value.

\[ \frac{d}{dx}(x - b)^n = n(x - b)^{n-1} \]  \hspace{1cm} (SUBE)

The above temporary attribute is added to the attribute-list of the corresponding dummy object.
$\frac{d}{dx}(a + (x - b)^n) = n(x - b)^{n-1}$

The above temporary attribute is added to the attribute-list of the dummy object with the natural-value $(a + (x - b)^n)$.

$\frac{d}{dx}(a + (x - b)^n)^m = mn(a + (x - b)^n)^{m-1}(x - b)^{n-1}$

The above temporary attribute is added to the attribute-list of the dummy object in (1).

In the first order differentiation of the above expression, none of the attributes created are used to assist in the evaluation. The trace of:

$$\frac{d^2}{dx^2}(a + (x - b)^n)^m$$

as evaluated by this system is given in Section (B.3).

5.4. Function driven system

5.4.1. Introduction

The function driven system with dynamic attribute-lists relieves the user from the task of estimating a suitable limit on the number of entries in an attribute-list. Apart from the different rules, (Section 5.3.3), used to control the growth of dynamic attribute-lists, this system inherits all the features of the function driven system with fixed-size attribute-lists which was described in Chapter 4, in particular:

(i) memo functions are identified by the user,

(ii) permanent and temporary attributes are kept apart in the same manner,
(iii) an attribute-list is attached to the property list of a memo function through the property indicator ROTE.

Finally, due to the absence of dummy objects in this system, the only extra task of the garbage collection routine is the review of temporary attributes (Section 5.3.4).

5.4.2. Operation of the interpreter

The modifications to the interpreter shown in the flow-chart of Fig. 5.2 closely resemble those used to create the expression driven system with dynamic attribute-lists, (Section 5.3.5). Essentially, this flow-chart fragment replaces the dashed rectangular box in the flow-chart of apply given in Section (A.1), and these modifications cause the size of the interpreter of the host system to increase by 1617 words. No illustrative example is given in this case. This is because, apart from the organisational differences, the same attributes are created in the same manner as in the expression driven system.

The trace of the evaluation:

$$\frac{d^2}{dx^2} (a + (x - b)^n)^m$$

as evaluated by the system is identical to the trace obtained using the expression driven system with dynamic attribute-lists, (Section B.3).

5.5. Repertoire of commands

The commands which have been added to the expression and
Fig. 5.2. Flow-chart shows the modifications which had to be made to the interpreter of the host system in order to make it workable for the function driven system with dynamic attribute-lists.
function driven systems with permanent attributes, (Section 3.5) are described below. All these commands are hand coded into the interpreter of the host system. The descriptions of the commands memo and promote are identical to those provided in Sections (3.5.2) and (4.5) respectively. M-expression definition of some of these commands, (as indicated below), together with examples of their use appear in Appendices B and C.

5.5.1. Repertoire of commands: EDS and FDS

The commands described below are common to both the expression and the function driven systems.

**increment** {n} :SUBR

Every time a temporary attribute is used, its life-time is increased by a number of units. increment is used to alter this number. No error message is generated and no value is returned. Examples of its use appear in Section (B.6).

**decrement** {n} :SUBR

In the course of execution, every time the garbage collection routine is invoked, the life-times of all the temporary attributes are reduced by a number of units. decrement is used to alter this number. No error message is generated and no value is returned. Examples of its use appear in Section (B.6).

**initial** {n} :SUBR

As each temporary attribute is created a certain unit of life-time is assigned to it. This command is used to alter this initial life-time. No error message is generated and no
value is returned. An example of the use of this command appears in Section (B.6).

**garblnt (n)** : **SUBR** pseudo function

This command is used to define the minimum number of cells under which the system will abandon the current evaluation. The user is informed if the value of the argument n is greater than the number of cells in the free word list. No value is returned. An example of its use appears in Section (B.6).

5.5.2. **Repertoire of commands**: EDS

**prinstck (x)** : **SUBR**

prinstck is used to display object-values or natural-values of a specified set of entries in the object-stack. The choice of natural-values or object-values which are to be displayed is determined interactively once the routine is entered. Depending on the format of the argument, x, four possibilities arise:

(a) \( x = 0 \)

In this case a set of positive integers corresponding to the entries in the object-stack being used is printed out.

(b) \( x = \text{NIL} \)

In this case the natural-values or the object-values of all the positions in the object-stack occupied by dummy objects are displayed.

(c) \( x = n \)

In this case the natural-value or the object-value of the specified entry in the object-stack is displayed.
In this case the natural-values or the object-values of the specified range of entries in the object-stack are displayed.

This facility is only used for displaying dummy objects because, in the case of objects defined by the user, the values can be displayed by other means, (Section 3.5).

An error message is generated if the specified entry numbers do not exist in the object-stack. The M-expression definition of `prinstck` together with an example of its use appear in Section (C.3).

```lisp
keepobjs (x) :SUBR pseudo function
  Dummy objects can be preserved by assigning names to them. The function `keepobjs` takes a list of pairs as its argument:
  
  ```lisp
  ((na_1 n)(na_2 n_2) ... (na_i n_i) ... (na_n n_n))
  ```lisp

  Each na_i is the name of a new object which is to be formed, and each n_i is an integer which is the label of the entry in the object-stack holding the object-value of the dummy object being named. Notice that the command `prinstck` described above is used to determine the values, n_i, and their corresponding objects. An error message is generated every time the specified entry number does not exist in the object-stack.

  The evaluation of each pair-list gives rise to the structure:
5.6. Analysis of practical results

Once again the three problems described in Section (2.3) are used to compare the performance of the expression and function driven systems with dynamic attribute-lists, with the performance of other systems available. In this case, it is also necessary to investigate the effect of giving different values to some of the life-time parameters.

Additional evaluations arising from other problems have also been attempted. They are:

(a) The Presburger Algorithm, (10), which is used in establishing the correctness of theorems, (Section D.5). This problem was attempted using the host system and a function driven system.
(b) The function _infix_, (Section D.7), which is used to convert expressions in prefix-polish notation into their equivalents in infix notation. This evaluation was carried out using the host system and the expression driven system.

Most of the data used to plot graphs and to construct the tables shown are taken from the tables of the experimental results provided in Section (B.5).

5.6.1. **Problem 1** (Differentiation 1: Section 2.3.1)

**Burden on the user**

In the expression driven system, since differentiation is a marked function, (a function defined by the user, (4)), no additional commands are needed to solve this problem. In the function driven system, on the other hand, the differentiation function must be declared as a memo function so that the link between the differentiation function and its attribute-list is established. In comparison with the host system, the use of systems with dynamic attribute-lists places very little extra burden on the user since no user defined objects and attributes are created.

All the evaluations in this section were carried out using a free word list of 7000 cells. The commands needed to solve the problem are given in Section (B.6.1).

**Storage space**

As the table in Fig. 5.3 shows, in comparison with the host system and the systems with permanent attributes there...
<table>
<thead>
<tr>
<th>Systems used</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>72</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EDS: 2 permanent attributes</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>35</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: 2 permanent attributes</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>52</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-lists</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>88</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-lists</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>FDS: dynamic attribute-lists</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5.3. Table shows the number of garbage collections required to carry out the evaluation of $d^p_\frac{d}{dx} (a + (x - b)^n)^m$ for $1 \leq p \leq 9$ using the specified systems and a free word list of 7000 cells.

<table>
<thead>
<tr>
<th>Systems used</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>47</td>
<td>139</td>
<td>528</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EDS: 2 permanent attributes</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>34</td>
<td>109</td>
<td>408</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FDS: 2 permanent attributes</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>38</td>
<td>121</td>
<td>455</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-lists</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>23</td>
<td>40</td>
<td>77</td>
<td>343</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EDS: dynamic attribute-lists</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>22</td>
<td>35</td>
<td>67</td>
<td>133</td>
<td>275</td>
<td>743</td>
</tr>
<tr>
<td>FDS: dynamic attribute-lists</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>22</td>
<td>37</td>
<td>72</td>
<td>146</td>
<td>344</td>
<td>917</td>
</tr>
</tbody>
</table>

Fig. 5.4. Table contains the time required to carry out the evaluation of $d^p_\frac{d}{dx} (a + (x - b)^n)^m$ for $1 \leq p \leq 9$ using the specified systems and a free word list of 7000 cells.
is no difference in the number of garbage collections required up to the third order of differentiation. This is because the temporary attributes which are being created and used in the intermediate stages are trivial and a free word list of 7000 cells is rather large for these simple evaluations. From the fourth order of differentiation onward, there is a reduction in the number of garbage collections required to solve the problem using systems with dynamic attribute-lists, and this reduction becomes larger as the order of differentiation increases. For instance, for fourth order of differentiation, the number of garbage collections required to solve the problem in the systems with dynamic attribute-lists is reduced by a factor of 3 in comparison with the host system, and by a factor of 2 in comparison with the systems with permanent attributes, and is the same as for the function driven system with fixed-size attribute-lists. With the sixth order of differentiation, however, although the number of garbage collections required to solve the problem in the systems with dynamic attribute-lists is comparable with that in the function driven system with fixed-size attribute-lists, it is reduced by a factor of approximately 18 in comparison with the host system, and by factors of 9 and 13 in comparison with the expression and function driven systems with permanent attributes, respectively.

The problem

$$\frac{d^7}{dx^7} (a + (x - b)^n)^m$$

which could not be solved in the host system and systems with permanent attributes requires only 5 and 7 garbage collections,
respectively, for its solution in the expression and function driven systems with dynamic attribute-lists. The same manipulations required 08 garbage collections in the function driven system with fixed-size attribute-lists. The inadequate number of entries in the fixed-size attribute-list, and the fact that some of the unused complex attributes are not discarded from the attribute-list, leads to substantial increases in the number of garbage collections required.

Finally, the evaluation of

$$\frac{d^8}{dx^8}(a + (x - b)^n)^m$$

which could not be solved using the function driven system with the fixed-size system attribute-list of 100 entries, can be solved using expression and function driven systems with dynamic attribute-lists. Further, notice that in both systems the number of garbage collections required is increasing slowly with respect to the order of differentiation. Therefore, it seems reasonable to expect these systems to be capable of carrying out higher order differentiation of the same expression, although no further attempt has been made to show this. In this particular case, therefore, the use of dynamic attribute-lists has a marked effect on the manipulative power of the system. Further, we notice that as order of differentiation increases the expression driven system begins to perform more efficiently than the function driven system.

**Time of evaluation**

Both systems with dynamic attribute-lists maintain the
same speed in performing repeated differentiations up to the fourth order, (See table of Fig. 5.4). For higher order differentiations, however, the expression driven system performs faster than the function driven system. This is partly because of the difference in the storage space requirement by attribute-lists in the two systems (Section 3.6.3), and partly because of the number of unsuccessful searches of attribute-lists in the function driven system.

The curves of Fig. 5.5 show the variation of the number of times that attribute-lists were searched and the number of times that the required attributes were obtained from the attribute-lists with respect to the order of differentiation in the systems with dynamic attribute-lists. The coincidence of the curves in the expression driven system with dynamic attribute-lists for most orders of differentiation is because an attribute-list is searched only if the argument list of the differentiation function contains a reference to an object, (Section 5.3.5). In this particular example all object references which appear as parts of arguments of differentiation functions have one attribute since differentiation is the only marked function. Consequently, if an object exists then it has the appropriate attribute. Note that, in problems where expressions appear as arguments (or parts of arguments) of a number of marked functions, the total number of times attribute-lists are searched may not be equal to the number of times that the required values are obtained from the attribute-lists.

Finally, the function driven system with fixed-size attribute-lists performs the evaluation.
Fig. 5.5. The above curves show the variation of the number of times that attribute-lists are searched and the number of times that attributes are obtained from the attribute-lists with respect to the order of differentiation in the system with dynamic attribute-lists.
\[ \frac{d^7}{dx^7} (a + (x - b)^n)^m \]

in 343 seconds, while the expression and function driven systems with dynamic attribute-lists perform the same evaluation in 133 and 146 seconds respectively. Although the speeds of the systems just mentioned are comparable up to the sixth order of differentiation, the indicated differences are due to the inadequate number of entries in the system attribute-lists of the function driven system with fixed-size attribute-lists, (Section 4.7.1) and the fact that 100 complex attributes have taken up a considerable amount of storage space. Further, the systems with dynamic attribute-lists perform approximately 7 and 6 times faster than the host system and the systems with permanent attributes, respectively.

5.6.2. Problem 2 (Differentiation 2: Section 2.3.2)

Burden on the user

As explained in Section (5.6.1), the only action by the user is to declare the differentiation function as the memo function in the function driven system. In solving this problem no objects or permanent attributes are created. The sets of commands needed to solve this problem in the expression and function driven systems with dynamic attribute-lists appear in Section (B.6.2).

Storage space

The table:
contains the number of garbage collections required to solve the problem using the specified systems and with a free word list of 7000 cells. As the table shows, the expression driven system with dynamic attribute-lists involves the minimum number of garbage collections. The difference in the number of garbage collections required to solve the problem in the systems with dynamic attribute-lists is due to the difference in the storage space required for the attribute-lists in the two systems, (Section 3.6.3). This difference is clearly shown in the curves of Fig. 5.6 where the number of garbage collections required is plotted against the storage space available.

The additional number of garbage collection cycles invoked in the function driven system with fixed-size attribute-lists is due to the reasons given in the previous section, (Section 5.6.1). The small difference in the number of garbage collections required between the systems is due to a reasonable choice of the values of the life-time parameters and the number of entries in the fixed-size system attribute-lists, (Sections 4.7.1 and 5.3.3).
Fig. 5.6. The above curves show the variation of the number of garbage collections required to solve the differentiation 2 problem in the expression and function driven systems with dynamic attribute-lists with respect to the storage space available.
With a free word list of 7000 cells, the use of expression and function driven systems with dynamic attribute-lists reduces the number of garbage collections required to solve the problem by factors of 2.6 and 7.6 in comparison with the systems with permanent attributes and the host system, respectively.

**Time of evaluation**

In contrast to the function driven system with fixed-size attribute-lists, (Section 4.6.3), the rates of creation and destruction of attributes differ widely during the evaluation. In order to give the reader some appreciation of this, Fig. 5.7 shows the curves for the rates of creation and destruction of attributes with respect to the garbage collection cycles invoked to solve the problem using a free word list of 5000 cells.

The table:

<table>
<thead>
<tr>
<th>systems</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>417</td>
</tr>
<tr>
<td>EDS: permanent attributes</td>
<td>167</td>
</tr>
<tr>
<td>FDS: permanent attributes</td>
<td>178</td>
</tr>
<tr>
<td>FDS: fixed-size attribute-lists</td>
<td>95</td>
</tr>
<tr>
<td>EDS: dynamic attribute-lists</td>
<td>65</td>
</tr>
<tr>
<td>FDS: dynamic attribute-lists</td>
<td>71</td>
</tr>
</tbody>
</table>

shows the time required to solve the problem using the specified systems with a free word list of 7000 cells. As shown in the table, the systems with dynamic attribute-lists
Fig. 5.7. The two curves convey the rate of creation and deletion of attributes with respect to the garbage collection cycle.
solve the problem 6.4 and 5.8 times faster than the host system; 2.5 and 2.3 times faster than the expression driven system with permanent attributes, and 2.7 and 2.5 times faster than the function driven system with permanent attributes, respectively. We also note that systems with dynamic attribute-lists are slightly faster than the function driven system with fixed-size attribute-lists.

The function driven system with fixed-size attribute-lists is slower than the systems with dynamic attribute-lists because:

(a) increasing the number of entries in the fixed-size system attribute-list causes more storage space to be taken up by temporary attributes. Consequently, the look-up time and the number of garbage collections are substantially increased,

(b) reducing the number of entries in the fixed-size system attribute-list causes the attributes required to be deleted before they are required again.

The curves showing the variation of the evaluation time with respect to the storage space available for the expression and function driven systems are given in Fig. 5.8. Two of the factors which are responsible for the gap between these curves are:

(a) the different numbers of garbage collection cycles invoked to solve the problem in the two systems, (Section 3.6.3),

(b) the difference in the look-up time in the expression and function driven systems, (Section 3.6.3).
Fig. 5.8. For the differentiation 2 problem, these curves show the variation of the evaluation time with respect to the storage space available in FDS and EDS with dynamic attribute-lists.
5.6.3. Problem 3 (prime numbers: Section 2.3.3)

To make use of temporary attributes, the functions \texttt{prs} and \texttt{prime} must be declared as memo functions in the function driven system. The performances obtained using either the iterative or the recursive definitions of \texttt{prs} under both the systems with dynamic attribute-lists are identical with those obtained using the function driven system with fixed-size system attribute-lists, (whose performance is described in Section (4.6.3).

The systems perform equally well because, even with a free word list of 3000 cells, there is enough storage space for all the attributes needed. In the system with dynamic attribute-lists the garbage collection cycle is invoked only once during the evaluation of \texttt{prs}, and consequently no temporary attribute is destroyed due to its positive life-time.

Similarly, in the function driven system with fixed-size attribute-lists, the system attribute-list is large enough to hold all the temporary attributes. As a result, the necessary attributes are created only once and stay in the attribute-lists during the remainder of the evaluation.

5.7. Parameters of the systems

It is not possible to specify a set of life-time numbers which is suitable for all types of evaluation. On the other hand, once the nature of the evaluation is known, one of the following possibilities can arise:

(a) The life-time numbers which are built into the system are suitable for the manipulation being performed.
(b) The values of some or all of these life-time numbers must be altered to suit the evaluation but these remain fixed throughout the evaluation.

(c) The value of the life-time numbers depends on the various phases of evaluation. In this case the user can intervene and alter the value of the life-time numbers directly or devise an algorithm which achieves this automatically according to the requirements of the evaluation.

It must be remarked that the selection of a general purpose algorithm for setting life-time parameters is unlikely to cater for all situations.

A number of problems are used to investigate the variation of the evaluation time with respect to different values of the life-time parameters, (Section D.6). Due to the cost and the time needed to carry out the necessary tests, only one of the life-time parameters is allowed to vary while the values of the remaining parameters are kept constant. In brief, a bad choice of the values of the life-time parameters can lead to the following situations:

(a) attributes are removed from attribute-lists prior to their use. This arises if the reduction of the life-time of the attributes at every garbage collection is large, or the initial life-time assigned to an attribute or its increment every time it is used is too small,

(b) attributes are not removed quickly enough. Obviously, as the evaluation continues, attributes accumulated exhaust the reservoir of free cells bringing the system
to a halt.

To prevent the first situation arising, the values of the life-time parameters must be correctly set. To overcome the second difficulty, a simple mechanism has been devised whereby the life-times of all system generated attributes are repeatedly reduced by a fixed quantity, until the system has collected enough cells to continue with the solution of the problem. The limit on the number of free cells under which the solution of the problem will not be continued, and under which the life-time of temporary attributes is repeatedly reduced, is specified by the user. garblint, whose description is given in Section (5.5.1), is used for this purpose. Notice that even in this scheme it is the less frequently used attributes which are being deleted from the attribute-lists.

The likelihood that, even with the deletion of all the temporary attributes, the number of free cells available may fall below the specified threshold cannot be ruled out. This is likely to be caused either by a wrong choice of the threshold limit by the user, or by the presence of an infinite recursion, or by the fact that the problem is too large for the system.

5.8. Conclusion

The systems with dynamic attribute-lists place almost no additional burden on the user and perform most manipulations faster and with fewer garbage collections than the other systems described in the previous chapters. The difference in performance between expression driven and function driven
systems becomes noticeable as the complexity of the problems increases. This difference in performance is due to the reasons given in Section (3.6.3).

The performance of the systems with dynamic attribute-lists depends on the choice of suitable values for their lifetime parameters, enabling evaluations to be performed which could not be performed by any of the previous systems. This increase in the manipulative power of the system is shown in Section (5.6.2).

Assigning large positive life-times to system generated attributes will exhaust the storage space available and bring the solution of the problem to a halt. Decreasing the lifetime of temporary attributes by a large amount at each review causes useful attributes to be deleted from the system attribute-lists, and so increases the evaluation time.

So far we have concentrated on the manner in which attributes are created, controlled and used, without paying any attention to the properties of attributes themselves. When performing a complex evaluation, it may be useful to control the creation and retention of attributes by making use of the attribute properties themselves. In the next chapter we investigate the application of this concept to some specific evaluations.
6.1. Introduction

When using a system with attribute facilities to solve a problem, a compromise has to be found between the use of storage space by attributes available and the use of storage space for the rest of the evaluation. For example, when using the system with fixed-size attribute-lists to solve a problem, it is not practicable to provide all the attribute-lists with an adequate number of entries, otherwise the storage space used by temporary attributes will exhaust the free word list and so bring the solution of a problem to a halt. The shortage of storage space in the systems with dynamic attribute-lists leads to a rapid deletion of temporary attributes from attribute-lists, and consequently increases evaluation time due to the re-construction of attributes.

For both systems, therefore, the solution to the drawbacks seems to be in a smaller number of more relevant temporary attributes - creation of significant attributes or suppression of insignificant ones.

In the following sections descriptions are given of two methods for making use of the characteristics of the functions involved in an evaluation to reduce the number of attributes by providing criteria to decide which temporary attributes are created. This is followed by a description of the implementation of the additional commands provided for the user. Finally, the problems described in Chapter 2 are used
to demonstrate the use of the new features and their effects on the performance of evaluations.

6.2. Basic Concepts

In the simple attribute systems which were described in Chapter 3, the user creates objects and gives them permanent attributes. As the evaluation continues, the user may decide to add to, to re-arrange or to delete these objects and their attributes. Consequently, not all the permanent attributes are subjected to the same fixed set of house-keeping rules. Instead, the user, by making use of his knowledge of the problem and the role which permanent attributes play in the various stages of the solution, can decide when to create (and when to destroy) a particular attribute. What the user is doing is selecting precisely those attributes which are useful to the system.

In the systems with temporary attributes described in Chapters 4 and 5, once attributes are created, under conditions specified by the user, all attributes are subject to exactly the same house-keeping rules. In a system with fixed-size attribute-lists, especially if the attribute-lists are not very large, (Section 4.7.1), the creation of relatively unimportant temporary attributes can cause the deletion of the significant ones, thus increasing the time of evaluation. In a system with dynamic attribute-lists, a reduction in the size of the free word list causes the garbage collection cycle to be invoked more frequently, which in turn reduces the life-time of every attribute, so that significant attributes are eventually deleted from the system attribute-lists, thus increasing the time of evaluation (Section 5.3.3).
What about evaluation time?
In any system, the presence of unwanted attributes increases the look-up time and reduces the effective size of the free word list, thus further increasing the time of an evaluation.

To counter the problems described above, we need to design systems which can select appropriate temporary attributes. In the simple systems described, the criteria by which a system selects appropriate temporary attributes are provided by the user. Such criteria are based upon properties of the functions, their arguments and their values, used in the evaluation.

The approach which has been selected is to organise the system so that before it creates a temporary attribute, it checks that the constituents of the attribute being constructed satisfy conditions which have been imposed by the user. The conditions to be satisfied by an attribute are specified in a new type of function which we shall refer to as a "selectivity function". A selectivity function is essentially a predicate function; for example, a selectivity function to prevent the creation of temporary attributes corresponding to the differentiation of a constant expression would take the form "do not create an attribute if the expression being differentiated is a constant".

In the systems described in chapters 4 and 5, when the value of a marked function is required, it is either obtained from its attribute-list or is evaluated using the definition of the function. In the latter case the system also generates a corresponding temporary attribute for use in subsequent evaluations. In the presence of a selectivity function, a
temporary attribute is only created when the selectivity function is satisfied. Hence, every time the definition of the function is used to calculate its value, the selectivity function is also evaluated. Diagrammatically this process can be expressed as:

From the diagram we see that in an expression driven system, a selectivity function is evaluated every time a marked function associated with a selectivity function is evaluated, and in a function driven system a selectivity function is evaluated every time a memo function associated with a selectivity function is evaluated. Thus, due to the frequency with which they are evaluated, the time required to evaluate a selectivity function must be kept to a minimum. Consequently, the provision of complex, general purpose selectivity functions is not appropriate.

This last observation leads to the provision of another facility which allows the user to specify that temporary attributes are to be created only as a result of evaluating a marked or memo function in a particular way. Given such a
facility, it is possible to use the structure of the function itself to either eliminate the need for a selectivity function or to simplify the selectivity function itself. To see how such a facility is provided consider the definition of a function using the conditional expression formalism. A conditional expression has the form:

\[(P_1 \rightarrow S_1; P_2 \rightarrow S_2; \ldots ; P_i \rightarrow S_i; \ldots ; P_n \rightarrow S_n)\]

where each \(P_i\) is a predicate and each \(S_i\) an expression. The value of the conditional expression is the value of \(S_i\) corresponding to the leftmost true predicate \(P_i\).

In the conditional expression, \(P_i\) is a set of conditions which, when satisfied, select the \(i\)th alternative value \(S_i\):

\[P_i \rightarrow S_i\]

Provided that the conditions used in the definition of the function include the conditions which would be used in constructing a selectivity function, there is no need for a separate selectivity function if the user is able to mark the alternative in the function definition rather than the function as a whole. A diagrammatic representation of this process is provided below.
If the conditions $P_1$ do not include all the conditions which are to be tested by the selectivity function, then a selectivity function is constructed to cover the conditions which are not included in $P_1$. In this case the diagram is modified as follows:

In the systems described, the task of selecting the appropriate attributes to control the generation of temporary attributes and the construction of selectivity functions is left to the user. Since, at the time of defining a function, the user is aware of the implication of its structure, the selectivity mechanism can be embedded in the function being defined without exerting a substantial additional burden on the user. Where possible, selectivity should be based on the structure of the functions involved because this:

(a) simplifies the selectivity functions required (or totally eliminates them),

(b) reduces the number of times that a selectivity function is evaluated.
6.3. Expression driven system

6.3.1. Basic features

As described in Section (5.3.2), each dummy object has one object-value. The two elements which make up an object-value are pointers to its natural-value and to its attribute-list. In addition, as shown in Section (5.3.5) by the flow-chart representing the operation of the interpreter, a dummy object must either exist or must be created before attributes are created. Thus one way to avoid the creation of unwanted attributes, is to avoid creating dummy objects. In the expression driven system, therefore, selectivity functions are used to control the creation of dummy objects.

Selectivity functions, which are constructed by the user, must be set up so that they are easily accessible. This is achieved by linking selectivity functions to marked functions selected by the user. In this way, as long as we have access to a function we can gain access to its selectivity function (if it has one). In this implementation, the selectivity functions are attached to the property lists of marked functions as the value of the property indicator SLCT.

6.3.2. Operation of the interpreter

In order to implement the selectivity feature in an expression driven system, further modifications had to be made to its interpreter as shown by the flow-chart in Fig. 6.1. The circles labelled E and H are the points at which this flow-chart fragment is inserted into the flow-charts of Figs. 4.1, 5.1. and 5.2. The additional modifications increased the size of the interpreter of the expression driven system.
**Fig. 6.1.** Flow-chart represents the modifications which had to be made to the interpreter of the system with temporary attributes in order to use the selectivity method which makes use of a separate selectivity function.

The circles labelled C and D are the points at which the above fragment is inserted in the flow-chart of Figs. 4.1., 5.1., and 5.2.

**Fig. 6.2.** Flow-chart represents the modifications which had to be made to the interpreter of the system with temporary attributes in order to allow for both methods of selectivity.
by 111 words. It is clear from this flow-chart that the only situation where a dummy object is not created is when the value of the selectivity function is false.

Example

To illustrate the process of evaluation, let us compute

\[
\frac{d}{dx} (a^2 + (x - b)^n)^m
\]

where a, b are constants and m, n are integers. In this evaluation we wish to create only those attributes corresponding to the differentiation of expressions which are raised to an integral power other than 1. Below we present the process of evaluation using the differentiation function and the conventions given in Section (E.2). We use SF as the name of the selectivity function - notice that it operates on the arguments of the marked function.

1. \[
\frac{d}{dx}(a^2 + (x - b)^n)^m = m(a^2 + (x - b)^n)^{m-1}
\] (D6)

Since \(a^2 + (x - b)^n\) is an expression raised to the power \(m\), the requirement of the selectivity function, SF, is satisfied and a dummy object with natural-value \((a^2 + (x - b)^n)^m\) is created.

\[
\frac{d}{dx}(a^2 + (x - b)^n) = \frac{d}{dx}(a^2) + \frac{d}{dx}(x - b)^n
\] (D2)

\[
\text{SF}\{(a^2 + (x - b)^n)}\) = F
\]

Since \((a^2 + (x - b)^n)\) is an expression raised to the power \(m\), the requirement of the selectivity function, SF, is satisfied and a dummy object with natural-value \((a^2 + (x - b)^n)^m\) is created.

The argument expression does not satisfy the selectivity function so no dummy object is created.
A dummy object with the natural-value $a^2$ is created.

At this point the attribute $\frac{d}{dx}(a^2) = 0$ is added to the attribute-list of the dummy object with natural-value $a^2$.

A dummy object with the natural-value $(x - b)^n$ is created.

At this point the attribute $\frac{d}{dx}(x - b)^n = 0$ is added to the attribute-list of the dummy object with natural-value $(x - b)^n$.
At this point the attribute \( \frac{d}{dx} (x - b)^n = n(x - b)^{n-1} \) is added to the attribute-list of the dummy object with natural-value \( (x - b)^n \).

The above attribute is added to the attribute-list of the first dummy object created.

During this evaluation the selectivity function, SF, was computed eight times. Its use reduced the number of temporary attributes created from 8 to 3.

The purpose of the example is to demonstrate the manner in which a selectivity function is used in an evaluation. In this example the reduction in the number of temporary attributes created is not of any practical importance in the evaluation.

Selectivity and the structure of functions

In the last example, the selectivity function, SF, was computed at every entry to the differentiation function and, as will be seen later, the time used for these repeated evaluations is significant, (Section 6.6). As pointed out in Section (6.2), no separate selectivity function is necessary if a mechanism to control the creation of attributes is built into the function definition. In the example above, if every time the alternative:
in the definition of the differentiation function, (Section B.2.1) is selected, a dummy object with a corresponding temporary attribute is created, then the same result is achieved as with the selectivity function. Thus using the structure of the differentiation function in this way is the same as using the selectivity function.

Suppose, in the example above, that dummy objects having as natural-values non-atomic expressions raised to an integral power are the only ones required. Then selection of the alternative:

(D6) \( \frac{d}{dx}(u^n) = u \cdot (n-1) \frac{d}{dx}(u) \)

alone does not satisfy all the conditions. However, the selectivity function: "do not create temporary attributes corresponding to the differentiation of an atomic expression raised to an integral power", in conjunction with selecting the rule of differentiation labelled, (D6), together satisfy the required conditions. This simple selectivity function is evaluated only three times, once for each use of the alternative D6. Its effect is to further reduce the number of attributes created from three to two.

The system cannot distinguish memo functions which include marked alternatives, unless we insert an indicators on their property lists. In order to distinguish such functions, the property indicator SLME is attached to their property lists. This property indicator acts only as a marker and has no value associated with it. Its purpose is clearly shown in the flow-chart fragment of Fig. 6.2.
auto is a function which provides the facility for marking alternatives in a function. When the function auto is encountered in the evaluation of a memo function which has the property SLME, a flag is set which is only cleared when the corresponding attribute is created. If, during the evaluation of a memo function which has the property SLME, the function auto is not encountered, then no temporary attribute is created. If the function auto is encountered during the evaluation of a normal function or a memo function without SLME, it has no effect. The role of auto in evaluation is clearly indicated in the flow-chart fragment of Fig. 6.2.

The flow-chart in Fig. 6.2. covers all the selectivity mechanisms described. The implementation of these facilities caused the size of the interpreter for the expression driven system to be increased by 217 words. This increase includes the 111 words mentioned previously, (Section 6.3.2).

6.4. Function-driven systems

The method used in expression driven systems to select temporary attributes can also be used in function driven systems. Selectivity functions are constructed by the user and are attached to the property list of marked functions under the property indicator SLCT. In the function driven system, it is possible to use the structures of memo functions to provide selectivity in exactly the same way as the structure of marked function is used in the expression driven system.

The flow-chart of Fig. 6.2 shows the modifications to
the interpreter needed to provide the mechanism for both types of selectivity in a function driven system. These modifications increase the size of the interpreter of the system by 163 words. Since the modifications made to the interpreter for the function driven system are almost identical to those made to the interpreter for the expression driven system, (Section 6.3.2), no further explanation is given. Further, since the process of evaluation in the function driven system is the same as that in the expression driven system, no illustrative example is given.

6.5. Repertoire of commands

The only additions to the user's set of commands are the functions select and selector which are common to both the expression and function driven systems.

select (x)

This function is used to establish the link between a function and a selectivity function which is provided by the user. The argument of select is a list of the form:

\[((u_1 v_1)(u_2 v_2) \ldots (u_i v_i) \ldots (u_n v_n))\]

where \(u_i\) is the name of the function and \(v_i\) is the name of its corresponding selectivity function. The effect of evaluating each pair-list, \((u_i v_i)\) is to attach the selectivity function \(v_i\) to the property list of the function \(u_i\) as the value of the property indicator SLCT. An error message is generated if the function \(u_i\) is not a marked or a memo function. No value is returned. Examples of its use appear in Section (E.6).
selector \{ x \} :SUBR pseudo function

The argument of selector, \( x \), is a list of those function names which are to be defined using the function auto in their structure. The effect of executing this command is to attach the property indicator SLME to the property lists of the specified functions. An error message is generated if any of the specified functions is not a marked or memo function. No value is returned. Examples of its use appear in Section (B.6).

6.6. Analysis of practical results

The problems described in Section (2.3) are used to examine the effect that selectivity, as described in Section (6.2), has on the performance of typical evaluations. In the examples below simple properties of the arguments of marked functions are used as criteria for selecting the attributes to be created and stored. Since the performance of the various systems relative to each other has already been established, there is little point in solving each problem in each of the three systems which have been provided with a selectivity feature.

The data in the various tables used in our discussions are taken from the appropriate tables provided in Section (B.5). In each case, the commands used to specify the solution and the selectivity function are listed in Section (B.6).

6.6.1. Problem 1 (differentiation 1: Section 2.3.1)

Burden on the user

The evaluation was carried out using a function driven
system with dynamic attribute-lists, and also using a function
driven system with fixed-size attribute-lists and a free
word list of 7000 cells. The differentiation function is
declared as a memo function and for the system with fixed-
size attribute-lists the system attribute-list was given 100
entries.

Suppose that we are interested in temporary attributes
corresponding to the differentiation with respect to \( x \) of
expressions which are raised to an integral power other than
one. As pointed out in Section (6.2), this form of selectivity
can be arrived at in two different ways. First, the user can
construct a selectivity function which returns the value true
only when its argument is an expression raised to some integral
power other than one, (Section B.6.1), and attach this to the
property list of the differentiation function. Second, the
auto mechanism can be used in the rule of differentiation which
deals with expressions raised to some integral power other
than one, (Section B.6.1).

The system uses any attributes created in order to assist
the evaluation without requiring any further action by the
user. At any point within the evaluation, however, the user
can intervene to modify his choice of attributes by altering
the definition of the selectivity function or by altering
other parameters of the system.

Storage space

The table in Fig. 6.4 shows the number of garbage
collections required to perform the evaluations using various
<table>
<thead>
<tr>
<th>Systems used</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>72</td>
<td>X</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + selectivity function</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + auto</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FDS: fixed-size attribute-list 100 entries</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>88</td>
</tr>
<tr>
<td>FDS: fixed-size attribute-list + selectivity function</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>19</td>
<td>41</td>
</tr>
<tr>
<td>FDS: fixed-size attribute-list + auto</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

Fig. 6.4. Table contains the number of garbage collections required to perform the differentiation I problem in the systems listed.

<table>
<thead>
<tr>
<th>Systems used</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>47</td>
<td>139</td>
<td>528</td>
<td>X</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>22</td>
<td>37</td>
<td>72</td>
<td>146</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + selectivity function</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>27</td>
<td>49</td>
<td>104</td>
<td>174</td>
</tr>
<tr>
<td>FDS: system attribute-list 100 entries</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>23</td>
<td>40</td>
<td>77</td>
<td>343</td>
</tr>
<tr>
<td>FDS: system attribute-list 100 entries + selectivity function</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>27</td>
<td>49</td>
<td>126</td>
<td>217</td>
</tr>
<tr>
<td>FDS: system attribute-list 100 entries + auto</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>22</td>
<td>40</td>
<td>250</td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + auto</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>22</td>
<td>35</td>
<td>79</td>
</tr>
</tbody>
</table>

Fig. 6.5. Table contains the time needed to perform the differentiation I problem in the systems listed.
systems. This table shows that selectivity, based on the use of the structure of the differentiation function, reduces the number of garbage collections required from the fourth order of differentiation onward. This is because most of the attributes which are not used in the evaluation or have very little effect on the performance have been prevented from entering the system attribute-lists. Consequently, the storage space which would have been otherwise occupied by these attributes is available for other uses, which in turn reduces the number of garbage collections required.

Using the alternative method, (a separate selectivity function), to obtain the same choice of attributes increases the number of garbage collections required from the fourth order of differentiation onward in the function driven system. The increase in the number of garbage collections is due to the evaluation of the selectivity function. For example, the additional garbage collections required to perform the sixth order of differentiation arise from the fact that the selectivity function provided is evaluated at least 1250 times (see table of Section B.5.1.).

In the function driven system with fixed-size attribute-lists containing 100 entries, a significant change occurs in the evaluation of the seventh order differentiation - the number of garbage collections decreases compared to the system without selectivity. This is because the selectivity function prevents trivial attributes from entering the system attribute-list, which in turn prevents the deletion of significant attributes. Notice that it was these trivial attributes, combined with the inadequate number of entries in the system.
attribute-list, which gave rise to a sharp increase in the number of garbage collections in the function driven system without the selectivity feature, (Section 4.6.1).

Time of evaluation

The table in Fig. 6.5 shows the time required to carry out the evaluations under various systems. The table shows that selectivity which is based on the structure of the differentiation function reduces the evaluation time as the higher orders of differentiation are reached. Some of the reasons for this reduction are:

(i) look-up time is reduced since the system attribute-lists have fewer entries,

(ii) the reduction in the number of garbage collections,

(iii) the rate of deletion of useful attributes is reduced.

In the function driven system with dynamic attribute-lists, this is because the number of garbage collections which govern the life-time of attributes is reduced.

In the function driven system with fixed-size attribute-lists, this is because fewer attributes are entering the system attribute-lists.

The use of alternative methods to select the same attributes normally increases the evaluation time. This is because the selectivity function has to be calculated for every call to the differentiation function. The only exception to this is in the case of the seventh order differentiation using the function driven system with fixed-size attribute-lists. The reasons given above in the storage
space section also explain this behaviour.

6.6.2. Problem 2 (differentiation 2: Section 2.3.2)

Burden on the user

This problem was solved using an expression driven system with dynamic attribute-lists and incorporating selectivity. The criteria used to select attributes for three different evaluations are listed below.

(a) A selectivity function is constructed and attached to the property list of the differentiation function, and only allows the creation of attributes corresponding to the product of two or more terms.

(b) auto is used to modify the rule in differentiation which handles product terms in order to arrive at the same choice of attributes as in (a).

(c) Attributes are selected according to the number of atomic symbols in their arguments.

In comparison to the host system, the only extra burden imposed on the user is the choice of the selectivity criteria, the construction of the necessary selectivity functions and the writing of the commands which activate selectivity during the evaluation, (Section B.6.2).

Storage space

This problem was solved using a free word list of 7000 cells. The table:
shows how the two different methods used to create attributes corresponding to the product of two or more terms affects the number of garbage collections required. Due to the reasons given in Section (6.6.1), the number of garbage collections in the approach which does not make use of structure of the differentiation function for selectivity purposes is higher than the alternative method. With both types of selectivity method used, the number of garbage collection cycles invoked is larger than when the problem was solved in the expression driven system without the selectivity feature. One of the reasons could be that the selectivity function prevents some of the significant attributes from entering the system attribute-list.

The curve of Fig. 6.6 shows how the imposition of a limit on the number of atomic symbols in the argument components of attributes affects the number of garbage collections. The curve shows that as the number of atomic symbols in the argument component of attributes increases, the number of garbage collections required to solve the problem decreases. This is because the amount of evaluation saved is related to the complexity of the expression being differentiated.

<table>
<thead>
<tr>
<th>host system</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDS: dynamic attribute-list</td>
<td>5</td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + selectivity function</td>
<td>9</td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + auto</td>
<td>7</td>
</tr>
</tbody>
</table>

The table:
Fig. 6.6. Curve shows how the number of garbage collections varies with respect to the number of atomic symbols in the argument component of attributes for the differentiation 2 problem.
<table>
<thead>
<tr>
<th>host system</th>
<th>417</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDS: dynamic attribute-list</td>
<td>71</td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + selectivity function</td>
<td>125</td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + auto</td>
<td>97</td>
</tr>
</tbody>
</table>

shows the evaluation time for the systems and the methods of selectivity used to solve the problem. In comparison with the expression driven system without the selectivity function, the use of both types of selectivity has increased the time of evaluation. Apart from the reasons given in the storage space section which account for this increase, in one of the methods the selectivity function provided has to be evaluated 392 times.

The curve of Fig. 6.7 shows how the number of atomic symbols in the argument components of attributes affects the time of evaluation. For the reasons given above, the evaluation time decreases as the number of atomic symbols in the argument component of attributes increases.

6.6.3. Problem 3 (prime numbers: Section 2.3.)

Burden on the user

The evaluation was carried out using the function driven system with a fixed-size attribute-list of 20 entries and with a function driven system with dynamic attribute-lists. All evaluations used a free word list of 7000 cells. It was pointed out in Section (3.6.4), that if the user explicitly creates the appropriate permanent attributes, then greatly improved performance is achieved. With the use of selectivity functions, however, the same choice of attributes can be
Fig. 6.7. The curve shows how the evaluation time varies with respect to the number of atomic symbols in the argument component of attributes for the differentiation 2 problem.
obtained by constructing the appropriate selectivity function and attaching it to the property list of \texttt{prs}, (section 3.6.3). Consequently, only temporary attributes corresponding to the argument of \texttt{prs} which are divisible by 5 are created.

\textbf{Storage space}

The table in Fig. 6.8 shows the number of garbage collections and time of evaluation required to carry out the evaluation with the recursive and iterative definitions of \texttt{prs} using the systems and conditions indicated. In comparison with the systems with permanent attributes, the use of the selectivity feature increases the number of garbage collections required for both the recursive and the iterative definitions of \texttt{prs} by a small amount (3 and 7 respectively). Two of the reasons for this increase are as follows. First, in the systems with permanent attributes, the permanent attributes were made available before the initiation of the evaluation cycle. In the present situation, however, the system must create the attributes when it first encounters those arguments which satisfy the requirements of the selectivity function. This reduces the number of times that the attributes are obtained from the attribute-lists by 20 for both types of definition of \texttt{prs}. Second, on every call to the function \texttt{prs} the selectivity function is evaluated.

A significant change occurs in connection with the function driven system with fixed-size attribute-lists. By utilizing the criteria provided by the user to make an intelligent choice of 20 temporary attributes, the system performs the evaluation, (using the recursive definition of
<table>
<thead>
<tr>
<th>Systems used</th>
<th>garbage collections</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>host system: recursive definition</td>
<td>130</td>
<td>3620</td>
</tr>
<tr>
<td>host system: iterative definition</td>
<td>204</td>
<td>3429</td>
</tr>
<tr>
<td>FDS: 20 permanent attributes: recursive definition</td>
<td>6</td>
<td>178</td>
</tr>
<tr>
<td>FDS: 20 permanent attributes: iterative definition</td>
<td>104</td>
<td>2896</td>
</tr>
<tr>
<td>FDS: fixed-size (100) and dynamic system attribute-lists: recursive definition</td>
<td>0</td>
<td>84</td>
</tr>
<tr>
<td>FDS: fixed-size (100) and dynamic system attribute-lists: iterative definition</td>
<td>52</td>
<td>605</td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list of 40 entries: recursive definition</td>
<td>190</td>
<td>4852</td>
</tr>
<tr>
<td>FDS: fixed-size (100) and dynamic attribute-lists + selectivity function: recursive definition</td>
<td>9</td>
<td>197</td>
</tr>
<tr>
<td>FDS: fixed-size (100) and dynamic attribute-lists + selectivity function: iterative definition</td>
<td>111</td>
<td>2939</td>
</tr>
</tbody>
</table>

Fig. 6.8. Table compares the number of garbage collections and evaluation time required to evaluate prs when a separate selectivity function is associated with it.
prs), by invoking the garbage collection routine only 9 times. As the table of Fig. 6.8 shows, allowing all the attributes to enter the fixed-size system attribute-list of 40 entries indiscriminately causes the garbage collection routine to be invoked 190 times.

**Time of evaluation**

The time required to carry out the evaluation in the various systems is provided in the table of Fig. 6.8. Once again the use of the selectivity function to arrive at a specific choice of temporary attributes has increased the evaluation time for both the recursive and the iterative definitions of prs. Apart from the reasons given in the storage space section which account for this increase, an additional factor is the increase in the number of garbage collection cycles invoked and the time required to construct the attributes.

**6.7. Conclusion**

Whether selectivity can or should be used is a characteristic of the problem. The effects that its use has on the performance of an evaluation depends on how efficiently properties derived from the functions, their arguments, and their corresponding values can be used to select useful temporary attributes.

In the function driven system, if the fixed-size system attribute-lists have an inadequate number of entries, both methods of implementing selectivity lead to some degree of improvement in performance.
of functions to provide the selectivity feature gives a greater improvement in performance.

Using a separate selectivity function which has to be evaluated at every call to the function with which it is associated gives a performance significantly better than that of the host system, but worse than the best performance achieved by other systems. This is largely due to the time and space used in the evaluation of the selectivity function.

In the systems described so far, a derived value can be obtained from an attribute-list if the appropriate argument value pair appears explicitly on the attribute-list. In the next chapter we examine a method by which attributes can be "classified", and show how this concept leads to implicit use of the entries of attribute-lists.
7.1. Introduction

The introduction of a method of classifying attributes is an attempt to make it possible to use attributes already available to derive the values of similar attributes. The subsequent sections describe the concepts underlying classification, explain how they have been implemented, and outline the effects that these features have on an evaluation. Next, the additional commands provided for the user are described. Finally, a number of problems are used to investigate the effects of the use of classification on the performance of the system. The practical results obtained indicate that the use of the classification feature in an appropriate problem can lead to a substantial saving in the evaluation time and the number of garbage collections invoked.

7.2. Basic Concepts

The idea of classifying attributes can be best explained with the help of a simple example. Consider the differentiations

\[
\frac{d}{dx}(ax + b) = a; \quad \frac{d}{dx}(3x + 2) = 3; \quad \frac{d}{dx}(qz + p) = q
\]

where a, b, p and q are constants. In the systems described so far, the attributes corresponding to each of these evaluations would be preserved separately on an appropriate attribute-list. Subsequently, if the values of:
are required, the appropriate attribute-list is searched and, after a successful match of the argument, the required value is retrieved. Up to this point, therefore, we have been dealing with the explicit use of the entries in attribute-lists.

A user, on the other hand, would notice at a glance that the expressions:

\[ ax + b; \quad 3x + 2; \quad qz + p \]

are all linear with respect to their independent variables \((x\text{ and }z)\), and knowing that

\[ \frac{d}{dx}(ax + b) = a \]

he would proceed to calculate the values of the other two differentiations as follows:

\[ \frac{d}{dx}(3x + 2) = \frac{d}{dx}(ax + b) \text{ with } a = 3; \quad x = x; \quad b = 2 \]
\[ = a = 3 \]

\[ \frac{d}{dx}(qz + p) = \frac{d}{dx}(ax + b) \text{ with } a = q; \quad x = z; \quad b = p \]
\[ = a = q \]

In addition, he would recognize, for example, that the differentiation with respect to \(x\) of all the following expressions:

\[(xa + b); \quad (b + ax); \quad (b + xa); \quad (xa); \quad (ax)\]

produces the same result. If a system is designed so that it is capable of making use of the results of past evaluations to perform a similar, but not identical, current evaluation, then entries in the attribute-lists can be used indirectly to
A number of points come out of the example:

(a) by inspection, the user grouped a number of expressions into a class identified by a property common to all the expressions (linearity in the example),

(b) the user recognised that the result of a past evaluation could facilitate the performance of similar evaluations,

(c) performing similar evaluations involved matching the corresponding constituents of the two expressions and then substituting the corresponding values in the known result.

The aim is to construct a programming system in which the user is able to classify expressions and the system can use classes of expressions to aid the performance of evaluations. The system will perform the first task listed above, the recognition of related expressions, using a function that we shall refer to as a "classification function". Classification functions are provided by the user and their performance will depend on the user's skill and ingenuity. A classification function which determines the class of an expression must also return a standard list of values which "define" the expression relative to the classification function and relative to a known "standard form". For example, the classification function \texttt{linear} applied to the expression \((b + ax)\) could be written to produce the list \((x a b)\) as value of

\[
\text{linear } (x; (b + ax)) = (x a b)
\]
The classification function must always return a value list, where the corresponding parameters occupy the same relative positions. The order of parameters in the value list is of significance because they are used in subsequent evaluations, (Section 7.3.3). As a result, a user deciding to classify an expression by inspection must provide an appropriate value list that takes account of the significance of the order of the parameters in the value list.

It is the user who must decide on the standard form for a class of expressions. For instance, \((ax + b)\) can serve as the standard form for expressions of the class *linear*, and \((ax^2 + bx + c)\) can serve as a standard form for expressions of the class *quadratic*. The user can also provide those attributes of the class expression which are likely to be useful in future evaluations. In brief, therefore, before the solution of a problem is initiated, the user must have provided the system with:

(a) one or more classification functions,

(b) the standard form of expressions for each of the classes,

(c) the value list obtained by applying the classification function to the standard form of expressions,

(d) attributes stated in terms of the standard form of expressions.

Subsequently, once an expression is identified as a member of a class, it can use any item of information belonging to its class, (Section 7.3.3).
The user need not provide classification functions if he can decide for himself the class of all expressions appearing in the evaluation. However, he must still provide the system with all the other items of information listed above.

We will see that another possibility may arise. The user may provide classification functions and decide that they are to be applied only to expressions that he designates explicitly to the system. If the expression belongs to a class, the list which the classification function produces as its value is used in conjunction with the attributes in the corresponding class in order to assist further evaluation.

We notice that when the user classifies expressions by inspection or designates the expressions which are to be classified to the system, he does this once only. Consequently, the system constructs and preserves all the communication links needed so that members of classes can make use of the information provided within their classes during the life-time of an evaluation.

If the system is designed to automatically decide on classes for expressions, the following possibilities can arise. First, an expression is classified each time it is encountered. Once the current evaluation is complete, the system discards both the class of the expression and the corresponding value list. Second, the system stores all the information which it has already obtained as a result of completing the current evaluation, including the classification of expressions, and uses this information in subsequent
evaluations. The second approach is to be preferred especially if either the expressions are encountered a large number of times, or classification functions are complex and costly to evaluate.

The use of a classification feature has a marked effect on the storage space. For example, if there are $m$ expressions which fall into one class, and associated with each argument expression there are $n$ attributes, and each attribute occupies an average of $t$ storage units, then $k_2$, the storage space required to house the attributes in an expression driven system without the classification feature, is given by the relation:

$$k_2 = mnt \text{ cells}$$

As pointed out above, by classifying the expressions only one attribute of each kind needs to be preserved so that the other members of the class can make use of them. Hence, if $k_1$ is the storage space required to house the set of attributes for the class then:

$$k_1 = nt \text{ units}$$

Consequently, a crude estimate of the saving in storage space due to the use of the classification feature is given by:

$$k_2 - k_1 = (m - 1)nt \text{ units}$$

Notice that the saving in storage is directly proportional to the number of expressions which are members of the class, the number of attributes that a class has, and the storage space needed to house the attributes of the class.
7.3. Classification in expression driven systems

7.3.1. Basic features

In some of the expression and function driven systems which have been described so far, an attribute is generated if the corresponding value of the memoized element can not be found on its attribute-list. Cases can arise where the argument components of a number of attributes have identical structure, although they differ in their atomic constituents or their constituent elements are re-arranged - i.e.

\[(3x + 2); (b + ax); (xa + b); (2 + x^3)\]

For complex evaluations we cannot expect to maintain a direct correspondence between attributes and the evaluations they represent. This is because:

(a) attributes occupy more and more space so that the storage space available for evaluation is rapidly exhausted and the solution of the problem has to be abandoned,

(b) in systems with either fixed-size or dynamic attribute-lists, the use of large amounts of storage space to store attributes results in rapid destruction of temporary attributes, (Sections 4.4.2 and 5.3.3).

When objects are grouped according to their natural-values, they are said to belong to the class defined by their natural-values. An object in a given class can make use of all attributes of the member objects of the class.

In this section we examine how the user and the system can arrange the indirect use of attribute-lists by making use of a classification feature. As with the expression driven
system with permanent attributes, the manual use of a classification feature places an extra burden on the user, the extent of which depends on such factors as the number of classification functions to be provided and the amount of information which must be stored in each class.

Before describing the system, let us define the terminology used. The terms and their corresponding meanings are listed below.

**standard expression**  Each class has an object which takes a standard expression as its natural-value. For example, the standard expression \((ax + b)\), where \(a\) and \(b\) are constants, serves as the natural-value for the object of class linear.

**class object**  An object possessing a standard expression as its natural-value is referred to as a class object.

**classification function**  A function which can determine whether a given object belongs to a specified class.

**similar object**  Apart from the class object, all other objects in a class are referred to as similar objects. Similar objects are distinguished by the property indicator SINF which appears on their attribute-lists.

**class attributes**  These are the attributes evaluated using the natural-values of the class objects.

**class attribute-list**  This is the attribute-list associated with the class object.

**reduced a-list**  The list returned as the value of the classification function is called a reduced a-list. The
term reduced a-list is used because, in obtaining the derived-values for a similar object, its reduced a-list is paired with the reduced a-list of the class object to form a pair-list which is used to evaluate derived-values. The pair-list is formed and used in evaluation in a similar way to the association list (a-list), in LISP 1.5.

A class can have an unlimited number of objects as members. The idea is to set up the frequently used attributes on the attribute-list of the class object, and to allow other members of the class to make use of them by referencing the class object. In order to illustrate this procedure, let us consider a simple example. First the object LINEAR is created:

\[
\text{LINEAR} = ax + b
\]

Second, the classification function \texttt{linearlaw} is applied to the natural-value of the object \texttt{LINEAR} in order to establish the object \texttt{LINEAR} as a class object.

\[
\text{linearlaw} \{ x; (ax + b) \} = (x a b)
\]

Notice that since the classification function \texttt{linearlaw} returns the reduced a-list \((x a b)\), the object created belongs to the specified class. Third, the class attribute

\[
\frac{d}{dx} (ax + b) = a
\]

is created and added to the attribute-list of the class object \texttt{LINEAR}. Finally, we create the object:

\[
\text{EXP} = (3t + 2)
\]

and classify it as follows:
\begin{align*}
\text{linearlaw } \{t; (3t + 2)\} &= (t \, 3 \, 2) \\
\end{align*}

Since the classification function returns the reduced a-list 
\((t \, 3 \, 2)\) as value, then the object EXP also does belong to the 
specified class and therefore is a similar object.

A symbolic description of the internal representation 
for the class object and the similar object resulting from 
this example is given below.

\begin{description}
\item[class object:]
\begin{align*}
\text{linearlaw} & \quad \{t; (3t + 2)\} \\
\end{align*}
\end{description}
where the meanings of the geometrical symbols used in the above diagrams appear in Section (5.1). We use the above diagrams to draw attention to a number of points.

1. The definition of the classification function and the reduced a-list are attached to the property list of the class object LINEAR under the property indicator CLASS.

2. Through the name of the class object, we have access to the classification function, (linearlaw), and the reduced a-list of the class object.

3. The similar object EXP references the class object LINEAR. The reason for referencing the name of the class object rather than its attribute-list is because this provides access to the property list of the class object as well as its object-value.

4. The value of the property indicator SIM is a list of references to class objects.

5. In comparing these diagrams with those given in Section (3.3.1), it becomes clear that the classification of objects whose natural-values are only a few atomic symbols long saves very little storage, unless the object has a large number of attributes which can be derived indirectly from class attributes.

7.3.2. Attribute house-keeping operations

The user is responsible for the house-keeping of all the attributes and objects that he creates explicitly. He is also responsible for the creation and deletion of class objects,
class attributes and similar objects. Finally, of course, he is responsible for the creation and maintenance of classification functions.

The system, on the other hand, looks after the housekeeping for all dummy objects which it creates and classifies, as well as for the attributes which it generates. The system performs these tasks according to the set of rules which were used in the expression driven system with dynamic attribute-lists, (Section 5.3.3 and 5.3.4).

7.3.3. Operation of the interpreter

The system uses class attributes to assist evaluation. Modifications were made to the attribute-list searching mechanism in order to enable the system to use the class attribute-list. The changes to the interpreter are summarized in the flow-chart in Fig. 7.1. This flow-chart fragment is inserted between the circles labelled A and B in the flow-chart of Fig. 5.1. These changes increased the size of the interpreter of the expression driven system with dynamic attribute-lists by 317 words.

The setting and unsetting of the flag in the flow-chart is used to avoid the occurrence of infinite recursion. This can arise in connection with class objects which reference themselves (see below). The presence of this flag prevents the search routine from accessing the value of the property indicator SIMI more than once within the same search.

Private attributes

Putting all the attributes required by the members of a
Fig. 7.1 Flow-chart shows the modification which had to be made in the interpreter of the expression driven system in order to make use of the attributes of a class object. The circles labelled A and B are the points at which the above fragment is inserted in the flow-chart of Fig. 5.1.
class on the class attribute-list may prolong the search time unnecessarily. To avoid this, we introduce the concept of private attributes. In addition to having access to the attribute-list of the class object, each member of the class can have its own private attributes. The decision as to whether an attribute should be a class attribute or a private attribute is left to the user. This simple mechanism provides an effective means of controlling the size of the class attribute-lists, and thus keeping the access time within a tolerable limit. In searching for an attribute, the search of any private attributes takes precedence over the search of the class attribute-lists. For this reason, the property indicator SIMI is always the last property indicator which appears on the attribute-lists of similar objects.

Constraints

Allowing the members of different classes to reference one another indiscriminately has two disadvantages. First, the search for an attribute is continued until all the accessible entries of all the attribute-lists of class objects and the similar objects referencing each other are examined. This is likely to prolong the search time. Second, the possibility of infinite recursion can not be avoided by the use of a single flag. This is because the elements of the various classes may reference each other in a circular fashion. Constraints which eliminate both of these drawbacks are listed below.

(i) A class object may only reference itself.

(ii) All similar objects in a class may only reference their
class objects.

(iii) Elements of different classes may not reference one another.

Note that this does not exclude elements being in several different classes.

Example 1

A class object and a similar object with the natural-values \((ax + b)\) and \((3t + 2)\) were created in Section (7.3.1). Below, we show how the system makes use of class attributes in order to evaluate the differentiation

\[ \frac{d}{dt}(3t + 2) \]

when it is encountered later in the evaluation.

(a) The private attribute-list of the similar object EXP is searched for the attribute corresponding to \(\frac{d}{dt}(3t + 2)\); this does not exist.

(b) The similar object has the property indicator SIMI on its attribute-list, which takes a list of references to class objects as its value.

(c) The reduced a-lists \((x \ a \ b)\) and \((t \ 3 \ 2)\) are accessed and used to form a list of pair-lists \[((x\ t)(a\ 3)(b\ 2))\] which transforms:

\[ \frac{d}{dt}(3t + 2) \rightarrow \frac{d}{dx}(ax + b) \]

(d) Now the class attribute-list is searched for the attribute corresponding to \(\frac{d}{dx}(ax + b)\); this search yields:

\[ \frac{d}{dx}(ax + b) = a \]
By using the association list created in (c), we obtain
\[
\frac{d}{dt}(3t + 2) = \frac{d}{dx}(ax + b) \text{ with } ((x.t)(a.3)(b.2))
\]
\[= a = 3\]

Example 2

We use this example to show how the creation of one class attribute can assist in the evaluations of:
\[
\frac{\partial}{\partial x}(x^2 + y^2 + z^2)^n; \quad \frac{\partial}{\partial y}(x^2 + y^2 + z^2)^n; \quad \frac{\partial}{\partial z}(x^2 + y^2 + z^2)^n
\]

First, the object QP with the natural-value \((x^2 + y^2 + z^2)^n\) is created:
\[
QP = (x^2 + y^2 + z^2)^n
\]

Second, the object QP is established as the class object in a class defined by inspection with standard form \((x^2 + y^2 + z^2)^n\), and the variable \(x\) as the independent variable, to give a reduced a-list \((x y z)\) for future use. The power \(n\) is not included in the reduced a-list since all the expressions in the class are to be raised to the same power. Third, the object QP is a member of the same class with respect to the variables \(y\) and \(z\) and with the reduced a-lists \((y x z)\) and \((z y x)\) respectively, (expression symmetric in \(x, y\) and \(z\)).

At this point the symbolic description of the internal representation of QP has the form:
where the meanings of the symbols used in the diagram are provided in Section (B.1).

Having created the structure, let us examine its role in the following evaluations:

(a) \( \frac{\partial}{\partial x} (x^2 + y^2 + z^2)^n \)

In this case the value required is obtained directly from the attribute-list of the class object.

(b) \( \frac{\partial}{\partial y} (x^2 + y^2 + z^2)^n \)

The steps in this evaluation are listed below.

1. The value of \( \frac{\partial}{\partial y} (x^2 + y^2 + z^2)^n \) does not exist on the class attribute-list.

2. The first reference in the value list of SIMI indicates that the class object is referencing itself. At this point, the reduced a-lists (y x z)
and \((x, y, z)\) are used to form the list:

\(((y.x)(x.y)(z.z))\)

3. Using the above pair-list, the evaluation is transformed:

\[\frac{\partial}{\partial y}(x^2 + y^2 + z^2)^n + \frac{\partial}{\partial x}(x^2 + y^2 + z^2)^n\]

4. The attribute-list of the class object is searched and the attribute:

\[\frac{\partial}{\partial x}(x^2 + y^2 + z^2)^n = 2nx(x^2 + y^2 + z^2)^{n-1}\]

is retrieved from the attribute-list.

5. The value required is obtained by using the a-list in (2) to give:

\[\frac{\partial}{\partial y}(x^2 + y^2 + z^2)^n = 2ny(x^2 + y^2 + z^2)^{n-1}\]

\[((x.y)(y.x)(z.z)) = 2ny(x^2 + y^2 + z^2)^{n-1}\]

7.4. **Automatic classification**

If expressions are classified either before or during an evaluation, then it may be possible to use the appropriate class attributes to speed up subsequent evaluations. To achieve this, as in Section (7.3), the user must create the necessary classification functions, class objects and class attributes. Notice that he can only classify those expressions known to him. The system by making use of the information supplied by the user, can decide on the class of some of the expressions which arise in an evaluation, and use the class attributes to obtain derived values involving these expressions.

To use class attributes to assist the evaluation of
functions involving expressions which the system discovers are members of a particular class, the system must be able to gain access to information supplied by the user. One way of achieving this is to associate the names of the class objects with the marked functions. For example, if in a differentiation the majority of the expressions being differentiated fall into the two classes of linear and quadratic, with the class objects called LINEAR and QUADRATIC, respectively, then

\[
\text{CLFN} \rightarrow (\text{LINEAR}; \text{QUADRATIC})
\]

is appended to the property list of the differentiation function. Notice that through the names of the class objects LINEAR and QUADRATIC we have access to the classification functions, reduced a-lists and class attribute-lists, (Section 7.3.1).

The system can be designed to classify an expression, and, if possible, to make use of the class attributes to obtain a value every time it is encountered. However, this approach is not as economical in terms of evaluation time because of the repeated application of the classification function. A second approach is to preserve information about the class of an expression once it is classified. For example, the system could create the structure:
in order to record the information that the expression 
\((ax + b)\) belongs to the class LINEAR with respect to \(x\), with 
the resulting reduced a-list of \((x\ a\ b)\). We adopt the 
second approach in spite of the fact that it is not 
economical in terms of storage space because of the many 
similar objects created.

7.4.1. Operation of the interpreter

The search for an attribute proceeds in the same manner 
as described in Section (7.3.3). The additional modifications 
needed so that the expression driven system can classify 
expressions automatically are summarized in the flow-chart of 
Fig. 7.2. This flow-chart fragment is inserted between the 
two circles labelled D and E in the flow-chart of Fig. 5.1.

Example

Having associated the name of the class object LINEAR 
created in Section (7.3.1), with the differentiation function
Enter flow-chart of Fig. 7.1. at point A.

Preserve the information on the classification.

If the classification function exists, inform the user and await his action.

If all the classification functions have been tried, access and apply the next classification function.

If the expression belongs to the class, continue.

Flow-chart shows the modifications which had to be made to the interpreter of the expression driven system with dynamic attribute-lists in order to enable the system to classify expressions automatically.
we proceed to show the manner in which the system utilizes the information available to assist the evaluation of:

\[ \frac{d}{dt}(p + q(2 + t)) \]

where \( p \) and \( q \) are constants. Since differentiation is a marked function, the argument list is examined for a reference to an object. None exists since the above expression has never been defined as a natural-value. Since the property indicator CLFN appears on the property list of the differentiation function, the system can make use of the appropriate class attribute of the class object LINEAR (if it exists) to indirectly obtain the result for the above evaluation, provided the expression belongs to the class linear as specified by the classification function linearlaw. Assuming that the expression satisfies the requirements of the classification function, then we have:

\[ \text{linearlaw } \{t; (p + q(2 + t))\} = (t q (p + 2q)) \]

Henceforth, the process of obtaining the value is the same as that described in Section (7.3.3). That is, the current reduced a-list is paired with the reduced a-list of the class object which is accessible through the object name associated with the differentiation function to obtain

\[ ((x.t)(a.q)(b.(p + 2q)) \]

The pair-list is then used to transform the evaluation of

\[ \frac{d}{dt}(p + q(2 + t)) \text{ to } \frac{d}{dx}(ax + b) \]

Consequently, the class attribute,

\[ \frac{d}{dx}(ax + b) = a \]
is used along with the pair-list to obtain the value required:

\[
\frac{d}{dt}(p + q(2 + t)) = \frac{d}{dx}(ax + b) \quad \text{with} \quad ((x,t)(a,q) \\
(b.(p + 2q))) = d
\]

In section (7.2), it was said that once an expression is classified the outcome of this classification procedure is also preserved within the system for future use. In this case, therefore, the structure created as a result of this classification has the form:

![Diagram](image)

7.5. Classification in the function driven system

The concepts described in Section (7.2) were not implemented in the function driven system with dynamic attribute-lists. Some of the reasons for this decision are
1. Apart from the organisational differences, there is no practical difficulty involved in implementing the concepts and the process of evaluation remains unchanged.

2. A classification feature is likely to be of more benefit for problems where complex attributes are involved. It was pointed out earlier, (Section D.1), that expression driven systems are the appropriate choice for such evaluations.

3. The amount of time required did not justify the implementation of the proposed system and the collection of the necessary practical results.

7.6. Repertoire of commands

This section describes the commands used to identify classes of expressions and to construct class objects and similar objects. These commands are hand coded into the interpreter of the expression driven system with dynamic attribute-lists. Where indicated, their M-expression definitions and examples of their use appear in Appendices B and C respectively.

classes (x)  

This command is used to declare the names of the classes to be used. Its only argument is a list of atomic symbols which are the names of the individual classes. On execution, the property indicator CLASS and its corresponding value, (NIL), are attached to the property list of every class name which appears in the argument list. The function defclass (see below),
replaces the contents of the first and second words of the
data cell by pointers to the definition of the classification
declaration and the reduced a-list of the class object,
respectively. No error message is generated and no value is
returned. Examples of its use appear in Section D.1.

\textbf{defclass} \{x\} :SUER pseudo function

The effect of executing this command is to establish
links between the class object and the definition of the
class, and between the class object and the reduced a-list
corresponding to the natural-value of class object. The
only argument of \textbf{defclass} is a list every element of which
consists of three parts, say \((x, y, z)\), where:

- \textbf{x} stands for the name of the classification function,
- \textbf{y} stands for the expression which is to be classified,
  i.e. the argument list for the classification function,
- \textbf{z} an optional argument.

If the classification function has been properly defined
and the optional argument \textbf{z} has the value \texttt{NIL}, then one of
the following possibilities can arise. If the standard
expression belongs to the designated class, then the
application of the classification function \textbf{x} to the expression \textbf{y}
produces a reduced a-list, (Section D.1). If the standard
expression does not belong to the designated class, the
application of the classification function \textbf{x} to the expression \textbf{y}
does not produce a reduced a-list, and the user is informed
of this fact.

If the optional argument \textbf{z} is not \texttt{NIL} then it must take
the form of a reduced $a$-list. In this case, the system is relieved of the task of applying the classification function, $x$, (which can be NIL if the function is not required) to the expression $y$ in order to obtain the reduced $a$-list. Once the class of the standard form is determined, the definition of the class and the reduced $a$-list are organised as follows

\[
\text{CLASS} \rightarrow \text{pointer to the reduced } a\text{-list} \rightarrow \text{pointer to the classification function}
\]

and this structure is appended to the property list of the class object. No value is returned. Its $M$-expression definition and an example of its use appear in Sections D.1 and (C.4) respectively.

\texttt{memclass \{x\}} \quad :\text{SUBR pseudo function}

The task of this command is to establish the links between a class object and other objects which are members of the same class. The only argument of \texttt{memclass} is a list, each element of which consists of three separate parts, say $(x, y, z)$, where:

$x$ stands for the name of the class object,

$y$ stands for a complete argument list for the classification function which can be accessed through the name of the class object. The natural-value of the object which is to be classified appears as part of $y$. 

- 186 -
z an optional argument.

The optional argument, z, plays the same role as in defclass. If the value of the optional argument z is NIL and if the application of the classification function (which is accessed through the name of the class object) to the expression y produces a reduced a-list as the result, then the structure:

![Diagram](class object; reduced a-list)

is created and appended to the end of the attribute-list of the similar object. Note that SIMI is the only property indicator which appears on attribute-lists and yet is not an attribute indicator. If the object which is to be classified does not belong to the specified class the user is informed. No value is returned. An example of its use and its expression definition appear in Section (D.1) and (C.4) respectively.

```plaintext
(jof (x)) :SUBR pseudo function
```

jof takes one argument, x, which is a list of pair-lists. The first element of each pair-list is the name of a marked function, while its second element is the name of a class object which is to be attached to the property list of the function. An error message is generated if the first element of a pair-list is not a marked function, or if its second
element is not the name of a class object. No value is returned, and examples of its use appear in Section (D.6.4).

7.7. Analysis of practical results

Four examples are used to compare the performance of the host system with that of an expression driven system with dynamic attribute-lists containing the classification feature. Due to the cost of computation, and because the performance of the various systems relative to each other has already been established, there is little point in solving each problem in each of the systems implemented. In the two problems which are presented in this section, (differentiation 2 and 5 x 5 determinants), having been provided with the necessary information to define the classes identified, the system can use this information to classify expressions and to obtain their corresponding values by making use of the class attributes. Two additional problems, (one in the area of vector analysis and one in integration), are provided in Sections (D.1) and (D.2). In the case of the problem in vector analysis, the expressions are classified by the user by inspection, while in the integration problem the automatic classification technique is used.

The layout of the results presented in this section is not the same as the layout of the analysis of the practical results in the previous chapters. This is because the material to be presented can not be conveniently broken up into the three subsections used before. In spite of this difference, our discussion still centres around the three main issues which are time of evaluation, use of storage space and burden
7.7.1. Discussion I (differentiation 2; Section 2.3.2)

Using the attribute systems implemented so far, any change in the constituents of the expression:

\[
(a_1 x^2 + b_1 x + c_1)^{n_1} (d_1 + \log(a_2 x^2 + b_2 x + c_2)^{n_2})^{n_3}
\]

\[
(d_2 + \exp(a_3 x^2 - b_3 x - c_3)^{n_3})^{n_4})^{n_5}
\]
gives rise to a number of new temporary attributes. In a third order differentiation one of the temporary attributes which is to be preserved corresponds to the above expression, thus requiring 4369 cells for its preservation. Ignoring the storage space required by the definitions, the working space and the other temporary attributes, a free word list with a maximum of 10,000 cells can only accommodate at most two such attributes. Admittedly, the presence of such complex attributes saves a considerable amount of evaluation time once their subsequent application is reached. However, at the same time they may worsen the performance of the systems with dynamic attribute-lists because such a large reduction in the storage space available causes frequent garbage collections and, as a result, faster deletion of temporary attributes. The use of a classification feature will resolve some of the difficulties described, provided that there are enough expressions of the same type in an evaluation to justify the burden that the use of the classification feature imposes on the user.

The use of a classification feature presents the user with additional work because, first, he must carefully
examine the problem and decide on the classes of expressions which are in use. Then he must construct efficient classification functions so that they can be used by the system during evaluation to classify expressions automatically. The user must also create class objects and class attributes, and associate the names of the appropriate class objects with those functions whose arguments, (or the constituents of their arguments), are to be classified. In this particular case, the commands to enable the system to use the classification feature automatically are listed in Section (B.6.4).

The ability of the system to recognize the class of an expression depends on the capability of the classification function provided. For example, in this particular case a pattern matching function of type RULE, (Section 2.2.2), is used to classify the expressions (Section B.6.4). By taking advantage of transformations and predicates which can be included in an assertion, (Section 2.2.2), the system can recognize a wide range of expressions belonging to the same class. For example, the definition of the classification function provided in Section (B.6.4) can recognize the expression:

\[ \{(\log(3t^2 + 4 + 5t)^3 + 2)^4 * (2 + 7x + x^2)^7 * \exp(11x^2 + 3 + 12x)^9)^6\}^4 \]

as a member of the class specified.

An evaluation consisting of 100 different expressions of the type given above was performed. After providing the classification information described in Section (B.6.4), the computation was initiated. The computation resulted in:
garbage collections  30  
evaluation time in seconds  968

for the expression driven system with dynamic attribute-lists. It was roughly estimated that the same evaluation would require:

garbage collections  4200  
time of evaluation in seconds  39700

in the host system. The above estimates were obtained by computing one of the evaluations in the host system and multiplying the result by a factor of 100.

As indicated above, the use of the classification feature has resulted in an impressive saving in the evaluation time and the number of garbage collections required. In this particular case the improvement in the performance seems to justify the extra burden involved.

It is interesting to note that the average time taken to classify an expression is 2.51 seconds, and the number of cells required to record the information on the class of an expression is 92. The user has two choices; either to save time by preserving the classification information, or to save storage space by determining the class of the expression every time. Such a decision can only be made in the light of the resources available. In this particular case, it would have taken 9200 cells to preserve the information regarding the classes of 100 expressions.

7.7.2. Discussion 2 (5 x 5 determinant: Section E.13)

Determinants and matrices are just groups of symbols. All that is needed in order to decide on the class of a group of symbols, say,
The computation of 100 different 5 x 5 determinants resulted in:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage collections</td>
<td>8</td>
</tr>
<tr>
<td>Time of evaluation (sec)</td>
<td>232</td>
</tr>
</tbody>
</table>

in the expression driven system with the classification feature.
It was estimated that the same evaluation would have required:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>garbage collections</td>
<td>500</td>
</tr>
<tr>
<td>time of evaluation in seconds</td>
<td>7200</td>
</tr>
</tbody>
</table>

in the host system. The above estimates were obtained by computing one of the evaluations in the host system and multiplying the result by a factor of 100. It is reasonable to assume that as the number of elements in a group and the number of class attributes in the class increases, so does the saving in the evaluation time and the number of garbage collections invoked.

In this instance, the storage space needed to house the attribute corresponding to one of the 5 x 5 determinants is 785 cells. Also, to preserve the classification information of a member of a class requires 63 cells, and an average time of classification is estimated to be 0.93 seconds.

7.8. Conclusion

Techniques have been developed so that the user or the system or both can classify expressions. The user must define the classification functions to be used, the class objects, and the class attributes, and look after their house-keeping. As a result, the burden which the use of the classification feature imposes on the user depends on the effort involved in examining the problem to see whether it is possible to use the classification feature, and in determining the information which has to be provided in order to establish each class.

As in the case of selectivity, (Section 6.2), classification
is a characteristic of a problem. Consequently its effects differ among problems and a general conclusion cannot be reached. In the context of the problems attempted, however, the use of the classification feature has lead to an impressive saving in the evaluation time and the number of garbage collections required. The extent of the reduction in the number of garbage collections required is an indication of efficient use of storage space. Similarly, the extent of the saving in the evaluation time is an indication of the amount of evaluation saved through the indirect use of the attributes available.
8.1. Overview and Conclusion

The objective of this thesis was to investigate the possibility of constructing a programming system in which the performance of function evaluations can be improved by re-using the results of past evaluations. The approach to the problem is based on the fact that a function is just a rule of correspondence mapping elements of an argument set to the corresponding elements of a value set, so that a function may be defined by specifying the rule of correspondence. This view leads to the concept of an attribute, that is, a property of a function expressed as a relation between the function, its arguments and the corresponding function values.

Function evaluation has been investigated in the context of symbol manipulation. The programming systems developed were implemented as extensions to a symbol manipulation system developed by F.V. McBride, {4}.

The effects on specific function evaluations of the provision of facilities for creating and using attributes were examined by noting how the systems provided affected the user's programming effort, the time to carry out an evaluation, and the storage space used in the course of the evaluation.

The use of a system with facilities for creating and using attributes imposes on the user the need to:
(a) understand the evaluation being performed in order to decide how to use and control attributes,
(b) learn the programming language in order to define the evaluation and to specify how it is to be performed,
(c) perform the house-keeping operations during an evaluation.

Below, we provide a brief summary of those features implemented and their effects on the performance of the evaluations attempted. This is followed by our conclusion which is based on this summary, and distinguishes those features whose inclusion in a symbol manipulation system is likely to improve the performance of function evaluations when appropriately used.

8.1.1. Systems with permanent attributes

The systems with permanent attributes were the simplest systems implemented. The advantage of these systems is that they allow the user to modify function definitions when the actual arguments of the function are known. By creating permanent attributes corresponding to commonly used function values, the user can greatly reduce the number of re-evaluations.

If he understands the host system, then the user only has to learn 14 new commands in order to be able to use a system with permanent attributes. The user must explicitly specify those elements which are to have attribute-lists, define the attributes, and perform the house-keeping operations involved during an evaluation. The burden involved in these tasks is
related to the number and the complexity of the permanent attributes involved.

The use of permanent attributes normally reduces the number of garbage collections required during an evaluation. Reduction in the number of garbage collections is an indication of a more efficient use of the storage space, and leads to a substantial saving in the time of evaluation. Typical factors by which the use of the permanent attributes reduced the number of garbage collections are provided in Table 8.1. Clearly the size of these factors depends on the choice of permanent attributes and no attempt was made to select the most useful attributes in each case.

Systems with permanent attributes perform the same evaluations faster than the host system. The increase in the speed of evaluation depends on the choice of permanent attributes, since each attribute saves a specified amount of direct evaluation. Typical factors by which the use of attribute systems reduced the evaluation time are shown in Table 8.2. It can be seen from this table that function-driven systems do not perform as efficiently as expression driven systems. This is due to the extra storage space required by the attribute-lists and the extra time used to look up attributes in the function driven system.

The use of permanent attributes increases the manipulative power of the system. For example, the differentiation problem which could not be solved in the host system with a free word list of 4000 free cells, (because of the lack of work space), can now be solved using an
### Fig. 8.1. Table contains the factors by which the use of the systems indicated have reduced the number of garbage collections required to carry out the same manipulations in the host system.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Systems</th>
<th>( \frac{d^3}{dx^3} (a+x-b)^m )</th>
<th>( \frac{d^6}{dx^6} (a+x-b)^n )</th>
<th>Differentiation</th>
<th>Prs: recursive definition</th>
<th>Prs: iterative definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDS: permanent attributes</td>
<td>I</td>
<td>2.1</td>
<td>3</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: permanent attributes</td>
<td>I</td>
<td>1.4</td>
<td>3</td>
<td>34</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system</td>
<td>I</td>
<td>18</td>
<td>6.2</td>
<td>204</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>attribute-list of 100 entries</td>
<td>I</td>
<td>18</td>
<td>8</td>
<td>204</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-list</td>
<td>I</td>
<td>24</td>
<td>9.2</td>
<td>204</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>FDS: dynamic attribute-list</td>
<td>I</td>
<td>18</td>
<td>8</td>
<td>204</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + auto</td>
<td>X</td>
<td>X</td>
<td>6.5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system</td>
<td>I</td>
<td>36</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>attribute-list + auto</td>
<td>I</td>
<td>36</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + selectivity function</td>
<td>X</td>
<td>X</td>
<td>5.1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FDS: fixed-size system</td>
<td>I</td>
<td>3.8</td>
<td>X</td>
<td>22.7</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>attribute-list + selectivity function</td>
<td>I</td>
<td>4.2</td>
<td>X</td>
<td>22.7</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Problems</td>
<td>System</td>
<td>[\frac{d^n}{dx^n}(a+x-b)^m]</td>
<td>[\frac{d^n}{dx^n}(a+x-b)^m]</td>
<td>[\frac{d^n}{dx^n}(a+x-b)^m]</td>
<td>[\frac{d^n}{dx^n}(a+x-b)^m]</td>
<td>[\frac{d^n}{dx^n}(a+x-b)^m]</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>EDS: permanent attributes</td>
<td>1.1</td>
<td>1.3</td>
<td>2.5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: permanent attributes</td>
<td>1.1</td>
<td>1.2</td>
<td>2.4</td>
<td>20</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list of 100 entries</td>
<td>1.8</td>
<td>6.9</td>
<td>4</td>
<td>43</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-list</td>
<td>1.8</td>
<td>7.9</td>
<td>6.4</td>
<td>43</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>FDS: dynamic attribute-list</td>
<td>1.8</td>
<td>7.3</td>
<td>5.8</td>
<td>43</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + auto</td>
<td>X</td>
<td>X</td>
<td>4.3</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list + auto</td>
<td>2.3</td>
<td>13.2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + auto</td>
<td>2.3</td>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EDS: dynamic attribute-list + selectivity function</td>
<td>X</td>
<td>X</td>
<td>3.3</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDS: fixed-size system attribute-list + selectivity function</td>
<td>1.8</td>
<td>15</td>
<td>X</td>
<td>19.3</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>FDS: dynamic attribute-list + selectivity function</td>
<td>1.8</td>
<td>5</td>
<td>X</td>
<td>19.3</td>
<td>1.1</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8.2. Table contains the factors by which the use of the systems indicated have reduced the time of evaluation required to carry out the same manipulations in the host system.
attribute system with a free word list of only 3000 free cells.

Systems with permanent attributes are suitable for evaluations where the use of very few attributes removes many re-evaluations. They suffer from two main drawbacks. First, the user can only specify attributes corresponding to evaluations known to him. Second, the burden that the use of the systems imposes on the user is substantial if a large number of attributes are involved.

8.1.2. Function driven system with fixed-size attribute-lists

In this system both the user and the system can create attributes corresponding to commonly required function evaluations. System generation of attributes has the advantage that attributes corresponding to common function evaluations which arise in the intermediate stages of an evaluation can be created and used. The number of attributes created by the system is controlled by fixing the maximum size of the system attribute-lists.

System generation of attributes reduces the burden on the user because now he need only create either those attributes which the system cannot create, or those attributes which improve system performance if they are placed on the user attribute-list. If he understands the host system, then the user must learn 9 additional commands to use this system. The user designates to the system functions which are to have attributes and specifies the maximum number of entries in each of their attribute-lists.
Typical factors by which the use of a function driven system with fixed-size system attribute-lists reduces the time of evaluation appear in Table 8.2. In most cases there is a substantial saving in the evaluation time arising from the large number of temporary attributes which are created and used in the intermediate stages of the evaluation. Two factors contribute towards this saving in the evaluation time—fewer garbage collections and a reduction in the number of direct evaluations due to the presence of certain attributes.

This system performs efficiently if the system attribute-lists have an appropriate number of entries. A bad choice for the sizes of the attribute-lists has the following consequences. With too few entries in the system attribute-lists, the system is continuously creating and deleting attributes without getting a chance to use them, (Section 4.7.1.). With too many entries in the system attribute-lists the look-up time of attributes is greater and the storage space available may be exhausted, thereby bringing the solution of a problem to a halt.

The presence of system-generated attributes increases the manipulative power of the system. For example, the differentiation 2 problem, (Section 2.3.2), which could not be performed by the systems with permanent attributes using a free word list of 2000 cells, can now be successfully performed, and this system can perform the evaluation:

\[ \frac{d^7}{dx^7}(a + (x - b)^n)^m \]

which could not be carried out by the systems with permanent attributes.
The drawbacks of this system are as follows. First, since the size of the system attribute-list remains fixed regardless of the use being made of it, unless the user intervenes to alter the size space may be wasted storing useless attributes. Second, the attributes of functions which are not being used for comparatively long periods of time occupy storage space which could have been used for other purposes. Third, determining the appropriate limit on the size of system attribute-lists is difficult.

8.1.3. Systems with dynamic attribute-lists

The replacement of fixed-size attribute-lists with dynamic attribute-lists leads to a more efficient use of storage space and relieves the user of the task of determining the maximum number of entries in the attribute-lists. In these systems, system generated attributes are maintained only as long as they are used often enough. This method of determining the life-time of an attribute causes the attribute-lists of functions in use to grow and the attribute-lists of functions that are inactive to shrink as an evaluation proceeds.

In comparison with other systems which have been dealt with so far, the use of these systems places the minimum burden on the user. In the function driven system the user need only identify the memo functions. In the expression driven systems, the following possibilities can arise. First, if the choice of memo expressions is left to the system no extra burden is imposed on the user, and all arguments of marked functions, (irrespective of their significance), are
treated as memo expressions. Second, the user chooses to identify the memo expressions himself. Third, the user and the system co-operate in identifying the memo expressions. In the last two instances, therefore, the extra burden on the user is related to the number of expressions that he chooses to identify himself. Apart from a knowledge of the host system, the user must learn 20 additional commands.

Provided the fixed-size attribute-lists have an appropriate number of entries, the systems with dynamic attribute-lists have approximately the same performance for simple evaluations. However, as the complexity of a problem grows, a system with dynamic attribute-lists performs evaluations more efficiently. This is because complex attributes are stored but not used by the system with fixed-size attribute-lists, thus increasing the look-up time and the number of garbage collections.

The performances of the expression and the function driven systems with dynamic attribute-lists differ for the reasons given in Section (8.1.1). The factors in the tables 8.1 and 8.2 show that the use of an expression driven system with dynamic attribute-lists is to be preferred in problems which are rich in common sub-expressions, (Section D.1).

The increase in the manipulative power of these systems is revealed by the evaluations

$$\frac{d^8}{dx^8}(a + (x - b)^n)^m \quad \text{and} \quad \frac{d^9}{dx^9}(a + (x - b)^n)^m$$

whose solution was not possible in any of the previous systems, (see tables of Section E.5.1). One of the reasons for this
increase in the manipulative power of the systems is that the storage space released by the temporary attributes deleted from the attribute-lists provides free storage space for the solution of the problems to continue.

It was found that a poor choice of decrement controlling the life-time of an attribute can lead to poor performance, but variations in the values of other life-time parameters does not affect performance significantly (Sections 5.7 and D.6).

8.1.4. Selectivity

In systems with permanent attributes, the user, by making use of his knowledge of the problem and the role which permanent attributes play in the various stages of the evaluation, can decide when to create (and when to destroy) particular attributes. A selectivity mechanism is designed so that the system using criteria provided by the user can create just those attributes which are significant in an evaluation. The properties which control the selection of attributes are derived from the characteristics of the arguments and the values of functions involved in an evaluation.

The burden that the use of selectivity imposes on the user arises from the detailed knowledge of the problem which is required to use the system. The user must decide on the type of attributes which are to be created and must determine properties of arguments and results which will achieve this. Then he must construct a selectivity mechanism in order to control the creation of attributes.
The results in the tables 8.1 and 8.2 show that only in once instance (Section 6.6.1) has the use of selectivity improved the performance. The improvement occurs because selectivity ensures that there are fewer attributes accommodated on a fixed-size system attribute-list, and consequently most of them are used before they are deleted.

Tables 8.1 and 8.2 show that in all other cases selectivity leads to a worse performance. This is partly because the arbitrary criteria used for the selection have excluded some of the significant attributes from the attribute-list, and partly because of the time used to evaluate the selectivity functions.

Using the structure of a function for selectivity shows a considerable improvement over the alternative method which uses a separate selectivity function because it avoids the time used in the evaluation of the selectivity function.

8.1.5. Classification

Classification is designed to make it possible to use attributes indirectly. This is particularly valuable in applications where argument expressions can be divided into a number of distinct classes so that class attributes can be created and used indirectly during an evaluation. The use of classification substantially reduces the number of attributes which need to be created, and therefore has a marked effect on the storage space used by an evaluation, (Section 7.2).

The user must decide on the classes of expression which are to be used, and determine the properties which can classify
the expressions occurring during the evaluation. In addition, the user must decide on the standard form of expressions for each class and provide a classification function which the system can use to determine the class of an expression.

For the four manipulations attempted, the use of classification substantially reduced the time of evaluation and the number of garbage collections, (Sections 7.7, D.1 and D.2). The number of garbage collections is reduced because the use of class attributes prevents the creation of a large number of attributes. The substantial reduction in the time of evaluation arises from the reduction in garbage collections, and the use of class attributes prevents many direct evaluations.

The use of classification increases the manipulative power of the system. For example, the problems, (Section D.1), which could not be solved using other attribute system due to the shortage of storage space can be solved with the use of classification.

8.1.6. Conclusion

It was stated at the beginning of this thesis that this investigation is not concerned with the best way of implementing the features which have been described in the previous chapters. Rather, the outcome of the practical work undertaken is used to distinguish those features whose incorporation in a symbol manipulation system is likely to improve the efficiency of function evaluations. Such features and their merits are briefly outlined below.
Mechanisms by which the user can identify functions and expressions which are to carry attribute-lists, and mechanisms by which the user can create permanent attributes have the following merits. Their availability enables the user to make use of his knowledge of the problem in order to make an appropriate choice of memoized elements. In addition, he can create attributes to optimize the performance of function evaluation or he can create attributes which the system cannot create. It was observed that most of the burden that the use of an attribute system imposes on the user is related to the use of these mechanisms. The use of these mechanisms alone is likely to satisfy all the requirements of the user in problem areas where there are few memoized elements and few permanent attributes.

Temporary attributes corresponding to those function evaluations which arise in the intermediate stages, and of which the user has no knowledge, can only be created by the system. This requires the provision of mechanisms by which the system can create and control attributes. Of the two methods used to control attributes, (Chapters 4 and 5), the one involving dynamic attribute-lists and a life-time mechanism is to be preferred since it normally leads to more efficient performance.

The provision of a classification feature in a symbol manipulation system depends on the type of manipulations that the system is designed to perform. Its availability allows the user to take advantage of the common properties of functions and expressions, without imposing an additional overhead on evaluations which do not use the mechanism. In this case,
part of the improvement in performance is at the expense of the additional burden on the user of providing the necessary information.

It appears that expressions and their attribute-lists can not be conveniently linked in all programming languages. In such programming languages functions must be used as carriers of attribute-lists. The practical results have also indicated that in the case of complex problems the expression driven systems perform more efficiently than the function driven systems. Hence in programming languages in which expressions as well as functions can conveniently serve as carriers of attribute-lists, the use of expressions is to be preferred.

8.2. Future Developments

8.2.1. A Symbol Manipulation System

Having implemented and examined the effect of various features for improving the efficiency of function evaluation, the next step is to consider the design and construction of an efficient symbol manipulation system incorporating an appropriate range of ideas investigated in this thesis.

If we use our experimental results as a guide then clearly the system should include the following facilities:

(a) a mechanism by which the user can identify functions and expressions which are to carry attribute-lists;

(b) mechanisms by which the user can create and control attributes;

(c) a mechanism by which the system can create and control
attributes, (involving dynamic attribute-lists and a life-time mechanism);
(d) a mechanism for function evaluation which can make use of existing attributes and create new ones.

Since it should be possible to implement a classification mechanism which would allow the user to take advantage of the common properties of functions and expressions without imposing an additional overhead on evaluations which do not use the mechanism, this facility should also be provided.

The current investigation has not been concerned with the best way to implement these facilities, since any satisfactory implementation should take proper account of the structure of the interpreter (or compiler) of the host system. In order to obtain an efficient implementation of an attribute based evaluation mechanism it will be necessary to consider the changes that must be made to the host system in order that attributes can be stored, manipulated and retrieved with a minimum of overhead. For example, the simple binary list structure used in the current implementation would not be a satisfactory way of adding attribute based evaluation to LISP 1.5 as a permanent feature.

8.2.2. Further Investigation

The most obvious area for further investigation is the selectivity mechanisms described in Chapter 6. The work reported in that chapter shows that selectivity mechanisms can be created which enable the user to control the generation of attributes in a wide variety of ways, but that
the cost of such a control mechanism is high in terms of
the time and space used to evaluate the control functions.

The experimental results suggest that further
investigations are required in order to establish:

(a) an efficient approach to selectivity based on using the
structure of the functions involved to trigger the
evaluation of a selectivity function;

(b) a method for constructing selectivity functions which
can be evaluated very rapidly and without using space
from the free word list.

The latter point suggests that selectivity functions should
be constructed from "built-in functions" designed to check
very simple properties, (for example, the size), of the
entities involved.

Any further investigation of selectivity mechanisms
should also consider the construction of additional mechanisms
for controlling the destruction of attributes, since the
mechanism described in Chapter 5 is only concerned with
controlling the creation of attributes. The sensible approach
would be to make use of selectivity functions as well as the
life-time parameters when reviewing attribute-lists during
garbage collection. If the selectivity functions satisfy
point (b) above then the extra time involved in a garbage
collection cycle should not be substantial. Typical criteria
that might be applied to some or all the attribute-lists are;
"delete the attribute containing the fewest elements", "delete
the attribute with the shortest life-time" (this avoids repeated
reviews) and so on. An obvious extension is to link the
creation and deletion of attributes by allowing the user to give an attribute a priority, when it is created, which can be used as an additional criterion when the life-time of that attribute is reviewed, (such a mechanism could be simulated on the present system by automatic adjustment of the life-time parameters).

Another area for further study is the automation of all the various control mechanisms. For example,

(a) providing a mechanism for the automatic adjustment of an attribute life-time according to the properties of the evaluation being performed,

(b) providing a mechanism which tallies the use of functions and named expressions, and automatically associates attributes with those whose tally exceeds some criterion score, (such a mechanism would probably also need to delete null attribute-lists and the corresponding status of the function or expression),

(c) providing a mechanism which tallies the relation between functions and arguments (analogous to the table described in Chapter 5), and which allows the system (possibly with the help of the user), to decide whether a given attribute should be associated with either a function or its argument.

While the provision of such automatic control mechanisms must reduce the burden on the user and will also reduce the amount of direct evaluation performed, they will not necessarily lead to an improvement in a given set of function evaluations because the space and time used by evaluating control functions
may balance or even exceed the savings achieved by the use of attributes.

One area of study which featured significantly in the development of the host system, but which has not been covered directly by the present project, is the "naturalness" of the way in which a user specifies an evaluation. As pointed out in Chapter 1, the use of attribute-lists is natural in the sense that it reflects the mechanism of evaluation used by a man with a pencil and paper. This is particularly true if the system provides a classification mechanism since much of the work we do with pencil and paper depends on recognizing common properties of expressions and functions. Any further investigation of attribute based evaluation should consider the possibility of improving the naturalness of the system by extending the attribute look-up mechanism to include the full range of features offered by the rule "matching mechanism" in the host system. Great care would be needed with this approach in order to ensure that it did not impose a substantial penalty in the form of a large increase in attribute look-up time.

8.3. Epilogue

The author believes that most of the concepts described in this thesis are language independent, although their effects on performance may be different for different languages. The use of an attribute based system improves the performance of function evaluations by bringing the evaluation procedure closer to that adopted by man.
REFERENCES


17. Personal Communication.


The Programming System

This appendix provides further information concerning the host system. It provides the functions of the interpreter, gives illustrative examples showing the working of the matching algorithm and some of the facilities that the host system provides.

A.1 Functions of the interpreter

In this section, the functions evaluate, apply, evalquote, 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.

\[
evaluate \text{[fn;args]} = \begin{cases} 
\text{marked [fn]} \rightarrow \text{evalquote [fn; replace [args; \%LIST]]} ; \\
T \rightarrow \text{evalquote [fn;args]}. 
\end{cases}
\]

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.

\[
evalquote \text{[fn;args]} = \begin{cases} 
\text{get [fn; FEXPR] } \rightarrow \text{get [fn; FSUBR]} \rightarrow \text{eval [cons [fn;args] ; NIL]} ; \\
T \rightarrow \text{apply [fn;args; NIL]}. 
\end{cases}
\]

This exhibits no change from the standard version of evalquote, described in the LISP 1.5 Programmer's Manual, [5].

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.
Fig. A.1 Flow-chart corresponds to the definition of apply given in Section (A.1).
In this description of apply (and in that of eval which follows), 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.

The flow-chart given in Fig. A.1 corresponds to the definition of apply just given. In all the expression and function driven systems with attributes, the changes concerning apply replaces that fragment of the flow-chart which is presented in dashed-line.
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.
A.2 EXAMPLES

In this section two examples are given; the first one does not make use of transformations; the second example compares the extended and ordinary matching system.

A.2.1 Differentiation

Consider the operation of a simple differentiator, which incorporates the following relations:

\[
\frac{dn}{dx} = 0 \quad \text{when } n \text{ is a number}
\]

\[
\frac{dx}{dx} = 1
\]

\[
\frac{d}{dx} (u + v) = \frac{du}{dx} + \frac{dv}{dx}
\]

\[
\frac{d}{dx} (u \star v) = u \star \frac{dv}{dx} + v \star \frac{du}{dx}
\]

\[
\frac{d}{dx} (-u) = - \frac{du}{dx}
\]

Assuming the expression to be differentiated is represented in prefix notation, the above relations can be easily transcribed into a RULE definition \( \tilde{d} \) as follows:

\[
d_1 \quad [n;x] \rightarrow 0 \quad \text{when numberP } [n];
\]

\[
d_2 \quad [x;x] \rightarrow 1;
\]

\[
d_3 \quad [+ [u;v];x] \rightarrow + [\tilde{d} [u;x]; \tilde{d} [v;x]];
\]

\[
d_4 \quad [\star [u;v];x] \rightarrow + [\star [u;\tilde{d} [v;x]]; \varepsilon [v;\tilde{d} [u;x]]];
\]

\[
d_5 \quad [- [u];x] \rightarrow - [\tilde{d} [u;x]]
\]

The rules +, - and \( \star \) must be defined at this stage, however, we give the following dummy definitions for these RULEs:

```
rule +
last [ a;b ] -> list [ + ; a ; b ]
```

```
rule *
last [ a ; b ] -> list [ * ; a ; b ]
```

```
rule -
last [ a ] -> list [ - ; a ]
```

These definitions are used to evaluate the substitutes in assertions of value \( \tilde{d} \).
The S-expression translations of RULEs \(\overline{d}, +, \ast\) and - are represented below:

\[
\begin{align*}
D1 &\quad ((N \times) \ 0 \ ((N \ \text{NUMBERP} \ N))) \\
D2 &\quad ((x \ x) \ 1) \\
D3 &\quad (((+ u v) \ x) \ (+ (D u x) \ (D v x))) \\
D4 &\quad (((\ast u v) \ x) \ (+ (\ast u \ (D v x)) \ (\ast v \ (D u x)))) \\
D5 &\quad (((- u) \ x) \ (- (D u x))) \\
\text{LAST} &\quad ((A \ B) \ (\text{LIST} + A \ B)) \\
\text{LAST} &\quad ((A \ B) \ (\text{LIST} \ast A \ B)) \\
\text{LAST} &\quad ((A) \ (\text{LIST} - A))
\end{align*}
\]

In the following the evaluation of \(D((+ (- (\ast 7 \ z) 3) \ z))\) (that is, \(\frac{d}{dz} (-7z + 3)\)) is demonstrated by presenting the argument expression at each entry to \(\overline{d}\) and the value at each exit. Level numbers in parentheses serve to link corresponding entries and exits to \(\overline{d}\); the label present opposite an argument expression denotes the assertion in \(\overline{d}\) where the expression has been matched.

Arguments (1): \(((+ (- (\ast 7 \ z)) 3) \ z)\) \quad D3

Arguments (2): \(((+ (\ast 7 \ z)) \ z)\) \quad D5

Arguments (3): \(((\ast 7 \ z) \ z)\) \quad D4

Arguments (4): \((z \ z)\) \quad D2

Value (4): \(1\)

Arguments (4): \((7 \ z)\) \quad D1

Value (4): \(0\)

Value (3): \((+ (\ast 7 \ 1) \ (\ast z 0))\)

Value (2): \((- (+ (\ast 7 \ 1) \ (\ast z 0))))\)

Arguments (2): \((3 \ z)\)

Value (2): \(0\)

Value (1): \((+ (- (+ (\ast 7 \ 1) \ (\ast z 0)))) \ 0)\)

In order to get a simplified result, some properties of addition and multiplication should be considered. To do this we add the assertions:

\[\text{apl: } [d;0] \rightarrow d\]

\[\text{to } + \text{ and }\]
Now if $D((+ (- (* 7 z)) 3) z) \text{ is evaluated, the result is } (-7)$.

### A.2.2 Test of linearity

Consider the function linear $[x; e]$ which is to return a true value if $e$ is linear with respect to $x$ and a false value otherwise. (For this illustration the only arithmetic operations involved in $e$ are $+$ and $\ast$). In term of the ordinary matching system, the RULE linear may be written as

\begin{align*}
\text{a1} & \quad [x;x] \rightarrow T; \\
\text{a2} & \quad [x; \ast (a;x)] \rightarrow T \text{ when free } [a;x]; \\
\text{a3} & \quad [x; \ast (x;a)] \rightarrow T \text{ when free } [a;x]; \\
\text{a4} & \quad [x; + (x;b)] \rightarrow T \text{ when free } [b;x]; \\
\text{a5} & \quad [x; + (b;x)] \rightarrow T \text{ when free } [b;x]; \\
\text{a6} & \quad [x; + (\ast (a;x); b)] \rightarrow T \text{ when free } [a;x] \land \text{free } [b;x]; \\
\text{a7} & \quad [x; + (b; \ast (a;x))] \rightarrow T \text{ when free } [a;x] \land \text{free } [b;x]; \\
\text{a8} & \quad [x; + (\ast (x;a); b)] \rightarrow T \text{ when free } [a;x] \land \text{free } [b;x]; \\
\text{a9} & \quad [x; + (b; \ast (x;a))] \rightarrow T \text{ when free } [a;x] \land \text{free } [b;x]; \\
\text{a10} & \quad [x; e] \rightarrow F
\end{align*}

In the extended matching system an equivalent rule may be written as

\begin{align*}
\text{al} & \quad [x; + (\ast (a;x); b)] \rightarrow T \text{ when } [a;x] \land [b;x] \\
\text{al} & \quad \text{with } + [\text{tpl}; \text{tp2}; \text{tp3}] \land [\text{tsl}; \text{ts2}; \text{ts3}]
\end{align*}

provided that the transformation RULE trfs is defined as :

\begin{align*}
\text{tp1:} & \quad + (a;b) \rightarrow + (b;a); \\
\text{tp2:} & \quad a \rightarrow + (a;0); \\
\text{tp3:} & \quad a \rightarrow + (0;a); \\
\text{ts1:} & \quad \ast (a;b) \rightarrow \ast (b;a); \\
\text{ts2:} & \quad a \rightarrow \ast (a;1); \\
\text{ts3:} & \quad a \rightarrow \ast (1;a)
\end{align*}

As an example of the use of the second definition consider the evaluation of LINEAR $(z z)$. The sequence of matching operations is as follows:
attempt to match:

(I) \[ f = (x \,(+\,(\#\,A\,x)\,B))\,;\,\,e = (z\,z) \]

Attempt to match the first elements of (I):

(II) \[ f = x\,;\,\,e = z \]

successful match: \((x\,z)\) appended to a-list.

Attempt to match the second elements of (I):

(III) \[ f = (+\,(\&\,A\,x)\,B)\,;\,\,e = z \]

unsuccessful match: reconstitute e by applying +’s transformations.

tpl is not applicable since \((+\,A\,B)\) does not match z; however, tp2 reconstitutes z as \((+\,z\,0)\).

Attempt to match:

(IV) \[ f = (+\,(\#\,a\,x)\,B)\,;\,\,e = (+\,z\,0)\].

Attempt to match the first elements of (IV):

(V) \[ f = +\,;\,\,e = + \]

successful match.

Attempt to match second elements of (IV):

(VI) \[ f = (\&\,A\,x)\,;\,\,e = z \]

unsuccessful match: reconstitute e by applying \&’s transformations.

The first suitable transformation is ts2 which reconstitutes z as \((\&\,z\,1)\).

Attempt to match:

(VII) \[ f = (\&\,A\,x)\,;\,\,e = (\&\,z\,1) \]

Attempt to match the first element of (VII):

(VIII) \[ f = \&\,;\,\,e = \& \]

successful match.

(IX) \[ f = A\,;\,\,e = z \]

unsuccessful match since the predicate associated with A is false:
reconstitute e in (VI) by applying ts3, the next suitable transformation of \&; hence z becomes \((\&\,1\,z)\).

Attempt to match:

(X) \[ f = (\&\,a\,x)\,;\,\,e = (\&\,1\,z) \]

successful match since the predicate associated with A is true and x
is already bound to z: (A.1) is appended to a-list.

Attempt to match second elements of (IV):

\[(\text{XI}) \quad f = B; \quad e = 0\]

successful match since the predicate associated with B is true:

\[(\text{B.0}) \quad \text{appended to a-list}.
\]

Therefore match (i) is successful and the a-list formed is

\[(\text{((B.0)(A.1)(x,z))}).\]

A.3 The built-in function of the system

In this section the built-in functions of the system are introduced. These functions are divided into six categories as follows:

\[A.3.1 \text{ Defining and editing facilities}\]

\[\text{defrules } [x]\]

where \(x\) is a list of the form

\[((((u_1 \ v_1)(u_2 \ v_2) \ldots (u_n \ v_n))\]

where each \(u\) is a name and the corresponding \(v\) is the S-expression definition for the RULE which is named \(u\).

The function defrules is a pseudo function, that is a function executed for its effect on the system in core memory, as well as its value. The value returned by this function will be the list

\[(u_1 \ u_2 \ u_3 \ldots u_n)\]

which is the list of the names of the newly defined RULEs.

\[\text{addrule } [\text{name; label; newassert} ]\]

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 be added to the RULE. The action of this function is to insert the new assertion into the named RULE immediately in front of the assertion with the indicated label.

If the second argument is NIL, then the new assertion is added after the last existing assertion. No value is returned.

\[\text{delrule } [\text{name; label}]\]

This function deletes the assertion, whose label is given as the second argument, from the RULE whose name is the first argument. No value is
return.

change [name; label; type; new-element] : SUBR pseudo function

This function 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. The type, given by the third argument, must be one of the following 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 nor 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 RULE named by the argument is searched for an assertion whose label is the second argument. If such a label is found the corresponding assertion is output as the value of fetch, 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.

set [name; ind] : SUBR pseudo function

The property list of the atom named by the first argument is searched for the occurrence of the indicator specified by the second argument. If such an indicator exists, then the associated property is output, otherwise the value NIL is returned.

pack [name; list of names] : SUBR pseudo function

As the result of executing this function the list of atoms specified by the second argument is set upon the property list of the atom represented by the first argument under the indicator PACK. The value returned is the list of the first argument.
The second argument is a list of the names of definitions, among which may be the names of other PACKed forms. The PACK indicator is used later in order to simplify operations involving disk-file transfers.

### A.3.2 Variable definitions

Facilities have been introduced for the definition of both local and global variables. Local bindings persist for the duration of the execution cycle in which they are defined whereas global bindings persist throughout a program run or until they are destroyed by user intervention. Global variable definitions can be used on backing-store.

**Global bindings**

\[ \text{ident \{ob; val\}} \] : SUBR pseudo function

The action of this function is to bind its second argument to the first argument which is an atom. The second argument, val, is also returned as a value.

\[ \text{idento \{ob; val\}} \] : SUBR pseudo function

The action of \text{idento} is similar to that of \text{ident} except that the second argument is evaluated before the binding is set up.

**local bindings**

\[ \text{where \{x\}} \] : SUBR pseudo function

Here, x is a list of dotted pairs called a where list of the form:

\[ ((u_1 v_1)(u_2 v_2) \ldots (u_n v_n)) \]

where each u is an atomic identifier and each v can be any S-expression.

Associated with each command there may be a where list. Then, before control passes to the main evaluation routines, with the issued command all occurrences of an atom which has a binding on the where list are replaced by the corresponding binding.

### A.3.3 Interactive facilities

These facilities are provided in order to increase the degree of control which the user may exercise over function evaluation by introducing programmed halts. For example, these facilities allow the user to input substitutes or predicates in the form of S-expressions during function
evaluation, or to suspend the evaluation of a function while any number of
other executions (such as defining or editing) are performed. In addition
facilities are introduced into the interpreter to cope with problems arising
from the absence (in core) of function definitions.

query [ S ]

The action of this function is to output its argument replaced
where appropriate by its a-list bindings. The user is then invited to respond. If he replies by typing YES, then the 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 invi-
tation to respond.

external [ S ]

The action of this function is to output its arguments replaced
where appropriate by their a-list bindings. The user is then invited to reply
and his response will be taken as the value of external.

pause [ S ]

The action of this function is to output its arguments replaced
where appropriate by their a-list bindings. The system then enters "pause",
which can only be terminated by the execution of one of the functions
restart, clear or resume (described in the next section). During the pause
the user may perform any operation he desires, including 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, PAUSE IN RECOVERY PHASE will
be output and this second "pause" made automatically terminated. The original
pause is still valid.

A.3.4 Recovery functions

restart [ ]

This function resets the system to the state that it was in before
entering the recovery mode, and recommences the execution from that point.
Recovery mode is entered either through pause or because an error condition
has been encountered as a result of the absence of a function definition.

**clear [ ]**  :SUBR pseudo function

The action of this function is merely to determine 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.

### A.3.5 Storage management functions

**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.

**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. No value is returned.

**remprop [ x; ind ]**  :SUBR pseudo function

This 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.

**Disk-files**

**get file [name]**  :SUBR pseudo function

`get file` expects its argument to be the name of a disk-file. Its action is to open the indicated file so that the other function described in this section can be executed.
store \([x_1; x_2; \ldots \ldots ; x_n]\) :FSUBR pseudo function

Where each \(x\) is an atomic symbol. This function causes the property list of each of these atomic symbols to be written on to the disk-file. However, if it is found that any of these symbols is already present in the dictionary, the existing definition cannot be overwritten unless the "open" flag is set by the open function (described next). If the open flag is not set, the user is informed, by a message, that his definition has not been stored.

open \([x_1; x_2; \ldots \ldots ; x_n]\) :FSUBR pseudo function

Where each \(x\) is an atomic symbol. Its action is to set the "open" flag against those \(x\)’s present in the dictionary. (The directory contains the names of all the definitions which are stored on the disk-file.)

restore \([x_1; x_2; \ldots \ldots ; x_n]\) :FSUBR pseudo function

Where each \(x\) is an atomic symbol. The definitions associated with these atomic symbols are read from the disk-file (previously opened by getfile) into core.

wipeout \([x_1; x_2; \ldots \ldots ; x_n]\) :FSUBR pseudo function

Where each \(x\) is an atomic symbol. The effect of this function is to remove from the current disk-file directory those entries appearing in the argument list.

n.b. In the functions store, open, restore and wipeout, if any argument \(x\) possesses a PACK property the action described for that function is carried out for every element of that property list.

dimp \([\ ]\) :SUBR pseudo function

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

Background material

The content of this appendix falls into two distinct parts. The first three subsections contain the information which is referred to in the description of the various systems which have been implemented. The next three sub-sections contain the information referred to in the analysis of the practical work reported in this thesis.
B.1 Symbolic description of an internal representation

The box-notation is normally used to give the symbolic description of an internal representation. For example, an atomic symbol DID which stands for the global binding

\[ \text{DID} = (3 \times x) \]

and serves as the name of a user defined function may be represented as:

![Diagram of DID IDEN EXPR definition of a user defined function (3 * x)](image)

using the box-notation. For the sake of compactness we adopt the following conventions:

(a) property indicator with the corresponding value:

![Diagram of indicator value becomes](image)

(b) property indicator without the corresponding value:

![Diagram of indicator becomes](image)

(c) end-marker

![Diagram of NIL becomes](image)
(a) pointers

(b) continuation symbols

The two new symbols whose meaning must be described are:

"→ ..." : in the given context we are not interested in the continuing structure,

"... →" : in the given context, we are not interested in the preceding structure.

In terms of these new symbols, the symbolic description of the internal representation of DID in the above example takes the form:

In terms of these new symbols, the symbolic description of the internal representation of DID in the above example takes the form:

EXP.RID3NDID

To remove any confusion which may have arisen from the definition of the terminology which has been introduced so far, we relate these terms to their corresponding expressions in the example given in section 3.3.1.

Object:

EXP

natural value:

ax + b

attributes:

\[
\begin{align*}
\frac{d}{dx}(ax+b) &= a; \quad \frac{d}{dk}(ax+b) = 0; \quad \int (ax+b)dx = \frac{a}{2}x^2+bx; \quad \text{linear}(x; ax+b) = T \\
\end{align*}
\]

attribute-list associated with ax + b:

\[
\begin{align*}
ax+b &\leftarrow \begin{cases}
\frac{d}{dx}(ax+b) &= a; \quad \frac{d}{dk}(ax+b) = 0; \quad \int (ax+b)dx = \frac{a}{2}x^2+bx; \\
\text{linear}(x; ax+b) &= T
\end{cases}
\end{align*}
\]
object-value associated with EXP

\[ \text{EXP} = \{ ax+b \leftarrow \{ \frac{d}{dx} (ax+b) = a; \frac{d}{dx} (ax+b) = 0; \int (ax+b) \, dx = \left(\frac{a}{2}\right)x^2 + bx; \}ight\} \]

attribute-indicators

\(D; \int\); LINEAR

user defined functions (marked functions)

\(D; \int\); LINEAR

derived-values

\(a; 0; \frac{a}{2}x^2 + bx; T\)

B.2 Rules of differentiation and the conventions

B.2.1 Rules of differentiation

In the previous chapters all the examples involving differentiation are evaluated according to the following rules:

\(D0\) \(\frac{d}{dx}(c) = 0\), where \(c\) is a constant

\(D1\) \(\frac{d}{dx}(x) = 1\)

\(D2\) \(\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx}\)

\(D3\) \(\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}\)

\(D4\) \(\frac{d}{dx}(u + v) = u \frac{d}{dx}(v) + v \frac{d}{dx}(u)\)

\(D5\) \(\frac{d}{dx}(u/v) = \frac{v \frac{d}{dx}(u) - u \frac{d}{dx}(v)}{v^2}\)

\(D6\) \(\frac{d}{dx}(u^n) = nu^{n-1} \frac{d}{dx}(u)\)

\(D7\) \(\frac{d}{dx}(\# u) = \# \frac{d}{dx}(u)\)

\(D8\) \(\frac{d}{dx}(e^n) = e^u \frac{d}{dx}(u)\)

\(D9\) \(\frac{d}{dx}(\log u) = \frac{1}{u} \frac{du}{dx}\)

I "\#" stands for unary minus.
We notice that differentiation is a recursive function. For example in evaluating
\[ \frac{d}{dx} (x^2 + 2x + 3) \]
we have:
\[ \frac{d}{dx} (x^2 + 2x + 3) = \frac{d}{dx} (x^2) + \frac{d}{dx} (2x + 3) \]
\[ \frac{d}{dx} (2x + 3) = \frac{d}{dx} (2x) + \frac{d}{dx} (3) \]
which is equivalent to directly writing,
\[ \frac{d}{dx} (x^2 + 2x + 3) = \frac{d}{dx} (x^2) + \frac{d}{dx} (2x) + \frac{d}{dx} (3) \]

2.2. Conventions

There are many instances where examples are used as illustrations. In most instances the function used is differentiation, whose set of rules have just been given. Below we use one example to describe the conventions according to which most of the illustrative examples are presented.

Let us consider the evaluation:
\[ \frac{d^2}{dx^2} (x^2 e^x) \]
where,
\[ \frac{d}{dx} (x^2 e^x) = x^2 e^x + 2xe^x \]
we further assume that the attributes associated with objects with natural-values \( x^2 \) and \( e^x \) are
\[ \frac{d}{dx} (x^2) = 2x \]
\[ \frac{d}{dx} (e^x) = e^x \]
respectively. Below we show the format of the representation:
\[ \frac{d^2}{dx^2} (x^2 e^x) = \frac{d}{dx} (\frac{d}{dx} (x^2 e^x)) \]
\[
\frac{d}{dx} \left( x^2 e^x + 2xe^x \right) = \frac{d}{dx} \left( x^2 e^x \right) + \frac{d}{dx} \left( 2xe^x \right) \quad (\text{SUBS})
\]

E1
\[
\frac{d}{dx} \left( x^2 e^x \right) = x^2 \frac{d}{dx} (e^x) + e^x \frac{d}{dx} (x^2) \quad (D4)
\]
\[
\frac{d}{dx} (e^x) \rightarrow e^x \quad (AL)
\]
\[
\frac{d}{dx} (x^2) \rightarrow 2x \quad (AL)
\]

E1
\[
\frac{d}{dx} \left( x^2 e^x \right) = x^2 e^x + e^x \cdot 2x \quad (\text{SUBS})
\]

E2
\[
\frac{d}{dx} \left( 2xe^x \right) = 2x \frac{d}{dx} (e^x) + e^x \frac{d}{dx} (2x) \quad (D4)
\]
\[
\frac{d}{dx} (e^x) \rightarrow e^x \quad (AL)
\]
\[
\frac{d}{dx} (2x) = 2 \quad (D4, D0, D1)
\]

E2
\[
\frac{d}{dx} \left( 2xe^x \right) = 2xe^x + e^x + e^x \cdot 2 \quad (\text{SUBS})
\]

1
\[
\frac{d^2}{dx^2} \left( x^2 e^x \right) = (x^2 + 4x + 2) e^x \quad (\text{SUBS})
\]

In the above presentation we have:

(a) Step 1 contains two sub-evaluations (i.e. E1 and E2). As shown above, these sub-evaluations are underlined, labelled and evaluated from left to right.

(b) The numbers and labels appearing on the left hand side act as brackets. Their first and second appearances indicate the initiation and termination of an evaluation or of a sub-evaluation, respectively. For brevity only those evaluation steps subsequently referred to are numbered.

(c) the use of " \rightarrow " indicates that the corresponding value is obtained from the attribute-list, while " = " indicates that the value is obtained by direct evaluation,
(d) the labels used on the right hand side fall into one of the following classes.

(i) the label of the differentiation rule used to carry out the evaluation - e.g. D2, D4,

(ii) SUBS indicates that the required value is obtained by substituting the previously obtained results,

(iii) AL indicates that the required value is obtained from the attribute-list,

(iv) the label of a previous evaluation.

B.3 Traces

The step by step evaluation of examples used to illustrate the process of evaluation in some of the chapters differs in two respects from the corresponding trace which is provided in this appendix. First, the trace is in the prefix polish notation. This is because the prefix notation is the language for the input and output of the host system. Second, we have not abided by the simplification rules of the system. For example, in our presentation in Section (3.3.3) we have written

\[
\frac{d}{dx} (a + (x - b)^n)^m = mn (a + (x - b)^n)^{m-1} (x - b)^{n-1}
\]

while according to the simplification rules of the system we have:

\[
\frac{d}{dx} (a + (x - b)^n)^m = m(a + (x - b)^n)^{m-1} n (x - b)^{n-1}
\]

host system:

The set of commands needed to provide the trace of the evaluation:

\[
\frac{d^2}{dx^2} (A + (x - b)^n)^m
\]

are:

\[
\text{TRACE}((\text{DRV})) \notin
\text{MULDRV}(2 X( + A( - X B)^n)) \notin
\]

where a description of the functions used appears in the appendix E.
DRV(X X) = 1
DRV(X B) = 0
DRV(X(X(- X B))) = 1
DRV(X( `(- X B)(- N 1)`) = ( *(N 1) `(- X B)(- N 1)`) )
DRV(X X) = 0
DRV(X X) = 1
DRV(X B) = 0
DRV(X(X(- X B))) = 1
DRV(X( `(- X B)(- N 1)`) = ( *(N 1) `(- X B)(- N 1)`) )

This is followed by the two commands which activate the trace mechanism and initiate the required evaluation - i.e.

TRACE((DRV))
MULDRV(2 X ( `U H))

A description of the functions used above appears in Appendices C and E.
**B.9**

**FDS: fixed-size system attribute-list:**

In this instance we declare the function `drv` as a memo function and specify a limit of 10 on its number of attributes—i.e.

\[
\text{MEMO((\text{DRV} \ 10)) = 0}
\]

The following two commands activate the trace package and initiate the evaluation process:

\[
\text{TRACE((\text{DRV})) = 0}
\]

\[
\text{MULDRV(2 X) \ 1((+ A( \ 1(- X B))(\ 1(- N 1)))) = 0}
\]

A reader interested in the \(M\)-expression definitions of the above functions is referred to Appendices C and E.

**FDS:**

\[
\text{MEMO((\text{DRV} \ 10)) = 0}
\]

\[
\text{TRACE((\text{DRV})) = 0}
\]

\[
\text{MULDRV(2 X) \ 1((+ A( \ 1(- X B))(\ 1(- N 1)))) = 0}
\]

A reader interested in the \(M\)-expression definitions of the above functions is referred to Appendices C and E.
The three commands which are needed to declare `dry` as a memo function, activate the trace package and initiate the evaluation process are as follows:

```
# TRACER
DRV(X A) = 0
DRV(X X) = 1
DRV(X B) = 0
DRV(X(- X B)) = 1
```

where a description of the above function appears in Appendices C and E.

```
EDS: Dynamic attribute-list:
```

```
RESULT:
```

```
EDS: Dynamic attribute-list:
```

```
where a description of the above function appears in Appendices C and E.
```

```
```
** Derived-Value From Attribute-List: **

\[
\text{DFV}(X(\uparrow A (\uparrow (- X B) N))(- M (- N 1))) = (\ast N (\uparrow (- X B) (- N 1)))
\]

\[
\text{DFV}(X(\uparrow (+ A (\uparrow (- X B) N))(- M (- N 1))) = (\ast (* (- M 1) (\uparrow (+ A (- X B) N))(- N 1)))
\]

\[
\text{DFV}(X(\uparrow A (\uparrow (- X B) N))(- M (- N 1))) = (\ast N (\uparrow (- X B) (- N 1)))
\]

\[
\text{DFV}(X(\uparrow A (\uparrow (- X B) N))(- N 1))) = (\ast N (\uparrow (+ A (- X B) N))(- M (- N 1)))
\]

\[
\text{DFV}(X(\uparrow A (\uparrow (- X B) N))(- N 1))) = (\ast N (\uparrow A (- X B) N))(- M (- N 1)))
\]

\[
\text{DFV}(X(\uparrow A (\uparrow (- X B) N))(- N 1))) = (\ast N (\uparrow A (- X B) N))(- M (- N 1)))
\]

\[
\text{RESULT: --}
\]

\[
(+ (* (\ast M (\uparrow (+ A (- X B) N))(- N 1))) (\ast N (\uparrow A (- X B) N)))
\]

\[
(* (\ast N (- M 1)) (\uparrow (+ A (- X B) N))(- N 1)))
\]

\[
(- M 1)) (\ast N (\uparrow (- X B) (- N 1)))
\]

** B.4 A set of random numbers **

The set of random numbers listed below is used in some of the evaluations containing functions with numerical arguments. In particular, all the evaluations concerning the function \text{prs} are carried out using this set of random numbers.

\[
\begin{align*}
17 & 43 & 1 & 65 & 36 & 81 & 10 & 80 & 46 & 2 \\
53 & 45 & 47 & 23 & 67 & 39 & 83 & 97 & 87 & 1 \\
71 & 68 & 15 & 28 & 84 & 99 & 91 & 99 & 29 & 31 \\
73 & 11 & 55 & 82 & 49 & 5 & 33 & 7 & 95 & 52 \\
12 & 51 & 48 & 79 & 88 & 65 & 53 & 81 & 58 & 71 \\
51 & 17 & 96 & 73 & 21 & 19 & 81 & 59 & 60 & 96 \\
6 & 28 & 62 & 67 & 97 & 25 & 30 & 6 & 13 & 73 \\
69 & 7 & 73 & 24 & 36 & 53 & 48 & 58 & 63 & 9 \\
40 & 3 & 10 & 87 & 79 & 41 & 88 & 13 & 31 & 21 \\
98 & 46 & 42 & 92 & 21 & 52 & 71 & 7 & 51 & 92 \\
41 & 74 & 21 & 12 & 33 & 34 & 59 & 23 & 23 & 84 \\
48 & 80 & 93 & 91 & 60 & 93 & 69 & 24 & 72 & 68 \\
14 & 22 & 55 & 77 & 19 & 70 & 96 & 97 & 72 & 7 \\
49 & 75 & 59 & 35 & 25 & 88 & 50 & 59 & 48 & 12 \\
90 & 76 & 1 & 67 & 44 & 14 & 38 & 50 & 6 & 37 \\
20 & 34 & 76 & 99 & 59 & 16 & 15 & 94 & 81 & 90 \\
64 & 24 & 17 & 36 & 10 & 83 & 63 & 80 & 64 & 13 \\
47 & 17 & 32 & 86 & 77 & 42 & 6 & 9 & 50 & 74 \\
42 & 39 & 4 & 23 & 53 & 61 & 43 & 61 & 26 & 56
\end{align*}
\]
### B.5 Tables

The tables included in this section contain only a fraction of the data obtained from the practical work undertaken. The contents of these tables are used in the discussion parts of previous chapters.
### B.5.4 Differentiation - 1 (Section 2.3.1)

#### Host system:

<table>
<thead>
<tr>
<th>order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>160</td>
<td>562</td>
<td>1607</td>
<td>4623</td>
<td>14279</td>
<td>48608</td>
<td>x</td>
</tr>
<tr>
<td>apply entry</td>
<td>258</td>
<td>202</td>
<td>579</td>
<td>1671</td>
<td>5173</td>
<td>17632</td>
<td>x</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>47</td>
<td>139</td>
<td>528</td>
<td>x</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>72</td>
<td>x</td>
</tr>
</tbody>
</table>

#### FDS: Permanent attributes:

<table>
<thead>
<tr>
<th>order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>66</td>
<td>349</td>
<td>1131</td>
<td>3452</td>
<td>10973</td>
<td>37839</td>
<td>x</td>
</tr>
<tr>
<td>apply entry</td>
<td>25</td>
<td>127</td>
<td>411</td>
<td>1257</td>
<td>4003</td>
<td>13819</td>
<td>x</td>
</tr>
<tr>
<td>total searches</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>22</td>
<td>66</td>
<td>221</td>
<td>x</td>
</tr>
<tr>
<td>attributes available</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>22</td>
<td>66</td>
<td>221</td>
<td>x</td>
</tr>
<tr>
<td>time in seconds</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>34</td>
<td>109</td>
<td>408</td>
<td>x</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>35</td>
<td>x</td>
</tr>
</tbody>
</table>

#### FDS: Permanent attributes:

<table>
<thead>
<tr>
<th>order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>66</td>
<td>349</td>
<td>1131</td>
<td>3452</td>
<td>10973</td>
<td>37839</td>
<td>x</td>
</tr>
<tr>
<td>apply entry</td>
<td>25</td>
<td>127</td>
<td>411</td>
<td>1257</td>
<td>4003</td>
<td>13819</td>
<td>x</td>
</tr>
<tr>
<td>total searches</td>
<td>2</td>
<td>11</td>
<td>39</td>
<td>126</td>
<td>143</td>
<td>1445</td>
<td>x</td>
</tr>
<tr>
<td>attributes available</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>22</td>
<td>66</td>
<td>221</td>
<td>x</td>
</tr>
<tr>
<td>time in seconds</td>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>38</td>
<td>121</td>
<td>455</td>
<td>x</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>52</td>
<td>x</td>
</tr>
</tbody>
</table>
## EDS: Fixed-size System attribute-list:

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>160</td>
<td>443</td>
<td>936</td>
<td>1799</td>
<td>3228</td>
<td>5455</td>
<td>26088</td>
<td>X</td>
</tr>
<tr>
<td>apply entry</td>
<td>58</td>
<td>160</td>
<td>342</td>
<td>664</td>
<td>1198</td>
<td>2028</td>
<td>9400</td>
<td>X</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>16</td>
<td>35</td>
<td>72</td>
<td>139</td>
<td>252</td>
<td>1025</td>
<td>X</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>25</td>
<td>56</td>
<td>110</td>
<td>234</td>
<td>X</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>23</td>
<td>40</td>
<td>77</td>
<td>343</td>
<td>X</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>88</td>
<td>X</td>
</tr>
</tbody>
</table>

## EDS: Dynamic system attribute-list:

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>160</td>
<td>443</td>
<td>936</td>
<td>1799</td>
<td>3228</td>
<td>5455</td>
<td>8748</td>
<td>13522</td>
</tr>
<tr>
<td>apply entry</td>
<td>58</td>
<td>160</td>
<td>342</td>
<td>664</td>
<td>1198</td>
<td>2028</td>
<td>3250</td>
<td>5012</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>16</td>
<td>35</td>
<td>72</td>
<td>139</td>
<td>252</td>
<td>431</td>
<td>703</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>25</td>
<td>56</td>
<td>110</td>
<td>197</td>
<td>328</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>22</td>
<td>37</td>
<td>72</td>
<td>145</td>
<td>344</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

## EDS: Dynamic attribute-list:

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>170</td>
<td>466</td>
<td>999</td>
<td>1929</td>
<td>3470</td>
<td>5878</td>
<td>9423</td>
<td>14012</td>
</tr>
<tr>
<td>apply entry</td>
<td>58</td>
<td>160</td>
<td>346</td>
<td>672</td>
<td>1210</td>
<td>2044</td>
<td>3575</td>
<td>5702</td>
</tr>
<tr>
<td>total searches</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>25</td>
<td>56</td>
<td>110</td>
<td>197</td>
<td>328</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>25</td>
<td>56</td>
<td>110</td>
<td>193</td>
<td>312</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>22</td>
<td>35</td>
<td>67</td>
<td>133</td>
<td>275</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
**FDS:** Fixed-size system attribute-list of 100 entries with auto mode of selectivity inserted in the rule of differentiation: \( \frac{d}{dx} u^n = nu^{n-1} \frac{du}{dx} 

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>162</td>
<td>288</td>
<td>528</td>
<td>928</td>
<td>1524</td>
<td>2675</td>
<td>16753</td>
</tr>
<tr>
<td>apply entry</td>
<td>60</td>
<td>107</td>
<td>197</td>
<td>347</td>
<td>569</td>
<td>991</td>
<td>6255</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>10</td>
<td>21</td>
<td>40</td>
<td>71</td>
<td>131</td>
<td>787</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>24</td>
<td>110</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>22</td>
<td>40</td>
<td>250</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

**FDS:** dynamic system attribute-list with auto mode of selectivity in the rule of differentiation: \( \frac{d}{dx} (u^n) = nu^{n-1} \frac{du}{dx} 

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>162</td>
<td>288</td>
<td>528</td>
<td>928</td>
<td>1524</td>
<td>2352</td>
<td>3448</td>
</tr>
<tr>
<td>apply entry</td>
<td>60</td>
<td>107</td>
<td>197</td>
<td>347</td>
<td>569</td>
<td>875</td>
<td>1277</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>10</td>
<td>21</td>
<td>40</td>
<td>71</td>
<td>118</td>
<td>185</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>21</td>
<td>95</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>22</td>
<td>35</td>
<td>79</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
**FDS:** With fixed-size system attribute-list of 100 entries and a selectivity function

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>216</td>
<td>441</td>
<td>1138</td>
<td>3332</td>
<td>10636</td>
<td>38162</td>
<td>271970</td>
</tr>
<tr>
<td>apply entry</td>
<td>86</td>
<td>174</td>
<td>452</td>
<td>1331</td>
<td>4262</td>
<td>15284</td>
<td>110821</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>14</td>
<td>37</td>
<td>110</td>
<td>353</td>
<td>1260</td>
<td>5300</td>
</tr>
<tr>
<td>attribute available</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>27</td>
<td>73</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>27</td>
<td>49</td>
<td>126</td>
<td>217</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>19</td>
<td>41</td>
</tr>
</tbody>
</table>

**FDS:** Dynamic system attribute-list and a selectivity function

<table>
<thead>
<tr>
<th>Order of differentiation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>216</td>
<td>441</td>
<td>1138</td>
<td>3332</td>
<td>10636</td>
<td>33781</td>
<td>262555</td>
</tr>
<tr>
<td>apply entry</td>
<td>86</td>
<td>174</td>
<td>452</td>
<td>1331</td>
<td>4262</td>
<td>15149</td>
<td>104932</td>
</tr>
<tr>
<td>total searches</td>
<td>7</td>
<td>14</td>
<td>37</td>
<td>110</td>
<td>353</td>
<td>1251</td>
<td>4912</td>
</tr>
<tr>
<td>attribute available</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>30</td>
<td>81</td>
</tr>
<tr>
<td>time in seconds</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>27</td>
<td>49</td>
<td>104</td>
<td>174</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>17</td>
<td>36</td>
</tr>
</tbody>
</table>
### B.5.2 Differentiation 2 (Section 2.3.2)

<table>
<thead>
<tr>
<th></th>
<th>EDS: Permanent attributes:</th>
<th>FDS: Permanent attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage in cells</td>
<td>7000 6000 5000 4000</td>
<td>7000 6000 5000 4000 3000</td>
</tr>
<tr>
<td>eval entry</td>
<td>40695 40695 40695</td>
<td>15492 15492 15492 15492 15492</td>
</tr>
<tr>
<td>apply entry</td>
<td>14387 14387 14387</td>
<td>5520 5520 5520 5520 5520</td>
</tr>
<tr>
<td>time in seconds</td>
<td>417 432 461</td>
<td>167 169 175 183 192</td>
</tr>
<tr>
<td>garbage collections</td>
<td>46 59 89</td>
<td>16 19 26 32 52</td>
</tr>
<tr>
<td>eval entry</td>
<td>15492 15492 15492 15492 15492</td>
<td>5520 5520 5520 5520 5520</td>
</tr>
<tr>
<td>apply entry</td>
<td>5520 5520 5520 5520 5520</td>
<td>167 169 175 183 192</td>
</tr>
<tr>
<td>total searches</td>
<td>100 100 100 100 100</td>
<td>16 19 26 32 52</td>
</tr>
<tr>
<td>attributes available</td>
<td>100 100 100 100 100</td>
<td>100 100 100 100 100</td>
</tr>
<tr>
<td>time in seconds</td>
<td>167 169 175 183 192</td>
<td>178 180 189 197 208</td>
</tr>
<tr>
<td>garbage collections</td>
<td>16 19 26 32 52</td>
<td>16 19 28 32 52</td>
</tr>
</tbody>
</table>
### FDS: Fixed-size system attribute-list:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>7581</td>
<td>7581</td>
<td>7581</td>
<td>7581</td>
<td>7581</td>
<td>7581</td>
</tr>
<tr>
<td>apply entry</td>
<td>2732</td>
<td>2732</td>
<td>2732</td>
<td>2732</td>
<td>2732</td>
<td>2732</td>
</tr>
<tr>
<td>total searches</td>
<td>298</td>
<td>298</td>
<td>298</td>
<td>298</td>
<td>298</td>
<td>298</td>
</tr>
<tr>
<td>attributes available</td>
<td>102</td>
<td>102</td>
<td>102</td>
<td>102</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>time in seconds</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>99</td>
<td>105</td>
<td>126</td>
</tr>
<tr>
<td>garbage collections</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>45</td>
</tr>
</tbody>
</table>

### EDS: Dynamic system attribute-list:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>6114</td>
<td>6164</td>
<td>6234</td>
<td>6712</td>
<td>7419</td>
<td>8012</td>
</tr>
<tr>
<td>apply entry</td>
<td>2091</td>
<td>2107</td>
<td>2138</td>
<td>2314</td>
<td>3303</td>
<td>3633</td>
</tr>
<tr>
<td>total searches</td>
<td>179</td>
<td>176</td>
<td>171</td>
<td>185</td>
<td>197</td>
<td>216</td>
</tr>
<tr>
<td>attributes available</td>
<td>179</td>
<td>176</td>
<td>171</td>
<td>185</td>
<td>197</td>
<td>216</td>
</tr>
<tr>
<td>time in seconds</td>
<td>65</td>
<td>68</td>
<td>73</td>
<td>78</td>
<td>93</td>
<td>101</td>
</tr>
<tr>
<td>garbage collections</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>18</td>
<td>42</td>
</tr>
</tbody>
</table>

### FDS: Dynamic system attribute-list:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>5422</td>
<td>5494</td>
<td>5817</td>
<td>6787</td>
<td>7829</td>
<td>9421</td>
</tr>
<tr>
<td>apply entry</td>
<td>1965</td>
<td>1991</td>
<td>2109</td>
<td>2449</td>
<td>2826</td>
<td>3012</td>
</tr>
<tr>
<td>total searches</td>
<td>216</td>
<td>220</td>
<td>231</td>
<td>263</td>
<td>306</td>
<td>331</td>
</tr>
<tr>
<td>attributes available</td>
<td>80</td>
<td>82</td>
<td>85</td>
<td>95</td>
<td>106</td>
<td>114</td>
</tr>
<tr>
<td>time in seconds</td>
<td>71</td>
<td>73</td>
<td>79</td>
<td>91</td>
<td>111</td>
<td>133</td>
</tr>
<tr>
<td>garbage collections</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>14</td>
<td>23</td>
<td>57</td>
</tr>
</tbody>
</table>
**EDS: Dynamic system attribute-list:** Length of expressions:

<table>
<thead>
<tr>
<th>length in atoms</th>
<th>eval entry</th>
<th>apply entry</th>
<th>total searches</th>
<th>attributes available</th>
<th>time in seconds</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>24997</td>
<td>8861</td>
<td>975</td>
<td>320</td>
<td>328</td>
<td>59</td>
</tr>
<tr>
<td>20</td>
<td>10940</td>
<td>3969</td>
<td>419</td>
<td>176</td>
<td>143</td>
<td>18</td>
</tr>
<tr>
<td>30</td>
<td>9046</td>
<td>3248</td>
<td>343</td>
<td>139</td>
<td>119</td>
<td>15</td>
</tr>
<tr>
<td>40</td>
<td>7213</td>
<td>2624</td>
<td>290</td>
<td>114</td>
<td>96</td>
<td>12</td>
</tr>
<tr>
<td>50</td>
<td>7159</td>
<td>2604</td>
<td>288</td>
<td>113</td>
<td>95</td>
<td>11</td>
</tr>
<tr>
<td>60</td>
<td>6263</td>
<td>2271</td>
<td>253</td>
<td>96</td>
<td>93</td>
<td>10</td>
</tr>
<tr>
<td>70</td>
<td>6263</td>
<td>2271</td>
<td>253</td>
<td>96</td>
<td>93</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>5939</td>
<td>2151</td>
<td>241</td>
<td>90</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td>90</td>
<td>5939</td>
<td>2151</td>
<td>241</td>
<td>90</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td>100</td>
<td>5939</td>
<td>2151</td>
<td>241</td>
<td>90</td>
<td>79</td>
<td>9</td>
</tr>
</tbody>
</table>

**B.5.3 prs (Section 2.3.3)**

**Host system: recursive definition:**

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval entry</td>
<td>1043479</td>
<td>1043479</td>
<td>1043479</td>
<td>1043479</td>
<td>1043479</td>
</tr>
<tr>
<td>apply entry</td>
<td>384281</td>
<td>384281</td>
<td>384281</td>
<td>384281</td>
<td>384281</td>
</tr>
<tr>
<td>time in seconds</td>
<td>3260</td>
<td>3648</td>
<td>3700</td>
<td>3730</td>
<td>3810</td>
</tr>
<tr>
<td>garbage collections</td>
<td>130</td>
<td>150</td>
<td>180</td>
<td>220</td>
<td>300</td>
</tr>
</tbody>
</table>
### Host system: iterative definition:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>eval entry</th>
<th>apply entry</th>
<th>time in seconds</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>1078328</td>
<td>407337</td>
<td>3429</td>
<td>204</td>
</tr>
<tr>
<td>6000</td>
<td>1078328</td>
<td>407337</td>
<td>3462</td>
<td>239</td>
</tr>
<tr>
<td>5000</td>
<td>1078328</td>
<td>407337</td>
<td>3518</td>
<td>284</td>
</tr>
<tr>
<td>4000</td>
<td>1078328</td>
<td>407337</td>
<td>3556</td>
<td>339</td>
</tr>
<tr>
<td>3000</td>
<td>1078328</td>
<td>407337</td>
<td>3642</td>
<td>434</td>
</tr>
</tbody>
</table>

### EDS: Permanent attributes: recursive definition:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>eval entry</th>
<th>apply entry</th>
<th>total searches</th>
<th>attributes available</th>
<th>time in seconds</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>44572</td>
<td>15440</td>
<td>2440</td>
<td>488</td>
<td>178</td>
<td>6</td>
</tr>
<tr>
<td>6000</td>
<td>44572</td>
<td>15440</td>
<td>2440</td>
<td>488</td>
<td>178</td>
<td>6</td>
</tr>
<tr>
<td>5000</td>
<td>44572</td>
<td>15440</td>
<td>2440</td>
<td>488</td>
<td>180</td>
<td>8</td>
</tr>
<tr>
<td>4000</td>
<td>44572</td>
<td>15440</td>
<td>2440</td>
<td>488</td>
<td>184</td>
<td>10</td>
</tr>
<tr>
<td>3000</td>
<td>44572</td>
<td>15440</td>
<td>2440</td>
<td>488</td>
<td>188</td>
<td>12</td>
</tr>
</tbody>
</table>

### FDS: Permanent attributes: iterative definition:

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>eval entry</th>
<th>apply entry</th>
<th>total searches</th>
<th>attributes available</th>
<th>time in seconds</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>873930</td>
<td>301083</td>
<td>500</td>
<td>102</td>
<td>2896</td>
<td>104</td>
</tr>
<tr>
<td>6000</td>
<td>873930</td>
<td>301083</td>
<td>500</td>
<td>102</td>
<td>2918</td>
<td>120</td>
</tr>
<tr>
<td>5000</td>
<td>873930</td>
<td>301083</td>
<td>500</td>
<td>102</td>
<td>2960</td>
<td>144</td>
</tr>
<tr>
<td>4000</td>
<td>873930</td>
<td>301083</td>
<td>500</td>
<td>102</td>
<td>2984</td>
<td>176</td>
</tr>
<tr>
<td>3000</td>
<td>873930</td>
<td>301083</td>
<td>500</td>
<td>102</td>
<td>3048</td>
<td>240</td>
</tr>
</tbody>
</table>
**FDS: Fixed-size system attribute-list of 100 entries, recursive definition:**

<table>
<thead>
<tr>
<th>Metric</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage in cells</td>
<td>7000</td>
<td>6000</td>
<td>5000</td>
<td>4000</td>
<td>3000</td>
</tr>
<tr>
<td>eval entry</td>
<td>14318</td>
<td>14318</td>
<td>14318</td>
<td>14318</td>
<td>14318</td>
</tr>
<tr>
<td>apply entry</td>
<td>4722</td>
<td>4722</td>
<td>4722</td>
<td>4722</td>
<td>4722</td>
</tr>
<tr>
<td>total searches</td>
<td>896</td>
<td>896</td>
<td>896</td>
<td>896</td>
<td>896</td>
</tr>
<tr>
<td>attributes available</td>
<td>498</td>
<td>498</td>
<td>498</td>
<td>498</td>
<td>498</td>
</tr>
<tr>
<td>time in seconds</td>
<td>84</td>
<td>86</td>
<td>86</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>garbage collections</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FDS: Fixed-size system attribute-list of 100 entries, iterative definition:**

<table>
<thead>
<tr>
<th>Metric</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage in cells</td>
<td>7000</td>
<td>6000</td>
<td>5000</td>
<td>4000</td>
<td>3000</td>
</tr>
<tr>
<td>eval entry</td>
<td>225004</td>
<td>225004</td>
<td>225004</td>
<td>225004</td>
<td>225004</td>
</tr>
<tr>
<td>apply entry</td>
<td>82588</td>
<td>82588</td>
<td>82588</td>
<td>82588</td>
<td>82588</td>
</tr>
<tr>
<td>total searches</td>
<td>5839</td>
<td>5839</td>
<td>5839</td>
<td>5839</td>
<td>5839</td>
</tr>
<tr>
<td>attributes available</td>
<td>4614</td>
<td>4614</td>
<td>4614</td>
<td>4614</td>
<td>4614</td>
</tr>
<tr>
<td>time in seconds</td>
<td>605</td>
<td>645</td>
<td>690</td>
<td>743</td>
<td>811</td>
</tr>
<tr>
<td>garbage collections</td>
<td>52</td>
<td>61</td>
<td>76</td>
<td>91</td>
<td>116</td>
</tr>
</tbody>
</table>
B.6 Commands

This section contains the commands, (in their S-expression forms which are acceptable to the systems), used to perform some of the evaluations, the outcomes of which were used in the discussion parts of the previous chapters. Since in most cases the output spreads over several pages of print, all outputs have been excluded from this presentation.

B.6.1 Differentiation - 1 (Section 2.3.1)

Host system

evaluations:

MULDRV(1 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(2 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(3 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(4 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(5 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(6 X (↑(↑ A (↑(- X B)n))m)) ≠
MULDRV(7 X (↑(↑ A (↑(- X B)n))m)) ≠

EDS: Permanent attributes

Objects:

IDEN((V(- X B)) ≠
IDEN((U(↑ A (↑ V N))) ≠

attributes:

ATTRIB((
((DRV X V))((DRV X U))) ≠

evaluations:

MULDRV(1 X (↑ U M)) ≠
MULDRV(2 X (↑ U M)) ≠
MULDRV(3 X (↑ U M)) ≠
MULDRV(4 X (↑ U M)) ≠
MULDRV(5 X (↑ U M)) ≠
MULDRV(6 X (↑ U M)) ≠
MULDRV(7 X (↑ U M)) ≠

EDS: Permanent attributes

memo function:

MEXO((DRV)) ≠

permanent attributes:

ATTRIB((
((DRV X(- X B))((DRV X(↑ A (↑(- X B)n)))) )) ≠

evaluations:

The set of commands needed here is identical to those in the host system listed above.
FDS: fixed-size attribute-list
memo function:

```
MEMO((DRV 100))
```
evaluations:

The commands used here are identical to those in the host system listed above.

FDS: dynamic attribute-list

The commands needed here are identical to those in the host system listed above.

FDS: dynamic attribute-list

```
MEMO((DRV))
```

To illustrate the use of commands for altering the life-time numbers and the rate of promotion:

```
PROMOTE(1)
INITIAL(4)
DECREMENT(1)
INCREMENT(2)
```

Notice that these numbers are identical to the ones which have been built into the system. The set of commands needed to carry out the required evaluation is identical to those in the host system listed above.

Selectivity: function driven systems

(a) Separate selectivity function:

```
DEFINE((
 (SEL (LAMBDA(X))
 (COND
 ((EQ (CAADR X) (QUOTE t)) T)
 (T NIL)) )))
SELECT((DRV SEL))
```

(b) Use of structure of differentiation function:

```
SELECTOR((DRV))
CHANGE(DRV AT SUBS
 (AUTO (* (* V(† U(- V 1))(DRV X U)))) )
```

B.6.2 Differentiation - 2 (section 2.3.2)

Host system

\[
\text{MULDRV}(3 \times \text{OBJ12})
\]

EDS: Permanent attributes

Objects:

\[
\text{IDENT(OBJ1(- (\uparrow X 2)(+ X 1)))} \notin \\
\text{IDENT(OBJ2(\uparrow \text{OBJ1 I}))} \notin \\
\text{IDENT(OBJ3(+ (\uparrow X 2)((\ast 4 X)1)))} \notin \\
\text{IDENT(OBJ4(\uparrow \text{OBJ3 J}))} \notin \\
\text{IDENT(OBJ5(+ 2(LOG OBJ4))}) \notin \\
\text{IDENT(OBJ6(\uparrow \text{OBJ7 K}))} \notin \\
\text{IDENT(OBJ7(- (\ast 3(\uparrow X 2))(- X 1)))} \notin \\
\text{IDENT(OBJ8(\uparrow \text{OBJ7 L}))} \notin \\
\text{IDENT(OBJ9(+ 3(EXP OBJ8)))} \notin \\
\text{IDENT(OBJ10(\uparrow \text{OBJ9 M}))} \notin \\
\text{IDENT(OBJ11(\ast OBJ2(\ast OBJ6 OBJ10))} \notin \\
\text{IDENT(OBJ12(\uparrow \text{OBJ11 N}))} \notin \\
\]

Permanent attributes:

\[
\text{ATTRIB((}
\text{((DRV X OBJ1))((DRV X OBJ2))((DRV X OBJ3))}
\text{((DRV X OBJ4))((DRV X OBJ6))((DRV X OBJ7))}
\text{((DRV X OBJ9))((DRV X OBJ10))((DRV X OBJ12))})
\]

Evaluation:

\[
\text{MULDRV}(3 \times \text{OBJ12})
\]

FDAS: Permanent attributes

memo function:

\[
\text{MEMO((DRV))} \notin
\]

attributes:

\[
\text{ATTRIB((}
\text{((DRV X (- (\uparrow X 2)(+ X 1)))})
\text{((DRV X (\uparrow (- (\uparrow X 2)(+ X 1)I))))}
\text{((DRV X (+ (\uparrow X 2)((\ast 4 X)1))))}
\text{((DRV X (\uparrow (+ (\uparrow X 2)((\ast 4 X)1))))})
\text{((DRV X (+ 2(LOG (\uparrow (+ (\uparrow X 2)((\ast 4 X)1))))))})
\text{((DRV X (- (\ast 3(\uparrow X 2))(- X 1))))}
\text{((DRV X (\uparrow (- (\ast 3(\uparrow X 2))(- X 1)I))))}
\text{((DRV X (+ 3(EXP (\uparrow (- (\ast 3(\uparrow X 2))(- X 1)I))))))}
\text{((DRV X (\uparrow (+ 3(EXP (\uparrow (- (\ast 3(\uparrow X 2))(- X 1)I))))))})
\]

B.24
We notice that in creating the above attributes, the appropriate expressions could have been named and used, as was the case in the expression driven system. However, it is worthwhile to name the expressions especially if the evaluation involves a considerable amount of attribute housekeeping. Operations.

evaluation:

\[
\text{MUL} \text{DRV}(3 \ X \ ( \{ \{ (\hat{t} (- (\hat{t} X 2)) (\hat{t} X 1))I \} \\
 \{ (\hat{t} (+ 2 (\hat{t} \log (\hat{t} (+ (\hat{t} X 2)) (-(\hat{t} X 1))I)))K \} \\
 (\hat{t} (+ 3 (\hat{t} \exp (\hat{t} (-(\hat{t} X 2)) (-(\hat{t} X 1))L)))M))N)))G
\]

**FDS:** fixed-size attribute-list

memo function

\[
\text{MEMO}((\text{DRV} 100)) G
\]

evaluation:

The command issued is identical to that in the host system given above.

**EDS:** dynamic attribute-list

The command needed here is identical to that in the host system given above.

**FDAS:** dynamic attribute-list

\[
\text{MEMO}((\text{DRV})) G
\]

This command is followed by the one used to evaluate the result in the host system.

**Selectivity:** expression driven system with dynamic attribute-lists

and function driven system with fixed-size attribute-lists

(a) Separate selectivity function:

\[
\text{DEFIN}E(\\n\text{(SEL)(\text{\text{LAMBDA}(X)}})\\n\text{COID}\\n\\n((\text{(EQ(CADDR X)(QUOTE *))T})\\n(T \text{NIL})) G) G\\n\text{SELECT}((\text{DRV SEL})) G
\]
(b) Use of structure for selectivity:

```
SELECTOR((DRY))
CHANGE(DRY A5 SUBS (AUTO(+ (AUTODRV X V))
                        (* V(AUTODRV X U))))
```

(c) Length of expressions:

```
DEFINE((
    (SEL(LAMBDA(X Y)
         ((EQ,(LENGTH X)Y)T)
         (T NIL)))))
```

B.6.3 prs (section 2.3.3)

Host system
evaluation:

```
PRS(N) /
```

The values assumed by \( N \) are the set of 500 randomly chosen numbers listed in section (B.5.1).

FDS: Permanent attributes

memo functions:

```
MEMO((PRS PRIME)) /
```

Permanent attributes:

```
ATTRIB((
    ((PRS 5))((PRS 10))((PRS 15))((PRS 20))
    ((PRS 25))((PRS 30))((PRS 35))((PRS 40))
    ((PRS 45))((PRS 50))((PRS 55))((PRS 60))
    ((PRS 65))((PRS 70))((PRS 75))((PRS 80))
    ((PRS 85))((PRS 90))((PRS 95))((PRS 100)) )
```

evaluation:

```
PRS(N) /
```

The values assumed by \( N \) are the set of 500 randomly chosen numbers listed in section (B.5.1).

FDS: fixed-size attribute-lists

memo functions:

```
MEMO((PRS 100)(PRIME 100)) /
```

evaluations:

The issued commands are identical to those in the host system listed above.
FDS: dynamic attribute-list

memo functions:

MEMO((PRS PRIME)) ≠

evaluations:

The issued command is the same as that in the host system listed above.

Selectivity: function driven systems

```
DEFINE((
  (SEL(LAMBDA(X))
  (COND
    ((ZEHOP(REMAINDER X 5))T)
    (T NIL))))) ≠
SELECT((PRS SEL)) ≠
```

B.6.4 Classification

Differentiation - 2 problem

```
DEFRULES((
  (DIFFLAW(DARG NIL
  A1 (( t( *( t(-( * A1( t X2)+( * B1 X)C1)))I)
  (* t(+ D1(LOG( t(+(* A2( t X 2))
  (-( * B2 X)C2))(J))))K)
  (t(+ D 2(EXP( t(-( * A3( t X 2))
  (-( * B3 X)C3))(L))))M)))N)
  (LIST X A1 B1 C1 A2 B2 C2 A3 B3 C3 D1 D2 I J K L M N)
  (-( * T4 T5 T6)(+ T9)(+ T1 T2 T3)))))) ≠
IDEET(DIFF( t( *( t(-( * A1( t X2)+( * B1 X)C1)))I)
  (* t(+ D1(LOG( t(+(* A2( t X 2))
  (-( * B2 X)C2))(J))))K)
  (t(+ D 2(EXP( t(-( * A3( t X 2))
  (-( * B3 X)C3))(L))))M)))W)) ≠
CLASSES((DIFF)) ≠
DEFCLASS((DIFFLAW DIFF NIL)) ≠
ATTRIB(((MULDRV 3 X DIFF)) ≠
JOF((DRV DIFF)) ≠
determinant
DEFINISH((
  (DETLAW(LAMBDA(X)(PROG(UVW)
  (SETQ Y X))
  A (COND
    ((NULL X)(GO C))
    (EQ(LENGTH(CAR X))5)(GO B))
    (T NIL))
  B (SETQ X (CDR X))
  (GO A))
```
(COND
  ((NULL V) (RETURN W)))
(SETQ W (NCONC W (CAR V)))
(SETQ V (CDR V))
(GO C) )))) )

DEFINITION(
  LENGTH (LAMBDA (X) (PROG (U)
    (SETQ U 0)
    (COND
      ((NULL X) (RETURN U)))
    (SETQ X (CDR X))
    (SETQ U (ADD1 U))
    (GO A) ))) )

IDENT(DEFP((A1 B1 C1 D1 E1)(A2 B2 C2 D2 E2)(A3 B3 C3 D3 E3)
  (A4 B4 C4 D4 E4)(A5 B5 C5 D5 E5)))

CLASSES(((DEF))

DEFCLASS(((DEFCLASS NIL)))

ATTRIB(((DEF5 DEF)))

JOP(((DEF5 DEF)))
APPENDIX C

Repertoire of Commands

This appendix gives the M-expression definitions for most of the commands which have been described in the previous chapters. Note that no M-expression definition is provided in cases where commands have a trivial definition. The M-expression definitions are usually followed by simple examples indicating the manner in which the commands are used. All the commands presented are in their S-expression form which is acceptable to the system.

The given M-expression definitions should not be interpreted too literally. This is because the functions involved have been hand-coded in the ICL assembly language, PLAN, and in many cases where recursion is specified by the definitions, the actual code is a store and transfer.

C.1 Modifications to the functions of the interpreter

In the M-expression definitions of \texttt{apply} and \texttt{eval} provided in Section (A.1) each line is numbered. In order to implement permanent and temporary attributes in the expression and function driven systems, the definitions of \texttt{apply} and \texttt{eval} are modified as follows (the line numbers correspond to those in Section A.1).

\begin{enumerate}
\item \texttt{get[fn;EXPR]} \rightarrow \texttt{[attribute exists} \rightarrow \texttt{derived-value;}}
\item \texttt{get[fn;RULE]} \rightarrow \texttt{[attribute exists} \rightarrow \texttt{derived-value;}}
\item \texttt{get[fn;SUBR]} \rightarrow \texttt{[attribute exists} \rightarrow \texttt{derived-value;}}
\end{enumerate}

What the modifications above ensure is that a function belonging to any of the specified types is only evaluated directly when it is established that the corresponding attribute does not exist.

In order to make use of attribute-lists at all levels of recursion, and in order to insert the attribute search routine at only one point within the interpreter, the following lines in the definition of \texttt{eval} are modified as shown:

\begin{verbatim}
atom[car[form]] \rightarrow [get[car[form]; EXPR]v
 [get[car[form]; RULE]v
 [get[car[form]; SUBR] \rightarrow apply[car[form]]
 evlis[ cdr[form]; a]; a];
\end{verbatim}

1. The value of \texttt{get} is set aside. This is the meaning of the apparent free or undefined variable.
The modifications above ensure that the evaluation of functions of the specified types is transferred to apply.

**C.2 Declarative commands**

**Ident:**

(a) M-expression definition:

\[
\text{ident}(x; y) = \text{prog}(u_1; u_2);
\]

\[
u_1 := \text{get}(x; \text{IDENT});
\]

\[
equal(\text{car}(u_1); y) \rightarrow \text{go}(A0);
\]

\[
u_2 := \text{cons}(\text{IDENT}; \text{cons}(\text{cons}(y; \text{NIL}); \text{NIL}));
\]

\[
u_1 := \text{cdr}(x);
\]

\[
\text{rplacd}(x; u_2);
\]

\[
\text{rplacd}(\text{cdr}(u_2); u_1);
\]

\[
\text{objstack}(y);
\]

\[
A0 \quad \text{return} [11.0.0].
\]

(b) Illustration:

The command needed to create an object called \(a1\) with the natural-value \((bx + z)\) is

\[
\text{IDENT}(A1 \ ((* . B \ X) 2)) \$
\]

**Identq:**

(a) M-expression definition:

\[
\text{identq}(x; y) = \text{ident}(x; \text{evaluate(\text{car}(y); \text{cdr}(y))}).
\]

(b) Illustration:

The command needed to create an object called \(a2\), having the result of the differentiation \(\frac{d}{dx}(x^2)\) as its natural-value, is:

\[
\text{IDENTQ} (A2 \ (\text{DRV} X \ (\# X \ 3))) \$
\]

The function returns \((* . 3(\# X \ 2))\) as its value.

**Memo:**

(a) M-expression definitions:

\[
\text{memo}(x) = \text{prog}([u_1; u_2]);
\]

\[
A0 \quad \text{null}(x) \rightarrow \text{return} [0];
\]

\[
u_1 := \text{car}(x);
\]

\[
\text{atom}(u_1) \rightarrow [\text{get}(u_1; \text{NOTE}) \rightarrow \text{go}(A3); \text{go}(A1)];
\]

\[
\text{get}(\text{car}(u_2); \text{NOTE}) \rightarrow \text{[REWRITE NOTE LIMIT AND ADJUST THE SIZE OF THE SYSTEM ATTRIBUTE-LIST APPROPRIATELY; go}(A3)];
\]

\[
A1 \quad u_2 := \text{cons}(\text{NOTE}; \text{cons}(\text{cons}(\text{NIL}; \text{NIL}); \text{NIL}));
\]

\[
\text{atom}(u_1) \rightarrow \text{go}(A2);
\]

\[
\text{rplacd}(\text{cdr}(u_2); \text{cdr}(u_1));
\]

\[
\text{rplacd}(\text{cdr}(u_1); u_2);
\]

\[
\text{go}(A3);
\]

I **objstack**

This function relates the natural-value and the object-value of an object.

II by return [0] we mean that no value is returned.
A2  rplacd[cdru2];cdr[u1];
rplacd[cdru1];u2;
A3  x:=cdr[x];
go[A0]].

C.3 Defining and editing facilities
EDS: permanent attributes
attrib:

(a) M-expression definition:
attrib[x] = prog[u1;u2;u3;u4;u5;u6];
A0  null[x]—return[0];
u1:=caar[x];
u2:=cadar[x];
u6:=get[lastcar[u1];IDEN];
null[u6]—print[quote[EXPRESSION lastcar[u1]HAS NOT BEEN DECLARED];go[A4]]; 

u3:=get[u6;car[u1]]; 
not[null[u2]]—[u4:=u2;go[A1]]; 
u4:=evaluate[car[u1];cdr[u1]]; 
A1  null[u5]—gq[A2]; 
u5:=assoc[u1;u3;NIL];
null[u5]—go[A2];
equal[cdr[u5];u4]—go[A4];
rplacd[u5;u4];
go[A4];
A2  u4:=cons[cons[cons[u1;u4];NIL];
null[u3]—go[A5]; 
A3  u4:=cons[car[u1];cons[u4;NIL]]; 
A4  u5:=cdr[u6];
rplacd[u6;u4];
rplacd[cdru4;u5]];
go[A0]].

(b) Illustration:
The set of commands needed to create the following permanent attributes:

(1) for ax where a is a constant,
   (i) its derivatives with respect to x and a
   (ii) its linearity with respect to x
   (iii) its integral with respect to x
(2) for (RED CELLS), its colour is red,
(3) for (ICE COLD WATER), its temperature is zero.
The definitions of the two functions colour and temperature are not available.

I I represents the remainder of the argument list of a marked function after the natural-value is deleted from it.
Commands to create the objects:

\[
\text{IDENT(A1( \ast A X))} \\
\text{IDENT(A2(RED CELLS))} \\
\text{IDENT(A3(ICE COLD WATER))}
\]

To create the permanent attributes required for the objects A1; A2; A3:

\[
\text{ATTRIB ((
((\text{DRV x A1})) ((\text{DRV A A1})) ((\text{LINEAR x A1})) ((\text{INT x A1}))
((\text{COLOUR A2) RED}) ((\text{TEMPERATURE A3) ZERO}) ))}
\]

destatts

(a) M-expression definition:

\[
\text{destatts}[x] = [\text{prog}[u_1; u_2; u_3; u_4];
\]

\[
\text{AO}
\]

\[
\text{null}[x] \rightarrow \text{return}[0];
\]

\[
u_1 := \text{car}[x];
\]

\[
\text{atom}[u_1] \rightarrow \text{[u_2 := get}[u_1; \text{IDEN}];
\]

\[
\text{null}[u_2] \rightarrow \text{[quote} \text{THE ATOMIC SYMBOL u_1 IS NOT AN OBJECT NAME}]; \text{go}[A1];
\]

\[
\text{rplacd} [u_2; \text{NIL}];
\]

\[
\text{go}[A1];
\]

\[
u_2 := \text{get}[\text{lastcar}[u_1]; \text{IDEN}];
\]

\[
\text{not}[\text{null}[u_2]] \rightarrow \text{go}[A2];
\]

\[
u_3 := \text{get}[\text{car}[u_1]; \text{IDEN}];
\]

\[
\text{null}[u_3] \rightarrow \text{[quote} \text{lastcar}[u_1]; \text{car}[u_1] \text{ARE NOT OBJECT NAMES}]; \text{go}[A1];
\]

\[
u_4 := \text{get}[u_3; \text{cadr}[u_1]);
\]

\[
\text{null}[u_4] \rightarrow \text{[quote} \text{ATTRIBUTE-INDICATOR cadr}[u_1] \text{DOES NOT EXIST ON THE ATTRIBUTE-LIST}]; \text{go}[A1];
\]

\[
\text{remprop}[u_3; \text{cadr}[u_1]];\]

\[
\text{go}[A1];
\]

\[
u_3 := \text{get}[u_2; \text{car}[u_1]];\]

\[
\text{null}[u_3] \rightarrow \text{[quote} \text{ATTRIBUTE-INDICATOR car}[u_1] \text{DOES NOT EXIST ON THE ATTRIBUTE-LIST}]; \text{go}[A1];
\]

\[
u_4 := \text{sassoc}[u_1; u_3; \text{NIL}];\]

\[
\text{null}[u_4] \rightarrow \text{[quote} \text{THE SPECIFIED ATTRIBUTE u_1 DOES NOT EXIT}]; \text{go}[A1];
\]

\[
\text{delatts}[u_3; u_4];\]

\[
\text{A1}
\]

\[
x := \text{cdr}[x];\]

\[
\text{go}[A0].
\]

(b) Illustration:

Given the objects and object-values created by \text{attrib}, a single command which performs the following operations:

I \text{lastcar}[x]  
This function returns the last element of its argument list as its value.

I \text{delatts}[x ;y]  
Given a list of argument-result pairs \( x \), this function deletes the specified argument-result pair \( y \) from it.
(i) destroys all the attributes of the object A2,
(ii) destroys all the attributes of the object A1
associated with INT,
(iii) destroys the specified attribute \( \frac{d}{da} (a1) = 0 \) from the
attribute-list of the object A1.

DESTATTS((A2 (A1 INT) (DRV A A1 )))

**dispatts:**

(a) M-expression definition:

\[
\text{dispatts}[x] = \text{prog}[u1; u2; u3; u4; u5];
\]

\[
\begin{align*}
A0 & \quad \text{null}[x] \rightarrow \text{return}(0); \\
& \quad u1 := \text{car}[x]; \\
& \quad \text{atom}[u1] \rightarrow \text{go}[A1]; \\
& \quad u2 := \text{get}[\text{car}[u1]; \text{IDEN}]; \\
& \quad \text{not}[\text{null}[u2]] \rightarrow \text{go}[A5]; \\
& \quad u2 := \text{get}[\text{lastcar}[u1]; \text{IDEN}]; \\
& \quad \text{null}[u2] \rightarrow \text{print}[\text{quote}[\text{lastcar}[u1] \text{ IS NOT AN OBJECT}]]; \\
& \quad \text{go}[A6]; \\
& \quad \text{print}[\text{quote}[\# \text{NATURAL VALUE:-}]]; \\
& \quad \text{print}[\text{car}[u2]]; \\
& \quad \text{print}[\text{quote}[\# \text{ATTRIBUTES:-}] ]; \\
& \quad u3 := \text{assoc}[u1; u2; \text{NIL}]; \\
& \quad \text{null}[u3] \rightarrow \text{print}[\text{quote}[\text{ATTRIBUTE u1 DOES NOT EXIST}]]; \\
& \quad \text{go}[A8]; \\
& \quad \text{print2}[\text{car}[u3]; \text{quote}[:]; \text{conc}[\text{car}[u3]; \text{car}[u2]]; \\
& \quad \text{quote}[:]; \text{cdr}[u3]]; \\
& \quad \text{go}[A8]; \\
A1 & \quad u2 := \text{get}[u1; \text{IDEN}]; \\
& \quad \text{null}[u2] \rightarrow \text{print}[\text{quote}[\text{u1 IS NOT AN OBJECT}]]; \text{go}[A8]; \\
& \quad \text{print}[\text{quote}[\# \text{NATURAL VALUE:-}]]; \\
& \quad \text{print}[\text{car}[u2]]; \\
& \quad \text{print}[\text{quote}[\# \text{ATTRIBUTES:-}]]; \\
& \quad u3 := \text{cdr}[u2]; \\
& \quad \text{null}[u3] \rightarrow \text{print}[\text{quote}[\text{NULL ATTRIBUTE-LIST}]]; \text{go}[A8]; \\
& \quad u5 := \text{car}[u3]; \\
A2 & \quad u4 := \text{cdr}[u5]; \\
A3 & \quad \text{print2}[u5; \text{quote}[:]; \text{conc}[\text{caar}[u4]; \text{car}[u2]]; \text{quote}[:]; \\
& \quad \text{cdr}[u4]]; \\
& \quad u4 := \text{cdr}[u4]; \\
& \quad \text{null}[u4] \rightarrow \text{go}[A4]; \\
& \quad \text{go}[A3]; \\
A4 & \quad u3 := \text{cddr}[u3]; \\
& \quad \text{null}[u3] \rightarrow \text{go}[A8]; \\
& \quad \text{go}[A2];
\end{align*}
\]

* prin2 places the information in its argument list into an output buffer without subsequently printing it.*
null[u2] → [print[quote[car[u1] IS NOT AN OBJECT]];go[A8]];  
print[quote[# # NATURAL VALUE:-]];  
print[car u2];  
print[quote[# # ATTRIBUTES:-]];  
u5:=cadr[u1];  
u4:=get[u2;u5];  
null[u4] → [print[quote[ATTRIBUTE-INCIDATOR cadr[u1] DOES NOT EXIST]];go[A8]];  
print2[u5]; quote [];print[quote[ATTRIBUTES:-]];  
cadr[u4]];  
null[u4]-> Sprint[quote[L a TTRIBUTE-INCIDATOR cadr Lull DOES NOT EXIST]];go LA8] ];  
u4:=cadr[u4];  
mull[u4]-> go[A6];  
go[A6];  
x:=cdr[x];  
go[A6]].

(b) Illustration:
Given the object created by attrib, a command which will print
out the following information:
(i) all the attributes of the object A3,
(ii) all the attributes of A1 associated with DRV;
(iii) the specific attribute colour [a2]= red
has the form:

\[ \text{DISPATS ((A3 (A1 DRV) (COLOUR A2)))} \%

This command causes the following information to be printed out:

\[ \# NATURAL VALUE:-
(ICE COLD WATER)
\# ATTRIBUTES:-
TEMPERATURE: NIL=ZERO
\# NATURAL VALUE:-
(* A X)
\# ATTRIBUTES:-
DRV: X=A
DRV: A=X
\# NATURAL VALUE:-
(RED CELLS)
\# ATTRIBUTES:-
COLOUR: NIL=RED

swopatts:

(a) M-expression definitions:

\[ \text{swopatts}[x] = [\text{prog}[u1;u2;u3;u 4;u5];}
\]

A0

null[x]—> return[0];  
u1:=caar[x];  
u2:=cadr[x];  
u3:=get[lastcar[u1];IDEN];  
u4:=get[lastcar[u2];IDEN];  
and[not[mull[u3]];not[mull[u4]]] — go[A1];  
or[not[mull[u3]];not[mull[u4]]—> print[quote[lastcar[u1];

lastcar[u2] ARE DIFFERENT OBJECTS]]; go[A3]];  
equal[car[u2];car[u1]] — go[A2];  
print[quote[car[u1];car[u2] ARE DIFFERENT OBJECTS]] ;

go[A5];
A2 \[ u_3 := \text{get}[\text{car}[u_1]; \text{IDEN}]; \]
\[ \text{pointer}[u_3; \text{cadr}[u_1]; \text{cadr}[u_2]]; \]
\[ \text{pointer}[u_3; \text{get}[u_3; \text{cadr}[u_1]]; \text{get}[u_3; \text{cadr}[u_2]]]; \]
\[ \text{go}[A3]; \]

A1 \[ \text{equal}[\text{car}[u_1]; \text{car}[u_2]] \rightarrow \text{go}[A4]; \]
\[ \text{print}[\text{quote}[\text{ATTRIBUTE-INDICATORS} \text{car}[u_1]; \text{car}[u_2] \text{ARE}\]
\[ \text{DIFFERENT}]]; \]
\[ \text{go}[A3]; \]

A4 \[ u_4 := \text{get}[u_3; \text{car}[u_1]]; \]
\[ u_5 := \text{sassoc}[u_4; u_1_; \text{NIL}]; \]
\[ u_3 := \text{sassoc}[u_4; u_2_; \text{NIL}]; \]
\[ \text{null}[u_5] \rightarrow [A5 \text{print}[\text{THE SPECIFIED ATTRIBUTE DOES NOT}
\[ \text{EXIST}]]; \text{go}[A3]]; \]
\[ \text{null}[u_3] \rightarrow \text{go}[A5]; \]
\[ \text{pointer}[u_4; u_5; u_3]; \]

A3 \[ x := \text{cdr}[x]; \]
\[ \text{go}[A0]. \]

(b) Illustration:

Given the objects created by \text{attrib}, a command which performs
the following tasks:
(i) interchanges the positions of the two attributes \[ \frac{d}{dx}(ax) = a \]
and \[ \frac{d}{dx}(ax) = x, \]
(ii) interchanges the positions of the two attribute-
indicators \text{DRV} and \text{INT} together with their corresponding
values.

has the form,
\[ \text{SWOPATTS} (((\text{DRV} x A_1) (\text{DRV} A A_1))
\[ (((A_1 \text{DRV})(A_1 \text{INT})))) \]

\text{prinstck:}

(a) M-expression definition:
\[ \text{prinstck}[x] = \text{[prog}[u_1; u_2; u_3]; \]
\[ u_1 := \text{value of the object-stack pointer} \]
\[ \text{print}[\text{quote}[\text{DO YOU WANT A DISPLAY OF NATURAL-VALUES?}]]; \]
\[ u_2 := \text{read[ ]}; \]
\[ \text{not}[\text{null}[x]] \rightarrow \text{go}[A0]; \]
\[ u_3 := 1; \]
\[ A1 \text{eq}[u_1; u_3] \rightarrow \text{return}[0]; \]
\[ \text{equal}[u_2; \text{YES}] \rightarrow [\text{print out the natural-value of the}
\[ u_3\text{th entry}; \text{go}[A3]]; \]
\[ \text{print[object-value of the } u_3\text{th entry}]; \]
\[ A3 u_3 := u_3 + 1; \]
\[ \text{go}[A1]; \]

\[ \text{pointer}[x; y; z] \]

This function is used to swop the positions of the two
pointers \text{y} and \text{z} on the list \text{x}.
(a) M-expression definition:

\[
A \quad \text{keepobjs} = \text{keepobjs}[x] = \text{[prog}[u; v];
\]

\[
\begin{align*}
A & : \text{null}[u] \rightarrow \text{return}[0]; \\
  & u := x; \\
  & \text{null}[u] \rightarrow \text{return}[0]; \\
  & v := \text{car}[u]; \\
  & \text{greaterp} \quad [\text{cadr}[v]; \text{value of the pointer of the object-stack}] \\
  & \rightarrow \text{[print}[\text{quote}[\text{SUCH A DUMMY-OBJECT DOES NOT EXIST}]]; \\
  & \text{return}[0]); \\
  & \text{rplacd} \quad [\text{car}[v]; \text{cons}[\text{IDENT}; \text{cons}[\text{object-value of the} \\
  & \text{cadr}[v]; \text{with entry}; \text{NIL}]]]); \\
  & u := \text{cadr}[u]; \\
  & \text{go}[A1]. \\
\end{align*}
\]
(b) Illustration:
Transfer the two dummy objects, whose object-values are assumed to be occupying the entries 2 and 4 of the object-stack, to the class of user defined objects by naming them EXP1 and EXP2 respectively. Notice that the user can only use this command successfully if he is aware of the appropriate entry numbers which can be obtained using the previous command printrck.

command:

\[
\text{KEEP0BJS((EXP1 2) (EXP2 4))}
\]

FDS: Permanent attribute

attrib:

(a) M-expression definition:

\[
\text{attrib} = \text{prog[u1;u2;u3;u4]};
\]

\[
\begin{align*}
A3 & \quad \text{null}[x] \to \text{return}[0]; \\
A2 & \quad u1 := \text{car}[x]; \\
A1 & \quad u2 := \text{cadar}[x]; \\
A0 & \quad u3 := \text{get[car[u1;ROTE]}]; \\
& \text{null}[u3] \to \left[\text{print[quote[car[u1] IS NOT A MEMO FUNCTION]]}; \text{go}[A2]\right]; \\
& \text{not[null[u2]]} \to \text{go}[A0]; \\
& \text{evaluate[car[u1];cdr[u1]]}; \\
& \text{null}[u4] \to \text{go}[A2]; \\
& \text{equal[cdr[u4];u2]} \to \text{go}[A2]; \\
& \text{rplacd[u4]:=}u2; \\
& \text{rplacd[u3]:=}u4; \\
& x:=\text{cdr}[x]; \text{go}[A3];
\end{align*}
\]

(b) Illustration:
The necessary commands to create the same attributes using the same functions and argument expressions given in connection with attrib described in the expression driven system are listed below.

To declare the necessary functions:

\[
\text{MEMO((DRV COLOUR TEMPERATURE))}
\]

To create the attributes required:

\[
\text{ATTRIB((
(DRV X (* A X)) ((DRV A (* A X)))
(LINEAR X (* A X)) ((INT X (* A X)))
(COLOUR (RED CELLS)) RED)
(TEMPERATURE (ICE COLD WATER)) ZERO))}
\]

Note that in this case the expression need not be named.

destatts:

(a) M-expression definition:

\[
destatts = \text{prog[u1;u2;u3;u4]};
\]

\[
\begin{align*}
A0 & \quad \text{null}[0x] \to \text{return}[0]; \\
A2 & \quad u1 := \text{car}[x]; \\
A1 & \quad \text{atom[u1]} \to \text{go}[A2]; \\
A0 & \quad u2 := \text{car[u1]}; \\
A1 & \quad u3 := \text{get[u2;ROTE]}; \\
A0 & \quad \text{null}[u3] \to \left[\text{A3 Print[quote[u2 IS NOT A MEMO FUNCTION]]}; \text{go}[A1]\right];
\end{align*}
\]
(b) Illustration

Given the attribute-list created by `attrib`, a single command which performs the following tasks:

(i) destroys the entire attribute-list of `LINEAR`,
(ii) destroys the attribute $\frac{dx}{dx}(ax) = a$.

has the form:

```
DESTATTS ((LINEAR; (DRV X (* A X))) )
```

**dispatts:**

(a) M-expression definition:

```
dispatts [x] = [prog[u;v;w];
A0 null[x] -> return[0];
w:=car[x];
atom[w] -> go[A3];
u:=get[car[w];ROTE];
A1 null[u] -> [print[quote[car[x] IS NOT A MEMO FUNCTION]]; go[A5]]; 
    u:=cdr[u];
A2 null[u] -> [print[quote [NO ENTRY ON THE USER ATTRIBUTE-LIST]]];
    go[A5];
    print[quote [# # ATTRIBUTES:-]]; 
    prin2 [car[w];quote[i];car[v];quote[=];cdr[v]]; 
    go[A5];
    u:=get[w;ROTE];
    null[u] -> go[A1]; 
    u:=cdr[u];
    null[u] -> go[A2];
    print[quote[# # ATTRIBUTES:-]]; 
    prin2[w;quote[i];car[u];quote[=];cdr[u]]; 
    u:=cdr[u];
    null[u] -> go[A5]; 
    go[A4];
    x:=cdr[x];
    go[A4];
    x:=cdr[x];
    go[A0]].
```

(b) Illustration:

Given the attribute-list created by `attrib`, a command which performs the following two tasks:
(i) prints out the entire user.attribute-list of DRV
(ii) prints out the attribute \( \int axdx = \frac{a}{2}x^2 \)

result:
```
;# ATTRIBUTES:-
DRV: (X(* A X)) = A
DRV: (A(* A X)) = X
;# ATTRIBUTES:-
INT: (X(* A X)) = (* (/12)(* A (X 2)))
```

**swopatts:**

(a) M-expression definition:
```
swopatts[x] = [prog[u1;u2;u3;u4;u5];
A0 null[x] → return[0];
u1:=caar[x];
u2:=caddr[x];
not[equal[car[u1];car[u2]] → [print[quote[car[u1];car[u2]]
UNIDENTICAL MEMO FORS]];go[A1]];
u3:=get[car[u1];ROTE];
null[u3] → [print[quote[car[u1] IS NOT A MEMO FUNCTION]];go[A1]];
u3:=cadr[u3];
null[u3] → [print[quote[NULLL ATTRIBUTE LIST]];go[A1]];
u4:=sassoc[cdr[u1];u3;NIL];
u5:=sassoc[cdr[u2];u3;NIL];
not[and[null[u5];null[u4]] → [print[quote[ATTRIBUTES CORRESPONDING TO car[x] DOES NOT EXIST]];go[A1]];
pointer[u3;u4;u5];
A1 x:=cadr[x];
go[A0]].
```

(b) Illustration:
The command which interchanges the positions of the two attributes in the user attribute-list created by attrib and associated with dry.

has the form:
```
SWOPATTS (((DRV X (* A X)) (DRV A (* A X))) )
```

**C.4 Classification commands**

**classes:**

(a) M-expression definition:
```
classes[x] = [prog[u;v];
A0 null[x] → return[0];
u:=car[x];
get[u;class] → go[A1];
v:=cons[CLASS;list[NIL]];rplacd[cdr[v];cdr[u]];rplacd[u;v];
```
defclass:
(a) M-expression definition:
\[
defclass[x] = [\text{prog}[u1; u2; u3; u4];
\]
\[
A0 \quad x := \text{cdr}[x];
\]
\[
go[\text{AO}].
\]

(b) Illustration:
The commands needed to create a class-object called LINEAR with the class natural-value \((ax + b)\) where \(a\) and \(b\) are constants and \text{linearlaw} may be taken as the classification function are listed below:

(i) declare the class:
\[
\text{CLASSES ([[LINEAR]])}
\]

(ii) create the class-object:
\[
\text{IDENT (LINEAR \ (+ (* A X) B))}
\]

(iii) in order to establish a relation between the class-object and the definition of the class and also detect whether \((ax + b)\) belongs to the specified class one of the two alternative forms of the commands is used. Without supplying the optional argument we have:
\[
\text{DEFCLASS ([[LINEARLAW (X LINEAR) NIL]])}
\]

and supplying the optional argument we have:
\[
\text{DEFCLASS ([[LINEARLAW (X LINEAR) (X A B)])}}
\]

memclass:
(a) M-expression definition:
\[
\text{memclass}[x] = [\text{prog}[u1; u2; u3; u4];
\]
\[
A0 \quad x := \text{cdr}[x];
\]
\[
go[\text{AO}].
\]
Illustration:
The necessary set of commands to establish the object with natural-value \((2x + 3)\) as the member of the class-object LINEAR created above is listed below.

(i) create the object called D1:

\[
\text{IDENT (D1 (+ (* 2 X) 3))} \rightarrow
\]

(ii) optional argument NIL:

\[
\text{MEMCLASS (((LINEAR (X D1) NIL)))} \rightarrow
\]

(iii) by supplying the optional argument \(- (X 2 3)\)

\[
\text{MEMCLASS (((LINEAR (X D1) (X 2 3)))} \rightarrow
\]

C.5 Miscellaneous commands

The commands to be described below have been added to the command repertoire of the host system and are therefore common to all the attribute systems implemented. While the first three commands deal with the implicit dependence of variables on each other, the last two commands were implemented for the practical work undertaken.

C.5.1 Implicit dependence

\texttt{depend[x]} \\
:SUBR pseudo function

(a) description
depend takes a list as its argument. Each element of this list is itself a list in which the first element depends implicitly on the remaining elements. The "dependence-list" appears as the value of the property indicator DEP on the property list of the atomic symbol \%DEP. For example, the dependence-list expressing the implicit dependence of \( x \) on \((y, t)\) and \( k \) on \((u, v)\) has the internal representation:

\[
\%DEP \quad DEP
\]

where the meaning of the symbols used is given in section (B.1).
Clearly, the state of the entire dependence-list must be reviewed and modified accordingly every time a new pair-list is added or the range of implicit dependence of the independent variables is extended. No value is returned and no error message is generated.

(b) M-expression definition:
\[
\text{depend}[x] = \text{prog}[u1;u2;u3;u4;u5;u6];
\]

\[
\begin{align*}
A0 & \text{null}[x] \rightarrow \text{return}[0]; \\
   & u1 := \text{car}[x]; \\
   & u2 := \text{get}[%\text{DEP}; \text{DEP}]; \\
   & u3 := \text{cons}[\text{car}[u1]; \text{cons}[(\text{cdr}[u1]; \text{NIL})]]; \\
   & \text{not}[	ext{null}[u2]] \rightarrow \text{go}[A1]; \\
   & u2 := u3; \\
   & u5 := \text{cons}[%\text{DEP}; \text{cons}[u3; \text{NIL}]]; \\
   & \text{rplacd}[(\text{cdr}[u5]; \text{cdr}[u5]; \text{DEP})]; \\
   & \text{rplacd}[\text{DEP}; u5]; \\
   & \text{go}[A5]; \\
   & u4 := u2; \\
A1 & \text{null}[u4] \rightarrow \text{go}[A4]; \\
   & \text{equal}[(\text{car}[u4]; \text{car}[u1]) \rightarrow \text{go}[A3]; \\
   & u5 := u4; \\
   & u4 := \text{cdr}[u4]; \\
   & \text{go}[A2]; \\
A3 & u6 := \text{union}[(\text{cdr}[u4]; \text{cdr}[u1]); \\
   & \text{rplacd}[(\text{cdr}[u5]; u6)]; \\
   & \text{go}[A5]; \\
A4 & \text{rplacd}[(\text{cdr}[u5]; u3); \\
   & u5 := u2; \\
A5 & u4 := \text{cdr}[u5]; \\
   & \text{get}[u4; \text{car}[u5]] \rightarrow \text{go}[A6]; \\
   & u5 := \text{cdr}[u5]; \\
   & \text{null}[u5] \rightarrow \text{go}[A7]; \\
   & \text{go}[A5]; \\
A6 & u4 := \\
   & \text{go}[A5]; \\
A7 & \text{null}[u2] \rightarrow \text{go}[A10]; \\
   & u4 := \text{car}[u2];
\end{align*}
\]
(c) Illustration:
A command which creates a dependence-list for the following variables:

(i) \( x \) implicitly depends on \( y \) and \( z \)
(ii) \( z \) implicitly depends on \( t \) and \( 0 \)
(iii) \( t \) implicitly depends on \( k \) and \( 1 \)

has the form:

\[ \text{DEPEND} (((X \ Y \ Z)(Z \ T \ 0)(T \ K \ L))) \]

The dependence-list created as the result of obeying this command has the form:

\[ (X(Y \ Z \ T \ 0 \ K \ L) \ Z(T \ O \ K \ L) \ T(K \ L)) \]

\[
\text{independ}[x] \quad \text{:SUBR pseudo function}
\]

(a) Description
This command is the antithesis of \texttt{depend} and destroys the specified implicit dependences. Although the argument list has the same format as in \texttt{depend} the elements of this list have the opposite interpretation, (first element of each list does implicitly depend on the remaining elements of that list). Again after deletion of an implicit dependence, the state of the entire dependence-list must be reviewed and modified accordingly. No value is returned. An error message is generated if the implicit dependence which is to be removed does not exist.

(b) M-expression definition:

\[
\text{independ}[x] = \texttt{[prog}[u1;u2;u3];}
\]

\[
\text{u2} = \texttt{get}\%\texttt{DEP};\texttt{DEP];}
\]

\[
\text{A0} \quad \text{null}[x] \rightarrow \texttt{return[0];}
\]

\[
\text{u1} = \texttt{car}[x];
\]

\[
\text{null}[u2] \rightarrow \texttt{[print}[\texttt{quote}[/\texttt{NULL DEPENDENCE-LIST}]];\texttt{go}[A3)];
\]

\[
\text{u3} = u2;
\]

\[
\text{A1} \quad \text{null}[u3] \rightarrow \texttt{go[A3]};
\]

\[
\text{equal}[\texttt{car}[u3];\texttt{car}[u1]] \rightarrow \texttt{go[A2]};
\]

\[
\text{u3} = \texttt{caddr}[u3];
\]

\[
\texttt{go}[A1];
\]

\[
\text{A2} \quad \texttt{rplaca}[\texttt{cdr}[u3]];\texttt{intersection}[\texttt{cdr}[u1];\texttt{cdr}[u3]]];
\]

\[
\text{A3} \quad x = \texttt{cdr}[x];
\]

\[
\texttt{go}[A0].
\]
(c) Illustration:

A command which will remove the implicit dependence of \( T \) on \( k \), \( L \) and \( X \) on \( T \), \( 0 \), \( K \), \( L \) in the dependence-list created by \texttt{depend} above has the form:

\[
\text{\texttt{INDEPEND (\((T K L)(X T 0 K L)\))}}
\]

The dependence-list remaining after the command has been executed has the form:

\[
(X (Y Z) Z(T 0 K L))
\]

\texttt{free\[x;y\]} : SUBR predicate

(a) Description

The effect of this command is to establish whether the first argument, \( x \), depends explicitly or implicitly on the second argument \( y \). If \( x \) does not depend on \( y \) the value \( \ast T \ast \) is returned; otherwise the value \( \text{NIL} \) is returned.

(b) M-expression definition:

\[
\text{\texttt{free\[x;y\]}} = \left[ \text{\texttt{prog\[v\];}} \right.
\]

\[
\text{\texttt{member\[x;y\]\to return\[\text{\texttt{NIL}}\];}}
\]

\[
\text{\texttt{v:=get\[\%DEP;\text{\texttt{DEPJ}}\];}}
\]

\[
\text{\texttt{null\[v]\to return\[\text{\texttt{T}}\];}}
\]

\[
\text{\texttt{v:=get\[v;y\];}}
\]

\[
\text{\texttt{null\[v]\to return\[\text{\texttt{T}}\];}}
\]

\[
\text{\texttt{member\[x;v\]\to return\[\text{\texttt{NIL}}\];}}
\]

\[
\text{\texttt{return\[\text{\texttt{T}}\].}}
\]

C.5.2 Other commands

\texttt{repeat\[x;n\]} : SUBR

This function takes an argument \( x \), which is itself a command, and evaluates it the number of times specified in \( n \). Once the evaluation is complete the following items of information are printed out in the order specified below:

- available storage space in cells \( m_1 \)
- number of eval entry \( m_2 \)
- number of apply entry \( m_3 \)
- number of garbage collections \( m_4 \)
- number of attribute-lists searched \( m_5 \)
- number of attributes available \( m_6 \)
- time of evaluation in seconds \( m_7 \)
- value of a single evaluation

An error message is generated if the definition of the function specified in \( x \) is not available.

\texttt{givecell\[n\]} : SUBR pseudo function

This function takes only one argument which is a positive integer. \( n \) specifies the number of free cells required in a free word list for an evaluation. The user is informed if the number of free cells required, \( n \), is larger than that which the system can make available. No value is returned.
APPENDIX D

Further Practical Results

In the "analysis of practical results" given in previous chapters, we have focussed our attention primarily on results arising from evaluations based on the problems described in Chapter 2. This appendix contains practical results and a brief discussion of evaluations arising from a variety of other problems described in Appendix E.

D.1 Vector analysis

The purpose of this example, chosen from the area of vector analysis, is to demonstrate how the storage space requirement of the expression and function driven systems differs, and how the classification feature can be used to solve problems whose solution cannot be evaluated in the systems with permanent attributes.

D.1.1 Systems with permanent attributes

So that a number of marked functions apply to the same memo expressions, we create the functions:

\[
\begin{align*}
\frac{\partial}{\partial x} & , \frac{\partial}{\partial y} , \frac{\partial}{\partial z} \\
\frac{\partial^2}{\partial x^2} & , \frac{\partial^2}{\partial y^2} , \frac{\partial^2}{\partial z^2} \\
\frac{\partial^3}{\partial x^3} & , \frac{\partial^3}{\partial y^3} , \frac{\partial^3}{\partial z^3}
\end{align*}
\]

all of which make use of the differentiation function, \(dry\), in order to evaluate their results. The \(M\)-expression definitions of these functions are provided in Section (E.10).

A typical example is the calculation of:

\[
\text{curl}_A; \quad \text{curlcurl}_A; \quad \text{curlcurlcurl}_A
\]

where \(A\) is the vector:

\[
A = A_1 \hat{\text{i}} + A_2 \hat{\text{j}} + A_3 \hat{\text{k}}
\]

For these evaluations, permanent attributes corresponding to the following intermediate results are provided.
\[
\begin{align*}
\frac{\partial A_1}{\partial x} &= D_{1x}[A_1]; \\
\frac{\partial A_1}{\partial y} &= D_{1y}[A_1]; \\
\frac{\partial A_1}{\partial z} &= D_{1z}[A_1]; \\
\frac{\partial^2 A_1}{\partial x^2} &= D_{2x}[A_1]; \\
\frac{\partial^2 A_1}{\partial y^2} &= D_{2y}[A_1]; \\
\frac{\partial^2 A_1}{\partial z^2} &= D_{2z}[A_1]; \\
\frac{\partial^3 A_1}{\partial x^3} &= D_{3x}[A_1]; \\
\frac{\partial^3 A_1}{\partial y^3} &= D_{3y}[A_1]; \\
\frac{\partial^3 A_1}{\partial z^3} &= D_{3z}[A_1];
\end{align*}
\]

In the expression driven systems, the following permanent attributes:
- \(D_{1x}[A_1]; D_{1y}[A_1]; D_{1z}[A_1]; D_{2x}[A_1]; D_{2y}[A_1]; D_{2z}[A_1]; D_{3x}[A_1]; D_{3y}[A_1]; D_{3z}[A_1]\)
are associated with the object with the natural-value \(A_1\), the following permanent attributes:
- \(D_{1x}[A_2]; D_{1y}[A_2]; D_{1z}[A_2]; D_{2x}[A_2]; D_{2y}[A_2]; D_{2z}[A_2]; D_{3x}[A_2]; D_{3y}[A_2]; D_{3z}[A_2]\)
are associated with the object with the natural-value \(A_2\), and the following permanent attributes:
- \(D_{1x}[A_3]; D_{1y}[A_3]; D_{1z}[A_3]; D_{2x}[A_3]; D_{2y}[A_3]; D_{2z}[A_3]; D_{3x}[A_3]; D_{3y}[A_3]; D_{3z}[A_3]\)
ares associated with the object with the natural-value \(A_3\). This is an instance where a number of marked functions have the same natural-values as their arguments.
In the function driven systems we have:

permanent attributes \( D_1 \times [A_1]; D_1 \times [A_2]; D_1 \times [A_3] \) associated with \( D_1 \)

\[
\begin{align*}
&D_1y[A_1]; D_1y[A_2]; D_1y[A_3] \quad D_1y \\
&D_1z[A_1]; D_1z[A_2]; D_1z[A_3] \quad D_1z \\
&D_2x[A_1]; D_2x[A_2]; D_2x[A_3] \quad D_2x \\
&D_2y[A_1]; D_2y[A_2]; D_2y[A_3] \quad D_2y \\
&D_2z[A_1]; D_2z[A_2]; D_2z[A_3] \quad D_2z \\
&D_3x[A_1]; D_3x[A_2]; D_3x[A_3] \quad D_3x \\
&D_3y[A_1]; D_3y[A_2]; D_3y[A_3] \quad D_3y \\
&D_3z[A_1]; D_3z[A_2]; D_3z[A_3] \quad D_3z
\end{align*}
\]

with the three components of the vector having the values:

\[
A_1 = \frac{\partial^2}{\partial x^2} (x^2+y^2+z^2)^n;\quad A_2 = \frac{\partial^2}{\partial y^2} (x^2+y^2+z^2)^n;\quad A_3 = \frac{\partial^2}{\partial z^2} (x^2+y^2+z^2)^n
\]

The permanent attributes listed above occupy 5249 cells and 7151 cells in the expression and function driven systems, respectively. None of the manipulations described above could be performed by either of the systems with a free word list of 10,000 cells. This is because the free word list must accommodate the definitions and the permanent attributes as well as providing an adequate working space - this is not possible in 10,000 cells.

With the three components of the vector having the values:

\[
A_1 = \frac{\partial^3}{\partial x^3} (x^2+y^2+z^2)^n;\quad A_2 = \frac{\partial^3}{\partial y^3} (x^2+y^2+z^2)^n;\quad A_3 = \frac{\partial^3}{\partial z^3} (x^2+y^2+z^2)^n
\]

the storage space required to accommodate the permanent attributes is estimated to be in the region of 19600 cells and 14500 cells in the function and expression driven systems, respectively.

From the examples above it is clear that, as the complexity of the common expressions in an evaluation increases, the difference in the storage space required by the two systems increases. This leads us to conclude that, in problems with a large number of complex common expressions, the use of EDS is to be preferred.

D.1.2 Classification

In this section we attempt to perform the manipulations described in section (D.1.1), and whose solutions cannot be evaluated using systems with permanent attributes due to the shortage of storage space. Consider the three components of the vector having the values:

\[
A_1 = \frac{\partial^2}{\partial x^2} (x^2+y^2+z^2)^n;\quad A_2 = \frac{\partial^2}{\partial y^2} (x^2+y^2+z^2)^n;\quad A_3 = \frac{\partial^2}{\partial z^2} (x^2+y^2+z^2)^n
\]
The symmetry property of the expression \((x+y+z)^n\) in the variables \(x, y\) and \(z\) leads to the following relations:

\[
\begin{align*}
dx[A1] &= dy[A2] \text{ with } x = y \\
dz[A3] &= dz[A3] \text{ with } x = z \\
d2x[A1] &= dy[A2] \text{ with } x = y \\
d2y[A2] &= dz[A3] \text{ with } x = z \\
d3x[A1] &= dz[A3] \text{ with } x = z \\
d3y[A2] &= dy[A2] \text{ with } x = y \\
d2z[A3] &= dy[A2] \text{ with } x = z \\
dz[A3] &= dz[A3] \text{ with } x = z \\
d2y[A2] &= d2x[A1] \text{ with } x = y \\
d2z[A3] &= d2x[A1] \text{ with } x = y \\
d3y[A2] &= d3x[A1] \text{ with } x = y \\
d3z[A3] &= d3x[A1] \text{ with } x = y \\
dy[A2] &= dx[A1] \text{ with } x = y \\
dz[A3] &= dz[A3] \text{ with } x = z \\
d2x[A1] &= d2y[A2] \text{ with } x = y \\
d2z[A3] &= d2x[A1] \text{ with } x = y \\
d3x[A1] &= d3z[A3] \text{ with } x = z \\
d3z[A3] &= d3x[A1] \text{ with } x = z \\
d3y[A2] &= d3y[A2] \text{ with } x = y \\
end{align*}
\]

It is clear from these relations that the creation of the class attributes:

\[
dx[A1]; \quad d2x[A1]; \quad d3x[A1]
\]

eliminates the need for all other permanent attributes which were created explicitly in the systems with permanent attributes, (Section D.1.1).

Below we list the commands, (in their S-expression form), needed to create the structure pertinent to the use of the classification feature. We take \(A1\) as the name of the class and the class object, and use the method of inspection to provide the necessary reduced a-lists. Since the reduced a-lists are provided there is no need to specify the definition of the classification function, and it is assumed to be NIL. The commands needed for this purpose are:

\[
\begin{align*}
\text{CLASSES}((A1))$  \\
\text{IDENTIQ}(A1((D2x((+(+(y2)((+(z2))))))))))$  \\
\text{IDENTIQ}(A2((D2y((+(+(y2)((+(z2))))))))))$  \\
\text{IDENTIQ}(A3((D2y((+(+(y2)((+(z2))))))))))$  \\
\text{DEFCLASS}(((NIL(X A 1) (X Y Z)) ))$  \\
\text{MEMCLASS}(((A1 (y A2) (x z)) (A1 (z A3) (x y z))))$  \\
\text{ATTRIB} (((Dx A1)) ((D2x A1)) ((D3x A1)) )$  \\
\end{align*}
\]

The structure created by these commands occupies 890 cells. This is a remarkable reduction in comparison with the EDS which requires 5249 cells to maintain the attribute-lists.

The use of the class object \(A1\), the similar objects \(A2\) and \(A3\), and the class attributes created above lead to the following practical result:

<table>
<thead>
<tr>
<th>evaluations</th>
<th>garbage collections</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>curlA</td>
<td>4</td>
<td>56</td>
</tr>
<tr>
<td>curlcurlA</td>
<td>24</td>
<td>259</td>
</tr>
<tr>
<td>curlcurlcurlA</td>
<td>132</td>
<td>918</td>
</tr>
</tbody>
</table>
Furthermore, taking as the value of the three components of the vector the expressions:

\[ A1 = \frac{\partial^3}{\partial x^3} (x^2 + y^2 + z^2)^n; \quad A2 = \frac{\partial^3}{\partial y^3} (x^2 + y^2 + z^2)^n; \quad A3 = \frac{\partial^3}{\partial z^3} (x^2 + y^2 + z^2)^n; \]

the use of the classification feature has reduced the storage space required to accommodate the objects from 14554 cells to 2210 cells in the EDS. The use of the three class attributes led to the following practical results:

<table>
<thead>
<tr>
<th>evaluations</th>
<th>garbage collections</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>curlA</td>
<td>13</td>
<td>119</td>
</tr>
<tr>
<td>curlcurlA</td>
<td>119</td>
<td>812</td>
</tr>
</tbody>
</table>

In this instance the function curlcurlcurlA cannot be evaluated due to the shortage of storage space.

In this particular case, therefore, the use of the classification feature has increased the manipulative power of the system as a result of its effect on the use of storage space.

D.2 Integration

This is an additional problem attempted in order to demonstrate how the use of the classification feature can effect the performance of the system. Consider the evaluations:

\[ \int x^{10} e^{ix} dx \quad i = 1, 2, \ldots, 100. \]

Using the host system, the computation of a single integral of the type given above takes 12 garbage collections and 189 seconds of evaluation time. Furthermore, during the solution of a single integral the user has to intervene on 10 separate occasions in order to sort out the queries which the system generates. This is because when the system cannot handle a situation it appeals to the user for help.

If the values of a number of these integrals are required, in the expression driven system an attribute is created for each of the integrals. Considering that each attribute occupies 1654 cells, the storage space can only accommodate a few of them.

This problem provides an ideal situation for the automatic use of the classification feature. Once the classification function:
D.6

DEFRULE((
(INTERN(DARG(NIL
A1 ((* X N)(EXP( * A X)))(LIST X N A)((N LESSP N 10))
((* T4)))
A2 (A NIL) )))) JT

the class of expressions:

\text{CLASSES((PEXP))}

the class object:

\text{IDENT(PEXP( *( X N)(EXP(* A X)))))}

DEFCCLASS(((INTERN(X PEXP) NIL)))

and the class attribute:

\text{ATTRIB(( (INT X PEXP) )))}

are defined, and the name of the class object is associated with the integration function:

\text{JOF((INT PEXP))}

then the system is capable of deciding on the class of expressions which are being integrated and, where possible, the class attribute is used to assist the evaluation.

The evaluations:

\[
\int_{x_1}^{x_{10}} e^{i x} \, dx \quad i = 1, 2, \ldots, 100
\]

were performed by making use of the information provided above. The total time in seconds and the number of garbage collections required are 167 and 31, respectively. In the host system, the same evaluations would have required 18900 seconds of evaluation time and 1200 garbage collections. This estimate was arrived at by evaluating one of the integrals in the host system and multiplying the result by the factor of 100. This represents a remarkable improvement in the manipulative power of the system. Finally, the storage space required to preserve the information regarding the classification of these 100 expressions is 2000 cells.

D.3 Greatest common divisor: GCD

This is an additional example used to show how the presence of temporary attributes can effect the performance of the host system. A rather inefficient definition of the \text{gcd} whose \text{M-expression definition} is provided in Section (E.5) was used to find the greatest common divisors of:
in the host system and function driven system with fixed-size attribute-lists. As the following table shows, due to the inefficient definition of gcd this evaluation proves to be costly in terms of both the evaluation time and the number of garbage collections required to solve the problem in the host system.

<table>
<thead>
<tr>
<th>storage in cells</th>
<th>7000</th>
<th>6000</th>
<th>5000</th>
<th>4000</th>
<th>3000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>time in seconds</td>
<td>5997</td>
<td>6056</td>
<td>6147</td>
<td>6250</td>
<td>6371</td>
<td>6534</td>
</tr>
<tr>
<td>garbage collections</td>
<td>230</td>
<td>263</td>
<td>317</td>
<td>395</td>
<td>492</td>
<td>595</td>
</tr>
</tbody>
</table>

The same evaluation was performed in the function driven system with fixed-size attribute-lists, where each function was associated with a system attribute-list of 100 entries. As the table of results provided below indicates, with a free word list of 7000 cells the system with attributes performs the same evaluation 30 times faster than the host system and the number of garbage collections required is reduced by a factor of 28. Such an improvement is mostly due to the fact that the inefficiencies in the function definition are not encountered every time since often attribute-lists, rather than definitions of functions, are used to obtain the value required.
D.8

storage in cells   7000  6000  5000  4000  3000  2000  
time in seconds   201   206   209   213   229   268  
garbage collections 8    10    12    16    24    53

D.4  Examples of translating ALGOL codes into a symbolic machine language,[25]

This is an additional example to show how the presence of temporary attributes in the function driven system with fixed-size attribute-list effects the performance of the host system.

What is involved in this translation task is best demonstrated with the help of a simple example. For instance, the result of translating the expression:

\[ F = A + B \ast C \]

might be

\[
\begin{align*}
\text{CLA C} & : \quad \text{Transfer content of \( c \) to accumulator} \\
\text{STR M} & : \quad \text{Store content of accumulator in \( M \)} \\
\text{CLA B} & : \quad \text{Transfer content of \( B \) to accumulator} \\
\text{MUL M} & : \quad \text{Multiply accumulator by content of \( M \)} \\
\text{STR Q} & : \quad \text{Store the accumulator in location \( Q \)} \\
\text{CLA A} & : \quad \text{Put content of \( A \) in the accumulator} \\
\text{ADD Q} & : \quad \text{Add content of \( Q \) to accumulator} \\
\text{STR F} & : \quad \text{Store the result in \( F \)}
\end{align*}
\]

where the translation is carried out according to the set of rules and the tables in the description referenced above.

The set of functions whose \( M \)-expressions are provided below will perform this translation task, that the final result has the form:

\[
((\text{CLA C})(\text{STR M})(\text{CLA B})(\text{MUL M})(\text{STR Q})(\text{CLA A})(\text{ADD Q})(\text{STR F}))
\]

The function \( tr \) whose \( M \)-expression definition is provided in Section (E.7) is used to translate a number of expressions into their corresponding symbolic machine language equivalents using the host system and the function driven system with fixed-size attribute-lists. In both cases the free word list contains 7000 cells. Further, in the system with attributes, \( tr \) is declared as a memo function and the size of its system attribute-list is limited to 100 entries. The table given below contains the experimental data for these two separate evaluations.

<table>
<thead>
<tr>
<th>Parameters of the systems</th>
<th>host system</th>
<th>FDS with attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>garbage collections</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>total searches</td>
<td>0</td>
<td>517</td>
</tr>
<tr>
<td>attributes available</td>
<td>0</td>
<td>312</td>
</tr>
<tr>
<td>time in seconds</td>
<td>186</td>
<td>69</td>
</tr>
</tbody>
</table>
By eliminating most of the respective evaluations involved, the function driven system performs approximately 3 times faster than the host system, and reduces the number of garbage collections required by a factor of 5.

D.5 Presburger algorithm, [10]

The Presburger decision method is used as an example to show how the presence of temporary attributes in the function driven system with dynamic attribute-lists affects the performance of the host system. The Presburger Decision method can be used in certain cases to establish the truth or falsity of decidable theorems in the area of number theory. One of the objectives in the dissertation by J. J. Boyle has been to produce a function which could be applied directly to theorems, either to establish their validity or to simplify them by eliminating quantifiers.

The following functions, whose $M$-expression definitions appear in Section (E.11), are used to carry out the evaluations:

\[
\begin{align*}
\text{PRESB}\left((\text{ALL}(X Y)(\text{EQ}(N E Q X Y)(O R(L T X Y)(L T Y X))))\right) & \\
\text{PRESB}\left((\text{ALL}(A B C)(I M P L I E S(A N D(L T A B)(L T B C))(L T A C)))\right) & \\
\text{PRESB}\left((\text{ALL}(X Y)(E X I S T(Z)(E Q U I V(A N D(L T X Z)(L T Y Z))(L T (+ X 1) Y))))\right) & \\
\text{PRESB}\left((\text{ALL}(X Y)(N E Q Y(+ x *))\right) & \\
\text{PRESB}\left((\text{ALL}(X)(L T X(+ X 1)))\right) & \\
\text{PRESB}\left((\text{ALL}(I N)(I M P L I E S(L T I N)(O R(L T(+ I 1)N)(E Q(+ I 1) N))))\right) & \\
\text{PRESB}\left((\text{ALL}(X)(E X I S T(Y)(E Q(+ Y 1) X)))) & \\
\text{PRESB}\left((\text{ALL}(X)(E X I S T(Y)(E Q Y(+ X 1))))\right) & \\
\text{PRESB}\left((\text{ALL}(X Y Z)(I M P L I E S(A N D(L T X Y)(L T Y Z))(L T X Z)))\right) & \\
\text{PRESB}\left((\text{ALL}(X Y)(E Q U I V(N L T X Y)(O R(E Q X Y)(L T Y X))))\right) & \\
\text{PRESB}\left((\text{ALL}(X Y)(E Q U I V(N E Q X Y)(O R(L T X Y)(L T Y X))))\right) & \\
\text{PRESB}\left((\text{ALL}(K I)(I M P L I E S(E Q K (+ I 1)))) & \\
\text{PRESB}\left((\text{AND}(I M P L I E S(L T 1 N)(L T 1 N))) & \\
\text{PRESB}\left((\text{ALL}(K)(I M P L I E S(A N D(O R(L T 1 K)(E Q 1 K))(O R(L T K 1)(E Q 1 K)))) & \\
\text{PRESB}\left((O R(L T 1 K)(E Q 1 K))) & \\
\end{align*}
\]

in the host system and in the function driven system with dynamic attribute-lists and a free word list consisting of 7000 cells. In the host system the above evaluations produced:

- time in seconds 88
- garbage collections 15

after declaring all the functions as memo functions and using the function driven system with dynamic attribute-lists to carry out the above evaluation, the corresponding quantities were:

- time in seconds 57
- garbage collections 8
Notice that the use of the system with attributes has reduced the
time of evaluation by 31 seconds, and the number of garbage collections
required by 7. Moreover, these gains have resulted from indiscriminately
declaring all the functions as memo functions, i.e. without making
a judicious choice among them.

D.6 Parameters of the system with dynamic attribute-lists

In this section a number of tables containing the experimental
results are used in order to show the effects that the different
values of the life-time parameters of the systems have on the time
of evaluation and the number of garbage collections. In these tests
only one parameter has been varied at a time, and the corresponding
time and the number of garbage collections required to solve the
problem have been recorded. The effects of the parameters of the
system on differentiation - 2 and prs are investigated in the function
driven system, while the corresponding effects on:

$$\frac{d^7}{dx^n} (a + (x - b)^n)^m$$

are investigated in the expression driven system. The choice of
problems used in the various systems is an arbitrary one. Due to
the cost of computation, we did not perform each of the manipulations
in both systems with dynamic attribute-lists. Each parameter of the
system is dealt with separately below.

 initial
The table:

<table>
<thead>
<tr>
<th>Values of initial</th>
<th>Differentiation - 2</th>
<th>prs</th>
<th>$\frac{d^7}{dx^n} (a + (x - b)^n)^m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time in seconds</td>
<td>time in seconds</td>
<td>time in seconds</td>
</tr>
<tr>
<td>2</td>
<td>75</td>
<td>84</td>
<td>170 10</td>
</tr>
<tr>
<td>4</td>
<td>71</td>
<td>84</td>
<td>133  5</td>
</tr>
<tr>
<td>6</td>
<td>71</td>
<td>84</td>
<td>134  8</td>
</tr>
<tr>
<td>8</td>
<td>71</td>
<td>84</td>
<td>138  8</td>
</tr>
<tr>
<td>10</td>
<td>71</td>
<td>84</td>
<td>138  8</td>
</tr>
</tbody>
</table>

shows the variation in the evaluation time and the number of garbage
collections required as a result of the different values of the
initial life-times assigned to temporary attributes when they are
created. Notice that the values:

initial =4;  increment = 2;  decrement = 1;  promote = 1

which are built into the systems, have produced a reasonable result.
When the initial value of the life-time assigned to a temporary
attribute is small, attributes which are subsequently required have
been deleted from the attribute-lists. This partly accounts for the
larger evaluation time when the value of the initial life-time is 2.
increment

The table:

<table>
<thead>
<tr>
<th>Value of increment</th>
<th>Differentiation - 2 time in seconds</th>
<th>Garbage collections</th>
<th>PRS time in seconds</th>
<th>Garbage collections</th>
<th>Time in seconds</th>
<th>Garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>71</td>
<td>84</td>
<td>0</td>
<td>133</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>72</td>
<td>84</td>
<td>0</td>
<td>138</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>72</td>
<td>84</td>
<td>0</td>
<td>138</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>72</td>
<td>84</td>
<td>0</td>
<td>138</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>72</td>
<td>84</td>
<td>0</td>
<td>138</td>
<td>7</td>
</tr>
</tbody>
</table>

shows the effects of the different increases in the life-times of attributes every time they are used, on the time of evaluation and the number of garbage collections required. According to the above table, the larger increments in the life-times of attributes has hardly produced a noticeable difference. This is because the storage space of 7000 cells is large enough to hold most of the attributes. However, in problems where attributes with positive life-times can exhaust the storage space available, the value used for increasing the life-times of attributes is significant.

promotion

The table:

<table>
<thead>
<tr>
<th>Rate of Promotion</th>
<th>Differentiation - 2 time in seconds</th>
<th>Garbage collections</th>
<th>CRS time in seconds</th>
<th>Garbage collections</th>
<th>Time in seconds</th>
<th>Garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>83</td>
<td>6</td>
<td>139</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>71</td>
<td>6</td>
<td>133</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>71</td>
<td>6</td>
<td>137</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>71</td>
<td>6</td>
<td>134</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>71</td>
<td>6</td>
<td>134</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>71</td>
<td>6</td>
<td>134</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>71</td>
<td>6</td>
<td>134</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

shows the effect of the rate of promotion of attributes on the time of evaluation and the number of garbage collections required. In comparison with the function driven system with fixed-size attribute-lists, (Section 4.7.4), this table can be used to make one point. With no promotion of the attributes used, the resulting changes in the evaluation time and the number of garbage collections required are not as large as the function driven system with fixed-size attribute-lists. This is because the rate of promotion in this system only affects the look up time, and does not play any role in the deletion of attributes from the attribute-lists.
Decrement

<table>
<thead>
<tr>
<th>Value of decrement</th>
<th>Time in seconds</th>
<th>Garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>145</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>133</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>207</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>648</td>
<td>71</td>
</tr>
</tbody>
</table>

The above table shows how the decrease in the life-time of attributes every time the garbage collection cycle is invoked affects the time of evaluation and the number of garbage collections of the evaluation:

\[
\frac{d^7}{dx^7}(ae(x-b)^n)^m
\]

Such a sharp increase in the time of evaluation and the number of garbage collections is because attributes are deleted before they are required again.

D.7 Conversion from prefix-polish notation to infix notation

This is an additional example used to compare the performance of the host system and the expression driven system with dynamic attribute-lists. The function \texttt{infix} whose M-expression definition appears in Section (E.12) is used to convert a number of reasonably long expressions from their prefix-polish notation into their corresponding infix notation. The above evaluation was carried out in the host system and in the expression driven system with dynamic attribute-lists. In both cases the free word list used consisted of 7000 entries. The table provided below contains the experimental data for these two separate evaluations:

<table>
<thead>
<tr>
<th>Systems used</th>
<th>Host system</th>
<th>EDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters of the system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Garbage collections</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>146</td>
<td>22</td>
</tr>
</tbody>
</table>

By eliminating most of the repetitive evaluations, the expression driven system performs 6.5 times faster than the host system, and reduces the number of garbage collections required by a factor of 6.
APPENDIX E

Descriptions of problems and functions

This appendix contains a brief description of all the problems referred to in this thesis. Each description is accompanied by the definition in M-expression form of the functions involved in the solution of the problem. Due to the large volume of the material involved, the S-expression definitions of the functions are not included but they are available on request from the author. Further information about these problems can be obtained by consulting the references provided.
E.1 Differentiation, [4]

This function calculates the derivative of its second argument with respect to its first argument.

driv[x; e]:

\[
\begin{align*}
a_1 & \quad [x; y] \rightarrow 0 \quad \text{when} \quad \text{np[y];} \\
a_2 & \quad [x; y] \rightarrow 0 \quad \text{when} \quad \text{free[y;x]} \\
a_3 & \quad [x; x] \rightarrow 1; \\
a_4 & \quad [x; [u; v]] \rightarrow [+\text{driv}[x; u]; \text{driv}[x; v]]; \\
a_5 & \quad [x; [u; v]] \rightarrow [-\text{driv}[x; u]; \text{driv}[x; v]]; \\
a_6 & \quad [x; [u; v]] \rightarrow [u; \text{driv}[x; u]; *v; \text{driv}[x; u]]; \\
a_7 & \quad [x; [u; v]] \rightarrow [-[y; \text{driv}[x; u]; *u; \text{driv}[x; v]]; \text{driv}[x; u]] \quad \text{when} \quad \text{free[v;x];} \\
a_8 & \quad [x; \text{exp}[u]] \rightarrow [\text{exp}[u]; \text{driv}[x; u]]; \\
a_9 & \quad [x; \text{sin}[u]] \rightarrow [*\text{cos}[u]; \text{driv}[x; u]]; \\
a_{10} & \quad [x; \text{cos}[u]] \rightarrow [*\text{sin}[u]; \text{driv}[x; u]]; \\
a_{11} & \quad \text{log}[u] \rightarrow /\text{driv}[x; u]; u]; \\
\text{last} & \quad [x; y] \rightarrow \text{list[DRV}; x; y]\]
\end{align*}
\]

multiple-differentiation

muldriv[n;x;e]:

\[
\begin{align*}
a_1 & \quad [0; x; e] \rightarrow e; \\
a_2 & \quad [1; x; e] \rightarrow \text{driv}[x; e]; \\
\text{last} & \quad [n; x; e] \rightarrow \text{muldriv}[n-1; x; \text{driv}[x; e]]. \\
\end{align*}
\]

E.2 Integration, [4]

This function calculates the integral of its second argument with respect to the first argument.

int[x; e]:

\[
\begin{align*}
i_1 & \quad [x; e] \rightarrow *[e; x] \quad \text{when} \quad \text{free[e;x]}; \\
i_2 & \quad [x; [a; b]] \rightarrow [\text{Int}[x; a]; \text{int}[x; b]]; \\
i_3 & \quad [x; [a; b]] \rightarrow -[\text{int}[x; a]; \text{int}[x; b]]; \\
i_4 & \quad [x; a] \rightarrow \# \quad \text{int}[x; a]; \\
i_5 & \quad [x; [a; b]] \rightarrow *[a; \text{int}[x; b]] \quad \text{when} \quad \text{free[a;x] with} \quad *[t_4]; \\
i_6 & \quad [x; [a; b]] \rightarrow /[[a; x; \text{int}[n; 1]] + [n; 1]] \quad \text{when} \quad \text{free[n;x] with} \quad ^\land \text{neg[n; -1] with} \quad ^\land [t_4]; \\
i_7 & \quad [x; [b; a]] \rightarrow *[\text{defact}; \text{log}[a]] \quad \text{when} \quad \text{mdriv}[x; a; b]; \\
i_8 & \quad [x; [a; b]] \rightarrow \text{Int}[x; a]; [1; b]]; \\
i_9 & \quad [x; [a; n; b]] \rightarrow *[\text{defact}; [+n; 1]]; \text{Int}[x; a]; [n; 1]]; \\
i_{10} & \quad [x; [a; n; b]] \rightarrow \text{Int}[x; a]; \text{mdriv}[x; a; b] \quad \text{with} \quad *[t_4] \land ^\land [t_9]; \\
i_{11} & \quad [x; \text{log}[x]] \rightarrow [x; -\text{log}[x]; 1]; \\
i_{12} & \quad [x; \text{sin}[x]] \rightarrow [\text{cos}[x]]; \\
i_{13} & \quad [x; \text{cos}[x]] \rightarrow \text{sin}[x]; \\
i_{14} & \quad [x; \text{tan}[x]] \rightarrow *[\text{log}[\text{cos}[x]]]; \\
i_{15} & \quad [x; \text{exp}[x]] \rightarrow \text{exp}[x]; \\
i_{16} & \quad [x; [\text{log}]; b] \rightarrow *[\text{defact}; \text{subs}[x; \text{int}[x; \text{op}[x]]; \text{op}; \text{op}] \land \text{mdriv}[x; a; b] \quad \text{with} \quad *[t_4]; [t_6]; \\
\end{align*}
\]
### E.3 Transformations [4]

These transformations are used by functions of the type RULE. Transformations $t_1$ and $t_4$ express the property of commutativity for the operators $+$ and $*$ respectively. The other transformations represent some of the axiomatic properties of operators and constants zero and one. Transformation $t_{11}$ is included because it is needed by the integration function.

$$\text{trf}[ ]:\n\begin{align*}
t_1 & +[a;b] \rightarrow +[b;a]; \\
t_2 & a \rightarrow +[a;0]; \\
t_3 & a \rightarrow +[0;a]; \\
t_4 & *[a;b] \rightarrow *[b;a]; \\
t_5 & a \rightarrow *[1;a]; \\
t_6 & a \rightarrow *[a;1]; \\
t_7 & a \rightarrow -[a;0]; \\
t_8 & a \rightarrow /[a;1]; \\
t_9 & a \rightarrow /[a;1]; \\
t_{10} & a \rightarrow [\#\#]; \\
t_{11} & [a;n] \rightarrow [a;\text{eval[quotient[n;2]]};2] \text{ when even}[n]; \\
t_{12} & *[a;*[b;c]] \rightarrow *[a;*[c;b]]; \\
t_{13} & -[a;b] \rightarrow +[\#b;a]; \\
t_{15} & [\#a] \rightarrow *[\#\#e]; \\
\end{align*}$$

### E.4 Operators [4]

The operators are $+$, $-$, $\ast$, $/$, $\uparrow$, $\#$ where the first five have their usual meaning and the last represents unary minus. Trigonometric functions sine, cosine and tangent, represented by $\sin$, $\cos$ and $\tan$, and the exponentiation and logarithm functions, represented by $\exp$ and $\log$ are also treated as operators.

Each operator ends with a dummy assertion; furthermore, in the functions $+$, $-$, $\ast$, $/$ and $\#$ the penultimate assertion tries to simplify the argument expression as far as possible. Also in each of the functions one assertion is included to simplify the argument expression when it is composed of numbers only.

In the following definitions, the functions difference, fixP, lessP, minus, minusP, numberP, plus, quotient, remainder, subP, times and zeroP are standard LISP 1.5 functions.

**+ Operator:**

$$\begin{align*}
a_1 & [/a;b;1/[c;d]] \rightarrow [/\text{times}[a;d];\text{times}[b;c];\text{times}[b;d]] \text{ when numberP } [a] \land \text{numberP } [b] \land \text{numberP } [c] \land \text{numberP } [d] \\
& \text{ with } /[t8]; \\
a_2 & [a;0] \rightarrow a \text{ when } T \text{ with } +[t7]; \\
a_3 & *[a;b;1/[c;b]] \rightarrow *[a;c;b] \text{ when } + \text{ with } *[t4;t5]; \\
a_4 & [b;\#a] \rightarrow [-b;a] \text{ when } T \text{ with } +[t1]; \\
\text{last } & [a;b] \rightarrow \text{list}[+,a;b].
\end{align*}$$
- Operator:

a1  \([a;0] \rightarrow a\);

a2  \([0;a] \rightarrow \#[a]\);

a3  \([+[a];b];+[a;c]] \rightarrow [b;c] when T with \([+[t_1];t_2]];

a4  \([/[a];b];/[c;d]] \rightarrow /[times[a;c]; times[b;d]]
    when numberP [a] \land numberP [b] \land numberP [c] \land numberP [d]
    with /[t_5]];

a5  \([a;a;b];*[c;d]] \rightarrow *[a;c];b] when T with \([*[t_4];t_5]];

a6  \([b;\#[a]] \rightarrow \#[b;a]\);

a7  \[#[a];b]] \rightarrow \#[+[a];b]];

last \([a;b]] \rightarrow list[-;a;b]]

* Operator:

a1  \([a;0] \rightarrow 0 when T with \([*[t_4]];

a2  \([a;1] \rightarrow A when T with \([*[t_4]];

a3  \([a;-1] \rightarrow \%[a] when T with \([*[t_4]];

a4  \([/[a];b];/[c;d]] \rightarrow /[times[a;c];times[b;d]]
    when numberP [a] \land numberP [b] \land numberP [c] \land numberP [d]
    with /[t_6]];

a5  \([*[a];b];+[a;c]] \rightarrow \#[a;+[b;c]] when T with \([+[t_4];t_5]];

a6  \([a;*[b;c]] \rightarrow \#[times[a;b];c] when numberP [a] \land numberP [b]
    with \([*[t_4]];

a7  \([a;*[b;c]] \rightarrow \#[b;*[a;c]] when numberP [b] \land numberP [t_4]];

a8  \([a;\#[b]] \rightarrow \#[+[a;b]] when T with \([*[t_4]];

a9  \([a;b]] \rightarrow list*[a;b;a] when numberP [b]];

last \([a;b]] \rightarrow list[*;a;b]]

/ Operators:

a1  \([a;0] \rightarrow list[UNDEFINED;A;1]];

a2  \([a;1] \rightarrow a;]

a3  \([a;b]] \rightarrow quotient[a;b] when np[a] \land np[b] \land zeroP [xremainder[a;b]];

a4  \([a;b]] \rightarrow list[/;quotient[a;hef[a;b];quotient[b;hef[b;a]];quotient[b;hefact]]when
    fixP [a] \land fixP [b];

a5  \([/[a];b];/[c;d]] \rightarrow /[times[a;c];times[b;d]]
    when numberP [c] \land numberP [b]; \land numberP [d] \land numberP [t_6];

a6  \([a;\%[b]] \rightarrow \%[/[a];b]];

a7  \([\#[a];b]] \rightarrow \#[[/[a];b]];

a8  \([*[d];\#[a];b]];*[e;\#[a;c]]]] \rightarrow \#*[d];e];\#*[a;-[b];c]]]
    when T with \([*[t_4];t_5];t_12]] \land \#*[t_9];

last \([a;b]] \rightarrow list[/;a;b]]

\ Operator:

a1  \([a;b]] \rightarrow expt[a;b] when np[a] \land np[b];

a2  \([0;0] \rightarrow [UNDEFINED;O;0];

a3  \([0;1] \rightarrow 0;]

a4  \([a;0] \rightarrow 1;]

a5  \([1;a]] \rightarrow 1;]

a6  \([a;1] \rightarrow a;]

last \([a;b]] \rightarrow list[\#;a;b]].
Operator:

\[ a \rightarrow \text{minus}[a] \quad \text{when} \quad \text{np}[a]; \]
\[ a \rightarrow [a]; \quad a \rightarrow a; \]
\[ [-b;a] \rightarrow [-a;b]; \]
\[ a \rightarrow \text{list}[#;a]. \]

expt Operator:

With the assumption that \( a \) is a number not equal to zero and \( b \) is an integer \( \text{expt} \) calculates \( a^b \).

\[
\text{expt}[a;b] = \text{prog}[i];
\]
\[ i = 1; \]
\[ \text{zeroP}[b] \rightarrow \text{return}[i]; \]
\[ i = \text{times}[i;a]; \]
\[ b = \text{sub1}[b]; \]
\[ \text{go}[L]. \]

sin Operator:

\[ a \rightarrow 0; \]
\[ a \rightarrow \text{list}[\sin;x]. \]

cos Operator:

\[ a \rightarrow 1; \]
\[ a \rightarrow \text{list}[\cos;x]. \]

tan Operator:

\[ a \rightarrow 0; \]
\[ a \rightarrow \text{list}[\tan;x]. \]

exp Operator:

\[ a \rightarrow 1; \]
\[ a \rightarrow \text{list}[\exp;x]. \]

log Operator:

\[ a \rightarrow \text{list}[\text{UNDEFINED};\log;0]; \]
\[ a \rightarrow 0; \]
\[ a \rightarrow \text{list}[\exp[x];x]; \]
\[ a \rightarrow \text{list}[\log;x]. \]

E.5 Subsidiary functions of the integrator,

\[ \text{hcf = gcd( iterative)}; \]

It evaluates the highest common factor of the two positive integers. This value is assigned to \( \text{hfact} \) and also returned as the function value.

\[
\text{hcf}[m;n] = \text{prog}[v];
\]
\[ \text{greaterP}[n;m] \rightarrow \text{go}[L1]; \]
\[ v = m; \]
\[ m = n; \]
\[ n = v; \]
\[ L1 \]
\[ v = \text{remainder}[n;m]; \]
\[ \text{zeroP}[v] \rightarrow \text{go}[L2]; \]
\[ n = m; \]
\[ m = v; \]
\[ \text{go}[L1]; \]
\[ L2 \]
\[ \text{hfact} = m; \]
\[ \text{return}[m]. \]

\[ \text{hcf( recursive)}; \]

\[ \text{hcf}[m;n] = [\text{greaterP}[n;m] \rightarrow \text{hcf}[n;m]; \text{zeroP}[\text{remainder}[n;m]] \rightarrow m; T \rightarrow \text{hcf}[\text{remainder}[n;m];m]]. \]
muldrv:
(a) M-expression definition:
(b) Description:

This function differentiates the given expression successively, the specified number of times.

\[ a_1 \rightarrow [0; v; u] \rightarrow u; \]
\[ a_2 \rightarrow [1; v; u] \rightarrow \text{drv}[v; u]; \]
\[ a_3 \rightarrow [n; v; u] \rightarrow \text{muldrv}[-n; 1; v; \text{drv}[v; u]] \text{ when } n \text{ is non-negative.} \]

mdrv:

The purpose of \text{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 \text{DFACT} as an \text{APVAL} and the value \( T \) is returned, otherwise the value is \( F \).

\[
\text{mdrv}[x; a; b] = \begin{cases} 
\text{prog}[[v]]; \\
\text{free}[v; x] \rightarrow \text{go}[L1]; \\
\text{return}[F]; \\
\text{csetq}[\text{dfact}; v]; \\
\text{return}[T].
\end{cases}
\]

\text{opp}:

This predicate function determines if its argument is one of the allowable set of operators.

\[
\text{opp}[x] = \begin{cases} 
\text{or}[[\text{eq}[x; \text{quote}[	ext{LOG}]]; \text{eq}[x; \text{quote}[	ext{EXP}]]; \\
\text{eq}[x; \text{quote}[	ext{SIN}]]; \text{eq}[x; \text{quote}[	ext{COS}]]; \\
\text{eq}[x; \text{quote}[	ext{TAN}]]].
\end{cases}
\]

\text{edge}:

The \text{edge} heuristic 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. For a thorough description of this, the reader is referred to [4].

\[
e_1 \rightarrow [x; *[h; \text{exp}[g]]] \rightarrow \text{prog}[[a]; \\
a = \text{/[h; drv[x; g]]} \\
\text{return}[-*[a; \exp[g]]; \text{int}[x; *[\text{drv}[x; a]; \exp[g]]]] \text{ when } T \text{ with } *[t4];
\]
\[
e_2 \rightarrow [x; *[h; \text{log}[g]]] \rightarrow \text{prog}[[a; b; c]; \\
c = \text{int}[x; h]; \\
a = \text{sin p[\text{list}[[0; \text{log}[g]]; c]]}; \\
c = /[[-c; a; \text{log}[g]]]; \\
h = \text{int}[x; [[a; \text{drv}[x; g]]]; g]; \\
\text{return}[[*[c; \text{log}[g]]; 2]; *[a; \text{log}[g]]]; \\
\text{b}] \text{ when } T \text{ with } *[t4];
\]
\[
e_3 \rightarrow [x; *[h; /L1; +[1; [[g; 2]]]]] \rightarrow \text{[/\text{list}[[\text{ABC}]; C]]} \text{ when } \\
\text{mdrv}[x; g]; h \text{ with } *[t1]; *[t4]; *[t1]; \\
e_4 \rightarrow [x; *[h; \text{sin}[g]]] \rightarrow \text{prog}[[a; b]; \\
a = \# /[[h; \text{drv}[x; g]]]; \\
b = \text{int}[x; \# *[\text{drv}[x; a]; \text{cos}[g]]].
\]
E.7

\[ \text{return } [+[\{a;\cos(g)\};b]] \text{ when } T \text{ with } *[t4]; \]
\[ e5 \quad [x;*[h;\cos(g)]] \rightarrow \text{progl[a; b]}; \]
\[ a=/*[h;\text{drv}[x; g]]]; \]
\[ b=\text{int}[x; *[\{\text{drv}[x; a];\text{sin}(g)];]]]; \]
\[ \text{return } [+[\{a;\text{sin}(g)\};b]] \text{ when } T \text{ with } *[t4]. \]

E.6 Recursive functions of integers.[24]

This section is included as an example of the use of recursive function definitions in building up a class of functions \( C^F \) which are computable in terms of a given base set \( F \). In the domain of the non-negative integers, \( F = \{ \text{suc}, \text{eq} \} \) where \( \text{suc} \) is the successor function and is defined by
\[ \text{suc}(n) = n + 1 \]
and where the predicate \( \text{eq} \) is defined by
\[ \text{eq}(n_1, n_2) = T \text{ if } n_1 = n_2 \]
\[ = F \text{ if } n_1 \neq n_2 \]

First definition is that of the predecessor function, \( \text{pred} \) (not defined for \( n = 0 \)). This is given by
\[ \text{pred}(n) = \text{pred}_2(n, 0) \]

where
\[ \text{pred}_2(n, m) = (m^1 = n \rightarrow m, T \rightarrow \text{pred}_2(n, m^1)) \]

Denote \( \text{pred}(n) \) by \( \overline{n} \)

Now the sum is given by
\[ m + n = (n = 0 \rightarrow m, T \rightarrow m^1 + \overline{n}) \]

the product by
\[ mn = (n = 0 \rightarrow 0, T \rightarrow m + m \overline{n}) \]

and the difference by
\[ m - n = (n = 0 \rightarrow m, T \rightarrow \overline{m} - \overline{n}) \]

which is defined for \( m \geq n \).

The inequality predicate \( m \leq n \) is defined by
\[ m \leq n = (m = 0) \lor (n = 0) \land (m \leq \overline{n}) \]

the strict inequality \( m < n \) is defined by
\[ m < n = (m \leq n) \land \neg (m = n) \]

The integer valued quotient \( m/n \) is defined by
\[ m/n = (m < n \rightarrow 0, T \rightarrow ((m - n)/n)^1) \]

The remainder or dividing \( m \) by \( n \) is defined by
\[ \text{rem}(m/n) = (m < n \rightarrow m, T \rightarrow \text{rem}((m - n)/n)) \]

The divisibility of a number \( n \) by a number \( m \) is defined by
\[ m/n = (n = 0) \lor ((n \geq m) \land (m/(n - m))) \]

The primeness of a number is defined by
\[ \text{prime} (n) = (n \neq 0) \land (n \neq 1) \land \text{prime}^2 (n, 2) \]
where
\[ \text{prime}_2(n, m) = (m = n) \lor (\sim (m/n) \land \text{prime}_2(n, m^2)) \]

The Euclidean algorithm defining the greatest common divisor is given by
\[ \gcd(m, n) = (m > n \rightarrow \gcd(n, m), \quad \text{rem}(n/m) = 0 \rightarrow m, \quad T \rightarrow \gcd(\text{rem}(n/m), m)) \]

The Euler function \( \phi \) - function (\( \phi(n) \) is the number of numbers less than \( n \) and relatively prime to \( n \)) is given by
\[ \phi(n) = \phi_2(n, n) \]

where
\[ \phi_2(n, n) = (m = 1 \rightarrow 1, \quad \gcd(n; m) = 1 \rightarrow \phi_2(n, m^2)) \]

From the above set of functions we only select those which are necessary in finding the common division of any two positive integers. The M-expression definitions of these functions are listed below.

\[
\begin{align*}
\text{pre}[n] &= \text{[pred}[2 \ n; 0]]. \\
\text{pred}_2[n; m] &= \begin{cases} \\
\text{eq}[\text{suc}[m; n]] \rightarrow m; \\
T \rightarrow \text{pred}_2[n; \text{suc}[m]]]. \\
\end{cases} \\
\text{subtract}[m; n] &= \begin{cases} \\
\text{eq}[n; 0] \rightarrow m; \\
T \rightarrow \text{subtract}[	ext{pre}[m]; 
\text{pre}[n]]]. \\
\end{cases} \\
\text{lesseq}[m; n] &= \begin{cases} \\
\text{or}[\text{eq}[m; 0]; \text{and}[\text{not}[	ext{eq}[n; 0]]; \\
\text{lesseq}[	ext{pre}[m]; 
\text{pre}[n]]]]. \\
\end{cases} \\
\text{less}[m; n] &= \begin{cases} \\
\text{and}[	ext{lesseq}[m; n]; \text{not}[	ext{eq}[m; n]]]]. \\
\end{cases} \\
\text{gcd}[m; n] &= \begin{cases} \\
\text{less}[m; n] \rightarrow \text{gcd}[n; m]; \\
\text{eq}[	ext{rem}[n; m] 0] \rightarrow m; \\
T \rightarrow \text{gcd}[	ext{rem}[n; m]; m]]. \\
\end{cases} \\
\text{rem}[m; n] &= \begin{cases} \\
\text{less}[m; n] \rightarrow m; \\
T \rightarrow \text{rem}[	ext{subtract}[m; n]; n]]. \\
\end{cases}
\]

These functions are used in an evaluation carried out in the host system and function driven system with fixed-size attribute-lists, (Section D.3).

E.7 Examples of translating ALGOL codes into a symbolic machine language, [25]

Tr:

This function translates the assignment statement of ALGOL, as the source language into its symbolic machine language, as the target language.

\[ \text{I succ}[n] = \text{[add1 [n]]}. \]
E.9

It returns the level number of the given argument expression.

(b) M-expression definition:
\[ \text{llno}[x] = \begin{cases} 1 \quad & \text{or} \left[ \text{atom}[x]; \text{null} \left[ \text{cdr}[x] \right] \right] \rightarrow 10; \\
& \text{or} \left[ \text{eq} \left[ \text{cadr}[x]; + \right]; \text{eq} \left[ \text{cadr}[x]; - \right] \right] \rightarrow 2; \\
& T \rightarrow 3. \end{cases} \]

cla:

This function forms a list with its first element \text{CLA} and the second element the original argument expression.

\[ \text{last}[a] \rightarrow \text{list}[\text{CLA}; a]. \]

str:

This function forms a list with its first element as \text{STR} and the second element the original argument expression.

(b) M-expression definition:
\[ \text{last}[a] \rightarrow \text{list}[\text{STR}; a]. \]

E.8 Prime numbers:\[24\]

Given a positive integer \( n \), \text{pr} produces a list of all the prime numbers less than \( n \).

iterative definition:
\[
\text{pro}[\text{n}] = \begin{cases} \text{prog}[\text{n}] & \text{A} \\
\text{zeroP \[n\]} & \rightarrow \text{return}[\text{n}] \\
\text{primeP}[\text{n}] & \rightarrow \text{set2}[\text{n}; \text{cons}(\text{n}; \text{u})] \\
\text{set2}[\text{n}; \text{sub1}[\text{n}]] \\
\text{go}[\text{A}]. \end{cases}
\]
recursive definition:

\[ \text{prs}[n] = [\]
\[ \text{zeroP}[n] \rightarrow \text{NIL}; \]
\[ \text{prime}[n] \rightarrow \text{cons}[n; \text{prs}[\text{subi}[n]]]; \]
\[ T \rightarrow \text{prs}[\text{subi}[n]]. \]

\text{prime}[n]:

Given a positive integer \( n \), the functions \text{prime} and \text{prime2} determine whether the given number is a prime number. The function \text{prime} returns the value \( T \) if \( n \) is a prime number, otherwise \text{NIL} is returned as value.

\[ \text{prime}[n] = [\]
\[ \text{lessP}[n;2] \rightarrow \text{NIL}; \]
\[ \text{eq}[n;2] \rightarrow T; \]
\[ \text{zeroP}[\text{remainder}[n;2]] \rightarrow \text{NIL}; \]
\[ T \rightarrow \text{prime2}[n;3]. \]

\[ \text{prime2}[n;m] = [\]
\[ \text{greaterP}[\text{times}[m;m];n] \rightarrow T; \]
\[ \text{zeroP}[\text{remainder}[n;m]] \rightarrow \text{NIL}; \]
\[ T \rightarrow \text{prime2}[n;\text{plus}[m;2]]. \]

\text{divisors}[n] = [\]
\[ \text{map}[\text{product}[\text{gf}[\text{pf}[n]]]]. \]
\[ \text{map}[x] = [\text{null}[x] \rightarrow \text{NIL}; \]
\[ T \rightarrow \text{cons}[\text{arb}[\text{car}[x]]; \text{map}[\text{cdr}[x]]]. \]
\[ \text{product}[x] = [\text{null}[x] \rightarrow \text{list}[\text{NIL}]; \]
\[ T \rightarrow \text{f1}[\text{car}[x]; \text{product}[\text{cdr}[x]]]]. \]
\[ \text{gf}[x] = [\text{null}[x] \rightarrow \text{list}[1]; \]
\[ T \rightarrow \text{gf1}[\text{cdr}[x]; \text{list}[1]; \text{NIL}; x]]. \]
\[ \text{pf}[x] = [\text{zeroP}[\text{remainder}[x;2]] \rightarrow \text{cons}[2]; \]
\[ \text{pf}[\text{quotient}[x;2]]]; \]
\[ T \rightarrow \text{pf1}[x;3]]. \]
\[ \text{pf1}[x;y] = [\text{eq}[x;1] \rightarrow \text{NIL}; \]
\[ \text{lessP}[x;\text{times}[y;y]] \rightarrow \text{cons}[x; \text{NIL}]; \]
\[ \text{zeroP}[\text{remainder}[x;y]] \rightarrow \text{cons}[y; \text{pf1}[\text{quotient}[x;y]]; y]]; \]
\[ T \rightarrow \text{pf1}[x;\text{plus}[y;2]]. \]
\[ \text{gf1}[x;y;z;w] = [\text{null}[w] \rightarrow \text{cons}[y;z]; \]
\[ \text{eq}[\text{car}[w]; x] \rightarrow \text{gf1}[x; \text{cons}[\text{times}[x; \text{car}[y]]; y]; z; \text{cdr}[w]]; \]
\[ T \rightarrow \text{gf1}[\text{car}[w]; \text{list}[1]; \text{cons}[y; z]; w]]. \]
\[ \text{f1}[x;y] = [\text{null}[x] \rightarrow \text{NIL}; \]
\[ T \rightarrow \text{f2}[\text{car}[x]; y; \text{f1}[\text{cdr}[x]]; y]]. \]
\[ \text{f2}[x;y;z] = [\text{null}[y] \rightarrow z; \]
\[ T \rightarrow \text{cons}[[\text{cons}[x; \text{car}[y]]; \text{f2}[x; \text{cdr}[y]]; z]]]. \]
\[ \text{arb}[x] = [\text{lit}[1; x]]. \]
\[ \text{lit}[y;z] = [\text{null}[z] \rightarrow y; \]
\[ T \rightarrow \text{g}[\text{car}[z]; \text{lit}[y; \text{cdr}[z]]]]. \]
\[ g[x;y] = [\text{times}[x;y]]. \]
E.10 Vector analysis, \[27\]

The \( M \)-expression definitions of the functions required for the manipulations of Section (D.1) are listed below. The functions \( d1x, d2x, \ldots d3z \) are deliberately constructed in terms of \( drv \) and \( muldrv \), (Section E.1) so that to create a situation where a number of marked function are being applied to the same expressions. As the following \( M \)-expression definitions of the functions above show, the same result can be achieved by supplying the appropriate arguments to the functions \( drv \) and \( muldrv \).

\[
d1x[u] = [drv[quote[x]];u]].
\]

\[
d1y[u] = [drv[quote[y]];u]].
\]

\[
d1z[u] = [drv[quote[z]];u]].
\]

\[
d2x[u] = [muldrv[2;quote[x]];u]].
\]

\[
d2y[u] = [muldrv[2;quote[y]];u]].
\]

\[
d2z[u] = [muldrv[2;quote[z]];u]].
\]

\[
d3x[u] = [muldrv[3;quote[x]];u]].
\]

\[
d3y[u] = [muldrv[3;quote[y]];u]].
\]

\[
d3z[u] = [muldrv[3;quote[z]];u]].
\]

\[
curl[x;y;z] = [+[d1y[z];d1z[y]];quote[I]]; +*[+[d1z[x];d1x[z]];quote[J]]; *[+[d1x[y];d1y[z]];quote[K]]]].
\]

\[
curl curl[x;y;z] = [+[[-d1y[d1x[y]];+[d2y[x];d1z[d1x[z]]]];quote[I]]; +*[[-d1z[d1y[z]];[-d2z[y];+d2x[y]]; +[d1x[d1y[x]]]];quote[J]]; +*[[-d1x[d1z[x]];[-d2x[z];+d2y[z]]; d1y[d1z[y]]]];quote[k]]]].
\]

\[
curl curl curl[x;y;z] = [+[[-d1y[d2x[z]];[-d3y[z];+d1z[d2y[y]]]]; +[d1y[d2z[z]]]; quote[I]]; +*[[-d1z[d3y[z]];+d1x[d2z[y]]]];quote[J]]; +*[[-d1x[d2z[z]];[-d3z[x];+d1y[d3y[y]]]]; +[d1x[d2y[y]];+d1y[d2z[y]]]];quote[k]]]].
\]

E.11 Presburger algorithm, \[12\]

append[x;y] = [null[x] -> y; T -> cons[car[x];append[cdr[x];y]]].

linor[l] = [atom[l] -> list[l];
  eq[car[l]; OR ] -> append[linor[car[l]]; linor[car[cdr[l]]]];]
  T -> list[l]].

printlist[l] = [prog[u];
  u:=1;
  LO
  null[u] -> return[NIL];
  print[car[u];
  u:=cdr[u];
  go[LO].]
printlincor[1] = [prog[u];
    u := lincor[1];
    printlincor[u];
    return[u].

reduce[x]:

r0  [x1 -> x when atom[x];

r1 [[NOT;x]] -> rnot[x];

r2 [[OR;x;y]] -> ror [reduce[x];reduce[y]];  
r3 [[AND;x;y]] -> rand[reduce[x];reduce[y]];  
r4 [[IMPLIES;x;y]] -> ror[rnot[x];reduce[y]];  
r5 [[EQUIV;x;y]] -> ror[rand[reduce[x];reduce[y]];  
                rand[rnot[x];rnot[y]]];

r6 [[NEQ;x;y]] -> ror[list[LT;x;y];list[y;x]];  
r7 [[NLT;x;y]] -> list[LT[y;+[x;1]]];

last [x] -> x.

rnot[x]:

a1 [[NOT;x]] -> x;
a2 [[EQUIV;x;y]] -> ror[rand[rnot[x];reduce[y]];  
                rand[rnot[y];reduce[x]]];
a3 [[IMPLIES;x;y]] -> rand[reduce[x];rnot[y]];  
a4 [[OR;x;y]] -> rand[rnot[x];rnot[y]];  
a5 [[AND;x;y]] -> ror[rnot[x];rnot[y]];  
a6 [[EQ;x;y]] -> ror[list[LT;x;y];list[y;x]];  
a7 [[LT;x;y]] -> list[LT[y;+[x;1]]];  
a8 [[NEQ;x;y]] -> list[LT[x;y]];  
a9 [[NLT;x;y]] -> list[NLT;x;y];

last [x] -> list[NOT;x].

ror[x]:

r0 [x;x] -> x;

last [x;y] -> list[OR;x;y].

rand[x]:

r0 [x;x] -> x;
r1 [x;[OR;y;z]] -> ror[reduce[x];rand[y];rand[z]] with rand[com];

last [x;y] -> list[AND;x;y].

lt[x]:

10 [x;y] -> lessp[x;y] when np[x] \& np[y]

last [x;y] -> list[LT;x;y].

Transformations:

com rand[x;y] -> rand[y;x];
t1a eq[x;y] -> eq[y;x];
t1 +[x;y] -> +[y;x];
t2 x -> +[x;0].
presb[x]:

\[ \text{p0} \quad \text{[[\text{EXIST};x;y]]} \rightarrow \text{presb[[\text{EXIST};x;\text{presb}[y]]]}; \]

\[ \text{p1} \quad \text{[[\text{ALL};x;y]]} \rightarrow \text{[prog[u];} \]

\[ \text{u;} \rightarrow \text{presb[[\text{EXIST};x;\text{list}[\text{NOT};\text{presb}[y]]]]}; \]

\[ \text{eq}[u;T] \rightarrow \text{return[NIL]}; \]

\[ \text{eq}[u;\text{NIL}] \rightarrow \text{return[T];} \]

\[ \text{return[[\text{list}[\text{NOT};u]]]}; \]

\[ \text{p2} \quad \text{[[\text{op};x]]} \rightarrow \text{list[[\text{op};\text{presb}[x]]]} \text{ when genop[[\text{op}}]; \]

\[ \text{p3} \quad \text{[[\text{op};x;y]]} \rightarrow \text{list[[\text{op};\text{presb}[x];\text{presb}[y]]]} \text{ when genop[\text{op}}]; \]

\[ \text{last} \quad \text{[x]} \rightarrow \text{x when atom[x].} \]

\[ \text{presb1[x;y;z]} = (\]

\[ \text{eliminate[[\text{linor};\text{reduce}[\text{z};\text{y}}]]} . \]

\[ \text{genop[x]} = [\]

\[ \text{not[\text{or[[\text{eq}[x;\text{ALL}];\text{eq}[x;\text{EXIST}}]))].} \]

\([r]:\]

\[ \text{a1} \quad \text{[[x;y]]} \rightarrow \text{plus[[x;y]} \text{ when np[x] \text{\&\& np[y};} \]

\[ \text{a2} \quad \text{[[+;x;y];z]} \rightarrow \text{+[x;plus[y;z]] \text{ when np[y] \text{\&\& np[z};} \]

\[ \text{last} \quad \text{[x;y]} \rightarrow \text{list[[+;x;y}]. \]

\[ \text{unbindcomb[r]}: \]

\[ \text{a1} \quad \text{[[x;\text{LT};[[+;u;m]];[[+;x;n]]];[[+;x;p];[[+;v;q]]]]} \rightarrow \text{lt[[+;u;\text{plus}[m;p;1]];[[+;v;\text{plus}[n;q]]]} \text{ when np[m] \text{\&\& np[n] \text{\&\& np[p] \text{\&\& np[q}}] \text{ with } +[t1;t2];} \]

\[ \text{last} \quad \text{[[x;\text{LT};[[+;x;p];[[+;v;q]]];[[+;u;m];[[+;x;n]]]} \rightarrow \text{lt[[+;u;\text{plus}[m;p;1]];[[+;v;\text{plus}[n;q]]]} \text{ when np[m] \text{\&\& np[n] \text{\&\& np[p] \& np[q]} with } +[t1;t2];} \]

\[ \text{discrim[r]}: \]

\[ \text{d1} \quad \text{[[x;\text{EQ};[[+;x;p];[[+;x;q]]]]} \rightarrow \text{eq[p;q] \text{ when np[p] \& np[q with } +[t1;t2];} \]

\[ \text{d2} \quad \text{[[x;\text{LT};[[+;x;p];[[+;x;q]]]]} \rightarrow \text{lessp[p;q] \text{ when np[p] \& np[q with } +[t1;t2];} \]

\[ \text{d3} \quad \text{[[x;\text{LT};[[+;x;p];[[+;y;q]]]]} \rightarrow \text{list[[+;y;\text{difference}[q;p]]] \text{ when np[p] \& np[q with } +[t1;t2]\& eq[t1a]} \]

\[ \text{d4} \quad \text{[[x;\text{LT};[[+;x;p];[[+;y;q]]]]} \rightarrow \text{0 \text{ when np[p] \& np[q with } +[t1;t2];} \]

\[ \text{d5} \quad \text{[[x;\text{LT};[[+;y;q];[[+;x;p]]]]} \rightarrow \text{1 \text{ when np[p] \& np[q with } +[t1;t2];} \]

\[ \text{last} \quad \text{[x;a]} \rightarrow \text{2.} \]

\[ \text{sbst[r]}: \]

\[ \text{s1} \quad \text{[[x;y;y]]} \rightarrow \text{x;} \]

\[ \text{s2} \quad \text{[[x;y;[+;y;n]] \rightarrow +[x;n};] \]

\[ \text{s3} \quad \text{[[x;y;z]]} \rightarrow \text{z when atom[z];} \]

\[ \text{last} \quad \text{[[x;y;[z1;z2]] \rightarrow cons[sbst[[x;y;z1];sbst[[x;y;z2];]} \]
delinand[1] = [eq[car[1];NIL] —> list[OR;car[1];cadr[1]]; T —> list[AND;car[1];delinand[cadr[1]]]].

delinor[1] = [eq[cdr[1];NIL] —> list[OR;car[1];cadr[1]]; T —> list[OR;car[1];delinor[cadr[1]]]].

unbind[2;x] = [eq[2;T] —> T];
eq[2;NIL] —> NIL;

cadr[eq[2;EQ];eq[2;LT]] —> unbindatom[2;x];

delete[2;x;2] = [null[2;NIL];
eq[2;car[2]] —> cdr[2];
T —> cons[car[2];delete[2;x;cdr[2]]]].

linand[1] = [atom[1] —> list[1];
eq[car[1];AND] —> append[linand[cadr[1]]];
linand[car[cadr[1]]]]; T —> list[1]).

unbindvars[2;2] = [prog[2;2];

r;=;c;
u;=;v;

LOOP

null[u] —> return[r];
r;=;unbind[r;car[u]]; u;=;cdr[u];
go[LOOP].

Unbindatom[2;2]:

\[
\begin{align*}
a1 \quad [x;[EQ;[+;x;p];[+;x;q]]] & \rightarrow eq[p;q] \text{ when } np[p] \land np[q] \\
& \quad \text{with } [+;t1;t2]; \\
a2 \quad [x;[LT;[+;x;p];[+;x;q]]] & \rightarrow lessp[p;q] \text{ when } np[p] \land np[q] \\
& \quad \text{with } [+;t1;t2]; \\
a3 \quad [x;[EQ;[+;x;p];[+;y;q]]] & \rightarrow T \text{ when } np[p] \land np[q] \\
& \quad \text{with } [+;t1;t2]; \\
a4 \quad [x;[LT;[+;x;p];[+;y;q]]] & \rightarrow T \text{ when } np[p] \land np[q] \\
& \quad \text{with } [+;t1;t2]; \\
a5 \quad [x;[LT;[+;y;q];[+;x;p]]] & \rightarrow T \text{ when } np[p] \land np[q] \\
& \quad \text{with } [+;t1;t2];
\end{align*}
\]

\text{last } [x;a] —> a .
eliminate[1;2] = [prog[2;2];

u;=;1;

LOOP

null[u] —> return[NIL];
x;=;unbindvars[car[u];2];
u;=;cdr[u];
eq[x;T] —> return[x];
neq[x;NIL] —> v;=;cons[x;2];

null[u] —> go[LABE];
go[LOOP];

LABEL

null[v] —> return[NIL];
null[cadr[v]] —> return[car[v]]; 
return[delinor[v]]].

unbindlist[1;2] = [prog[u;v;s;xleft;xright;xfree];
u;=;1;]
E.15

LOOP1 null[u] → go[LABEL1];
s:=car[u];
u:=cdr[u];
v:=discrim[x;s];
eq[v;NIL] → return[NIL];
eq[v;T] → go[LOOP1];
eq[v;0] → xleft:=cons[s;xleft];
eq[v;1] → xright:=cons[x;xright];
eq[v;2] → xfree:=cons[s;xfree];
atom[v] → go[LOOP1];
v:=sbst[v;x;delete[s;]];null[cdrtv] ■ — > return[car Lvl];
return[delinand[v]];LABEL1
null[xleft] → go[LABEL4];
null[xright] → go[LABEL4];
x:=xright;
v:=xleft;
LOOP2 s:=unbindcomb[x;car[u];car[v]];
eq[s;NIL] → return[NIL];
eq[s;T] → go[LABEL2];
xfree:=cons[s;xfree];
LABEL2 u:=cdr[u];
null[u] → go[LABEL3];
go[LOOP2];
LABEL3 v:=cdr[v];
null[v] → go[LABEL4];
u:=xright;
go[LOOP2];
LABEL4 null[xfree] → return[T];
null[cdrtxfree] → return[car[xfree]];return[delinand[xfree]]).

E.12 Prefix-polish to Infix

Given an expression in prefix-polish notation, the function whose M-expression is provided below can be used to convert it to infix notation.

infix:

i0  [a] → a when atom[a];
i1  [+[a;b]] → list[a;+;b] when atom[a] ∧ atom[b];
i2  [-[a;b]] → list[infix[a];-;infix[b]]; 
i3  [-[a;b]] → list[a;-;b] when atom[a] ∧ atom[b];
i4  [-[a;b]] → list[infix[a];-;infix[b]]; 
i5  [*[a;b]] → list[a;*;b] when atom[a] ∧ atom[b];
i6  [*[a;b]] → list[infix[a];*;infix[b]]; 
i7  [[a;b]] → list[a;/;b] when atom[a]∧ atom[b];
i8  [[a;b]] → list[infix[a];/;infix[b]]; 
i9  [[a;b]] → list[a;@;b] when atom[a]∧ atom[b];
i10  [[a;b]] → list[infix[a];@;infix[b]]; 
i11  [[a]] → list[[H,a] when atom[a];
i12  [[# [a]]] → list[#;infix[a]]; 
i13  [log[a]] → list[log;a] when atom[a];
i14  [log[a]] → list[log;infix[a]]; 
i15  [exp[a]] → list[exp;a] when atom[a];
last [exp[a]] → list[exp;infix[a]].
E.13 Determinants

\[ \text{det2} \]
\[ \text{det2}[x;y] = [ -[*\text{car}[x];\text{cdr}[y]];[*\text{cdr}[x];\text{car}[y]]]]. \]

\[ \text{det3} \]
\[ \text{det3}[x;y;z] = [ +[*\text{car}[x];\text{det2}[\text{cdr}[y];\text{cdr}[z]]]; -[*\text{car}[y];\text{det2}[\text{cdr}[x];\text{cdr}[z]]]; [*\text{car}[z];\text{det2}[\text{cdr}[x];\text{cdr}[y]]]]. \]

\[ \text{det4} \]
\[ \text{det4}[x;y;z;k] = [ +[*\text{car}[x];\text{det3}[\text{car}[y];\text{cdr}[z];\text{cdr}[k]]]; -[*\text{car}[y];\text{det3}[\text{cdr}[x];\text{cdr}[z];\text{cdr}[k]]]; +[*\text{car}[z];\text{det3}[\text{cdr}[x];\text{cdr}[y];\text{cdr}[k]]]; [[[*\text{car}[k];\text{det3}[\text{cdr}[x];\text{cdr}[y];\text{cdr}[z]]]]. \]

\[ \text{det5} \]
\[ \text{det5}[x;y;z;k;l] = [ +[*\text{car}[x];\text{det4}[\text{car}[y];\text{cdr}[z];\text{cdr}[k];\text{cdr}[l]]]; -[*\text{car}[y];\text{det4}[\text{cdr}[x];\text{cdr}[z];\text{cdr}[k];\text{cdr}[l]]]; +[*\text{car}[z];\text{det4}[\text{cdr}[x];\text{cdr}[y];\text{cdr}[k];\text{cdr}[l]]]; -[*\text{car}[k];\text{det4}[\text{cdr}[x];\text{cdr}[y];\text{cdr}[z];\text{cdr}[l]]]; [*\text{car}[l];\text{det4}[\text{cdr}[x];\text{cdr}[y];\text{cdr}[z];\text{cdr}[k]]]]. \]
\((\pi A (\pi B C))\)
\((\pi A (\pi C B))\)
\((\pi A (\pi B))\)
\((\pi A B)\)
\((\pi B (\pi A C))\)