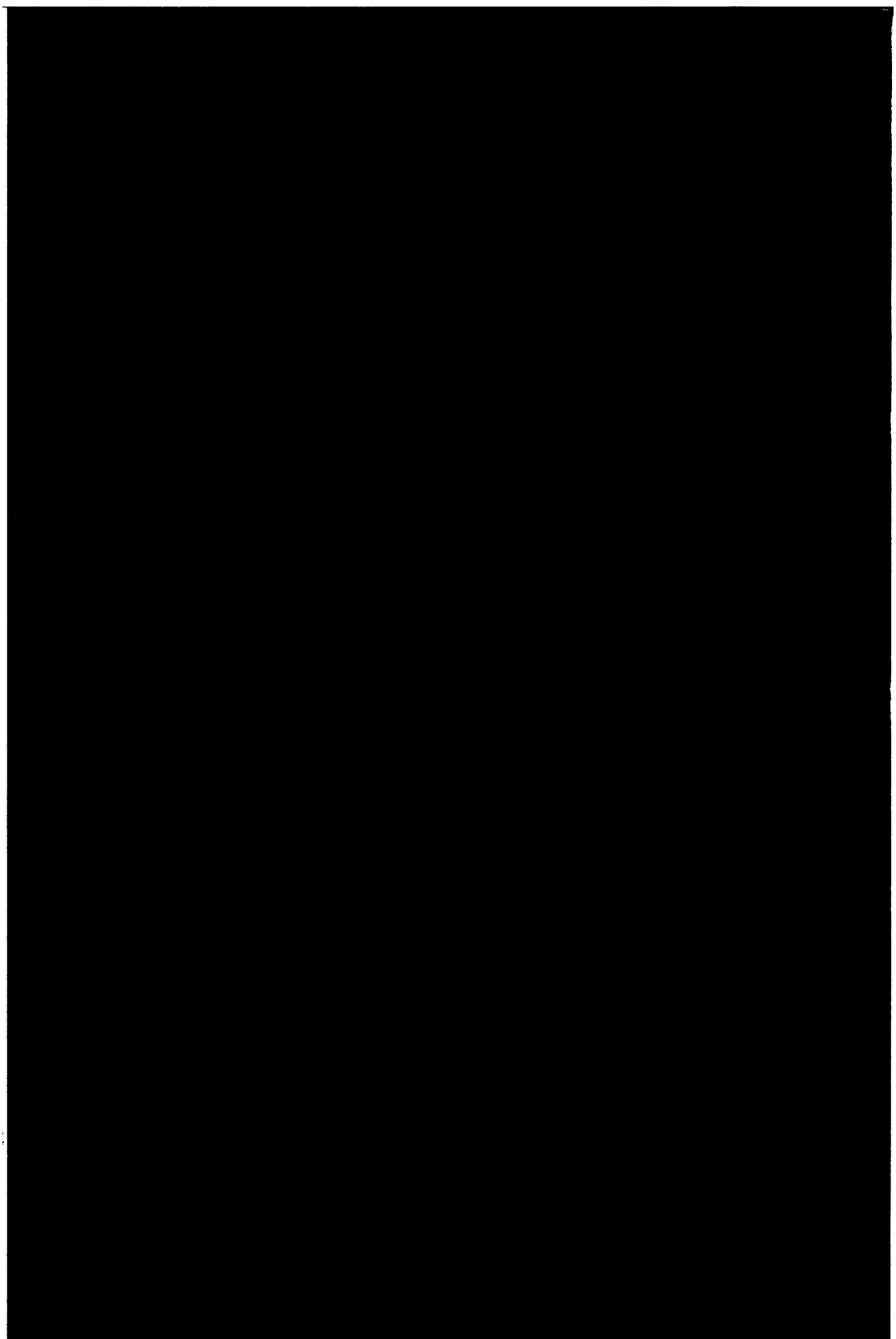

This blank page was inserted to preserve pagination.

An On-Line System for Algebraic Manipulation

A thesis presented

by

Robert Ross Fenichel

to

The Division of Engineering and Applied Physics

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Applied Mathematics

Harvard University

Cambridge, Massachusetts

July, 1966

Copyright reserved by the author

This empty page was substituted for a blank page in the original document.

PREFACE

This thesis began as a series of vague, groping memoranda. The computer program at the center of this work was operational -- in an early form -- long before it was clearly understood.

That a thesis ever condensed out of such vapor is largely to the credit of T. E. Cheatham, Jr., C. Christensen, Michael J. Fischer, Anatol W. Holt, Joel Moses, Anthony G. Oettinger, and Kirk Sattley. These men served as filters for a body of material which by volume, if not by weight, far outstripped the present thesis.

During the final throes, many of my friends had the good sense to avoid me. They, too, made an inestimable contribution.

This thesis could not have been produced had I not had access to the computational facilities of Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Richard G. Mills, Assistant Director of the project, was particularly cooperative in alloting great quantities of unique -- and therefore priceless -- resources. This thesis, having had this support from MAC, may be reproduced in whole or in part for any purpose of the United States Government.

Any felicities of style which this thesis may have are due to the patient and incisive attention of Professor Oettinger. His cheerful service far exceeded the call of his duty as my advisor.

The original thesis was prepared on the Compatible Time-Sharing System at MAC. It was expertly typed into that system by Miss Eileen Gannon. While preparing the manuscript, I received important advice from Jerome H. Saltzer and J. Anthony Gunn. Preparation of the manuscript was supported in part by Project TACT, a Harvard research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Contract Number SD-265.

SYNOPSIS

This thesis describes an approach to the problem of programming a computer for algebraic manipulation. The motivating threads of the work are first picked up in Chapter I.

To test the descriptive intuitions urged normatively in Chapter I, an experimental system was actually implemented. This system is described in Chapter II and in the Appendices.

The system was variously exercised, as reported in Chapters III, IV, and V. In addition to certain examples, Chapter III includes a more speculative discussion of the range of the system.

The exercises chosen for Chapters IV and V (algebraic "simplification" and "limit problems," respectively) proved to be worthy of some discussion not related to the system under test.

Finally, Chapter VI is a mass of hindsight, reconsideration, and evaluation. On the basis of the experience described in preceding chapters, future directions of work are suggested.

TABLE OF CONTENTS

Chapter		Page
PREFACE		iii
SYNOPSIS		iv
I	INTRODUCTION	1
II	SYSTEM DESCRIPTION	
	2.1 Data Base and Commands	5
	2.2 Expressions	5
	2.3 Application of Rules	6
	2.3.1 Structure and Utility of a Single Rule: Gross Description	6
	2.3.2 Structure and Utility of a Single Rule: Detailed Description	7
	2.3.3 Organization of Rules	11
	2.3.4 Manipulation of the EUC	12
	2.3.5 Quotations	14
	2.4 Function Definitions	16
III	POWER OF THE SYSTEM	
	3.1 Abstract Automata	17
	3.2 Modeling and the Wang Algorithm	20
	3.3 Semantics and Ω-Theory	25
IV	ALGEBRAIC SIMPLIFICATION	
	4.1 An Essay on Simplification	31
	4.1.1 The Measure of Simplicity	31
	4.1.2 The Context of Simplicity	34
	4.2 Three Essays at Simplification	36
	4.2.1 Wooldridge-Russell Simplify (WRS)	36
	4.2.2 AUTSIM	40
	4.2.3 An Exercise	44
V	LIMIT PROBLEMS	
	5.1 General Discussion	51
	5.2 A FAMOUS System for Two-Sided Limit Problems	55
VI	LOOKING AHEAD	
	6.1 Matching	65
	6.2 Assertions and Rules	68
	6.3 Efficiency	69
APPENDICES		
A	COMMAND DESCRIPTIONS	75
	A.1 Commands Concerned with Expressions	77
	A.1.1 Consider	77
	A.1.2 Save	77
	A.1.3 Reconsider	78
	A.1.4 Reset	78

TABLE OF CONTENTS

Appendix			Page
A	A. 2	Commands Dealing with Rules	79
		A. 2. 1 Reset	79
		A. 2. 2 Save	79
		A. 2. 3 Retrieve	79
		A. 2. 4 Suppress	80
		A. 2. 5 Scan	80
		A. 2. 6 Abbreviate	81
		A. 2. 7 Evaluate	82
		A. 2. 8 Expand	82
		A. 2. 9 Leave	83
		A. 2. 10 Replace	84
		A. 2. 11 Now	84
		A. 2. 12 Addition of Rules	85
	A. 3	Commands Concerned with Function Definition	86
		A. 3. 1 Assert	86
		A. 3. 2 The Inner Assert Facility	87
		A. 3. 3 Save	88
		A. 3. 4 Reassert	88
	A. 4	Miscellaneous Commands	90
		A. 4. 1 Continue	90
		A. 4. 2 Desk	90
		A. 4. 3 Hold	91
		A. 4. 4 Listen	91
		A. 4. 5 Quit	91
B		THE FAMOUS EVALUATION ROUTINE	93
C		FUNCTIONS	
	C. 1	Functions Which Must be Defined by the User	95
	C. 2	Numerical Predicates	96
	C. 3	Numerical Functions	96
	C. 4	Expression-Handling Predicates	97
	C. 5	Expression-Handling Functions	98
	C. 6	Miscellaneous Predicates	98
	C. 7	Miscellaneous Functions	99
D		CONSOLE INPUT - OUTPUT	
	D. 1	Input Side	101
		D. 1. 1 Rdline	101
		D. 1. 2 Clean	103
		D. 1. 3 Floydpolish	103
		D. 1. 4 Rephrase	104
	D. 2	External Representations	105
	D. 3	Output Side	106
E		OPERATING CONSIDERATIONS	
	E. 1	Error Procedures	107
		E. 1. 1 Mistyped Commands or Responses	107
		E. 1. 2 Host System Errors	107
		E. 1. 3 Interrupt Signals	107
	E. 2	Scanning	108
	E. 3	Command Levels	108
BIBLIOGRAPHY			109

CHAPTER I

INTRODUCTION

This thesis is concerned with a program for on-line algebraic manipulation. The program, called FAMOUS (my Algebraic Manipulator for On-line USe), is thoroughly described in Chapter II. The presentation in Chapter II is complete, but it has rather a cookbook tone. Chapter III is a more reflective attempt to define the power and nature of the system. Algebraic "simplification" has been a benchmark of algebraic manipulators, and it is discussed in Chapter IV. A more novel application, that of limit problems is discussed in Chapter V. Finally, Chapter VI consists of miscellaneous remarks about possible and impossible lines of further work.

FAMOUS is an on-line system for the manipulation of linguistic forms. Although these forms can have quite arbitrary interpretations, the standard interpretation will be that they are algebraic expressions. The system is prejudiced toward algebraic expressions by the following design features:

(1) FAMOUS expects that some expressions will be subject to evaluation. The system includes an evaluation operator and subroutines for many common mathematical functions. In addition, the system includes facilities which allow the user to define new functions and to redefine old ones.

(2) FAMOUS expects that unspecified, but trivial, syntactic transformations will be able to take external expressions to and from compositions whose phrases are

- (A) atomic expressions, such as '3' and 'x', and
- (B) lists, each consisting of
  - (i) a function name, such as '+' or 'sin',
  - (ii) phrases, fixed in number by the function name, which are arguments of the named function.

(3) Some problems associated with Abelian-group operators were not understood when the present FAMOUS system was implemented. (See § 6.1.) These operators are, for example, seriously cramped when each is permitted only a fixed number of arguments.

Toward the end of getting a system on the air, and in the belief that one insightless scheme would be as good as any other, FAMOUS was given special ad hoc preparation for dealing with the Abelian-group operators.

(4) Throughout, I have been forced to make programmer's judgments of the relative likelihoods of various pairs of user actions. These judgments reflect my feeling for algebraic manipulation.

On the other hand, a restriction to algebraic expressions is not necessarily much of a constraint. FAMOUS allows its "algebraic expressions" to include arbitrary functions which may or may not be defined. In this way, regular non-algebraic constructions may be concealed as arguments of ad hoc functions.

Rules of local change are the heart of FAMOUS. These rules are all supplied by the user: FAMOUS does not know that x+ 0 goes to x any more than it knows that v goes to ds/dt or that v goes to lwh.

Using these rules, FAMOUS looks at an algebraic manipulation as a series of local changes. Localness (or, more euphoniously, proximity) is not easy to define. It is not merely typographical, since the "x" and "y" of

$$x + y + z$$

are no more local to each other than the "x" and "z".

The centrality of proximity in FAMOUS was originally prompted by the austere elegance of Ω-theory [11, 30], which might, indeed, be called the study of proximity. While FAMOUS was being developed, however, a number of remarkable and perversely encouraging results were announced by Daniel Richardson [20].

It seems that all of the interesting non-local properties of algebraic expressions are recursively undecidable. Richardson explicitly proves that "identical to zero" is undecidable, and the other properties (e.g., "linear

in 'x'") are derivatively murky.* To those with a Turing-Church ontology, only the local properties (e.g., "literally 'x+2'", "of the form 'x-x'", etc.) are real.

The linguistic notions of Quine [17, 18, 19] have shaped FAMOUS in several different ways. The distinction between use and mention [18, § 4], for example, is maintained with some care throughout.

The most striking Quinian notions taken up by FAMOUS are those of referential transparency and opacity [19, § 30]. Because FAMOUS always stands ready to evaluate the expressions with which it deals, referential transparency is the rule. But even "algebraic expressions" occasionally include opaque functions. The definite integral operator is possibly not opaque, since one might reconstruct

$$\int_a^b x^2 dx \tag{1}$$

as

$$\text{definiteintegral}(a, b, \lambda x.\, x^2) \tag{2}$$

so that "x" is no longer an apparent argument of the integration. But now the inescapably opaque λ is introduced.

There would be some elegance in the restriction to a single opaque function. Nevertheless, FAMOUS will do its best for the user who introduces forms like (1). The precision of (2) is bought at the price of some naturalness, and the system can cope with a growing set of opaque functions as easily as it could cope with a single such function.

One other feature of FAMOUS should be mentioned in this introduction. It has been mentioned that FAMOUS stands ready to evaluate the expressions which it handles. In addition, certain constructed expressions are evaluated as part of the process of applying FAMOUS' rules. Naturally, a function must be defined if it is to occur in an expression which is to be evaluated.

---

*For example, is '(f(x)*sin x + 3)*x' linear in x? It is if f(x) is identical to zero.

FAMOUS comes to the user with a fair number of functions already defined. These cannot be expected to be enough, of course, so FAMOUS is equipped with a powerful facility for definition and redefinition of functions.

This definition facility was inspired by the Advice Taker work of McCarthy [13], although it is by no means as ambitious. The Advice Taker is a thesis subject in itself, and the FAMOUS function-definition facility will get little attention in this thesis. [Warren Teitelman has been working in this area, and his thesis, "PILOT: A Step Toward Man-Computer Symbiosis", is available as Project MAC Technical Report MAC-TR-32.]

CHAPTER II

SYSTEM DESCRIPTION

2.1 DATA BASE AND COMMANDS

A user's conversation with FAMOUS proceeds as he issues a series of commands (Appendix A). These interact with each other only to the extent that they alter the following data-base:

(1) a number of expressions, notably a distinguished expression under consideration (EUC),
(2) a set of transformational rules,
(3) a set of function definitions,
(4) a hold switch, and
(5) certain backup of (1)-(3).

2.2 EXPRESSIONS

The expression under consideration (EUC) is a two-part entity. The primary part, called the compact expression under consideration (CEUC), is the object of all of FAMOUS' manipulation. The remainder of the EUC is the wherelist, which provides information about abbreviations which may be used in the CEUC.

Thus, a sample EUC is*

$$a+f(x+3) \qquad (1)$$

WHERE

$$a = 2$$

$$f(t)=\sin(t)+\cos(t)$$

Here the CEUC and wherelist are shown in that order, separated by the word WHERE.

*Here and below, algebraic expressions are shown in legitimate FAMOUS input formats (Appendix D). The functions shown in them are self-explanatory or are explained in Appendix C.

The user may command FAMOUS to save the EUC, associating it with an arbitrary name. This expression-naming facility is quite powerful. If the CEUC ever comes to include among its variable-names the name of a saved expression, then that expression is immediately substituted for that name. This allows a large expression to be constructed of smaller ones, as in the command sequence

```
consider x+y+z
save expression as fred
consider h-i
save expression as sam
consider sam*fred**(sam+a)
continue
```

which would leave

$$(h-i)*(x+y+z)**(h-i+a) \qquad (2)$$

as the EUC. There is actually a simpler way to construct (2) out of the components shown; it is described in the second paragraph of § A.1.1.

2.3 APPLICATION OF RULES

2.3.1 Structure and Utility of a Single Rule: Gross Description

Each of the rules of FAMOUS has four parts: a form, a truth-functional expression (tfe), a substitute, and a descriptor list. Crudely, a rule is applicable to an expression unless

(a) the expression is not of the same shape as the form, or
(b) after the match of the form and the expression, the tfe does not have the value "$TRUE$".

If it is found that a rule successfully matches an expression, then

(a) if the substitute is "LEAVE", the expression is unchanged.
(b) if the substitute is "EVALUATE", the expression is replaced by its value. Not all expressions have values, of course, and the system will generally rankle if meaningless evaluations are attempted.

(c) if the substitute is neither "LEAVE" nor "EVALUATE", then the expression is replaced by the substitute.

2.3.2 Structure and Utility of a Single Rule: Detailed Description

2.3.2.1 Matching a form and an expression

If a form f is to match an expression e, then

(a) If the form is a number, the name of a defined function (Appendix C), or the name of a constant, then the expression must be identical to the form.

Examples:

The form

```
4
```

matches the expression

```
4
```

and no other. The form

```
sin
```

matches the expression

```
sin
```

and no other. If

```
constantp(e)
```

has the value "$TRUE$", then the form

```
e
```

matches the expression

```
e
```

and no other.

(b) If the form is any other single name, it matches any expression.

Example:

The form

```
x
```

matches all of the expressions

x

y

4

x+ y log 4

and all others as well.

(c) If the form f is the quotation of another form g, then the expression must be identical to g.

Examples:

the form

'x

matches the expression

x

but no others, not even

'x

x+ 0

The form

'('(x+ 3))

matches the expression

'(x+ 3)

but no others.

(d) If the form consists of a function name f_o and arguments f_1, f_2, ..., f_n, then

(1) The expression e must consist of a function name e_o and arguments e_1, ..., e_n, and

(2) For i=0, 1,...,n, f_i must match e_i, and

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

This case (with n=2) covers that of infix operators, which are internally represented in prefix form.

Examples:

The form

```
machine(state, symb, tape, dir, state2)
```

matches the expression

```
fsm(q1, b, 0, xxx, q5)
```

but not the expression

```
fsm(q5, 1, 1, q2)
```

The form

```
f('(x**3), g('x), 4, g(2), f(z))
```

matches the expression

```
g(x**3, sin(x), 4, sin(2), g(3 log x))
```

but not the expression

```
g(x**3, sin(x), 4, sin(2), tan(3 log x))
```

The form

```
x+x
```

matches the expression

```
2*a*b+2*a*b
```

but not the expression

```
4+5
```

2.3.2.2 Preparation of the A-list

The primary purpose of the form-matching routine, described above in § 2.3.2.1, is a simple pass-or-fail test. In addition, however, the matching routine has the important task of preparing an association list (a-list) for the use of the tfe and the substitute. The a-list is a list of pairs; each pair consists of a name and an associated expression.

As an expression is matched to a form, each non-constant name in the form is paired with some subexpression. All of these pairs are placed on the a-list.

Like the CEUC, the form, and the substitute, the tfe is algebraic in structure. Unlike these other expressions, however, the tfe must have a defined value. After the form of a rule has successfully been matched against an expression, the value of that rule's tfe is computed. Unless that value is "$TRUE$", the rule is not applied.

The evaluation routine is described in Appendix B.

Many functions which may be useful in constructing tfe's are built into the system; these functions are listed in Appendix C. In addition, the user may define his own functions (or redefine built-in functions) by using the assert facility described in § 2.4.

One built-in function, the leave function, is of particular importance. The two arguments of leave are a name and an expression; leave has the value "$TRUE$" and the effect of adding the name-expression pair to the a-list. This effect is immediate, and any previous entry with the same first element is thus immediately superseded and effectively lost upon evaluation of the leave.

2.3.2.3 Use of the Substitute

After an expression e has passed the tests of the form and the tfe, it is entirely replaced by an expression determined by the substitute.

(a) If the substitute is "LEAVE", then the expression e is replaced by itself. This use of the word leave (=let alone) must be distinguished from any mention of the leave (=deposit) function defined in § 2.3.2.2.

(b) If the substitute is "EVALUATE", then the expression is replaced by its value (Appendix B).

(c) In all other cases, the expression e is replaced by an expression e' which is similar to the substitute. This new expression e' is derived from the substitute by considering the a-list as a list of replacements. That is, if any name occurs both in the substitute and as the first part of an entry on the a-list, then e' includes the second part of that entry wherever its first part appeared in the substitute.

2.3.3 Organization of Rules

Rules presented to the system are catalogued by the ruletypes of their forms.

(a) If e is a single name, then the ruletype of e is "ATOM".

(b) If e is the quotation of f, then the ruletype of e is the ruletype of f.

(c) If e is $f(e_1, \ldots, e_n)$, where f is a defined function, then the ruletype of e is f.

(d) If e is $f(e_1, \ldots, e_n)$, where f is not a defined function, then the ruletype of e is "UNDEFINED".

This organization is directly related to the matching algorithm of § 2.3.2.1. Suppose that a rule applicable to the expression e is sought, and suppose that the form of rule r has ruletype t. Then for r to be applicable to e, it is necessary that

(a) t be "ATOM", or
(b) t be "UNDEFINED", and ruletype(e) not be "ATOM", or
(c) t be ruletype(e).

When FAMOUS is looking for a rule which might be applicable to an expression e, it searches in accordance with ruletype information. In particular,

(a) FAMOUS first searches among the rules of the same ruletype as e.

(b) If the rules of ruletype "UNDEFINED" have not been searched in (a), and if the ruletype of e is not "ATOM", FAMOUS then searches among the rules of ruletype "UNDEFINED".

(c) If the rules of ruletype "ATOM" have not been searched in (a), FAMOUS then searches among the rules of ruletype "ATOM".

Within each group, the rules are searched in first-in-last-found order. This order is of interest only in cases in which several rules are applicable to the same subexpression.

My guess is that the more recently supplied rules are special cases of the older ones. That is, the user doesn't really want these conflicts, but neither does he want to refine all his old rules to reflect his growing understanding of what he does want. If, for example, the user has said

evaluate sin x when numberp x

and if he later has added

leave sin n when integer n

then he probably expects the later rule to take precedence.

2.3.4 Manipulation of the EUC

When FAMOUS is released upon the EUC, a complex and highly recursive process is initiated. A detailed description could be presented, but it seems more useful to list the more important running features:

(a) FAMOUS makes every effort to find applicable transformations. Whenever a rule is successfully applied and a new subexpression is produced, that subexpression is sent back to the rules. The system is quite insensitive to the particular rules which are available, and it consequently goes through elaborate procedures to avoid missing a possible transformation.

(b) On the other hand, FAMOUS will not send an expression back to the rules if that expression has just returned from the rules unchanged.

(c) Nor is it sent back if it seems to be endlessly changing. The system uses patience, a function of no arguments, to discover the number of times which a single expression may be given the full treatment.

If any expression is still in flux after that many iterations, then the system complains and breaks out of its rut. The patience function provides a useful rein when the system happens to include such rules as

replace x by x+ 1

(d) Also along this line, FAMOUS always returns to the user after a certain period of time, even if some opportunities for transformation are unexamined. This period of time is given by the oracular maxtime function, which has no arguments and has a value which is taken to be a number of tenths of seconds.

(e) The system works recursively from the bottom up. That is, the arguments of a function are always transformed before the expression consisting of that function applied to those arguments.

(f) Addition and subtraction form a special case together, since the system treats all implicit two-operand subexpressions of a complex expression built of addition and subtraction. The final ordering of the terms is determined by the expless function, which is described in Appendix C.

(g) Similarly, multiplication and division form a special case together. The final ordering of the factors and divisors is determined by expless.

(h) If an expression consists of a function f applied to certain arguments, and if the value of

opaque (f)

is "$TRUE$", then only rules having the name f as a descriptor will be applied to any part of this expression. The set of currently necessary descriptors may be a useful argument for a function in the tfe: this set of descriptors may be found on the a-list as the value of the name "NEED".

(i) If a rule with the descriptor "ABBREV" is successfully applied, then the form f and substitute s of this rule are added to the where-list, in a form like

s = f

2.3.5 Quotation

It has already been mentioned in §2.3.2.1(c) that parts of forms may be quotations of other forms. This mechanism is simple enough, but it is associated with a fairly complex scheme which is an attempt to ease discrimination of literal and schematic forms.

If the CEUC is

(1) (sin 0+cos 0)*(sin 4 z+cos 4 z)*(sin x+cos x)

then the user who says

(2) replace sin y+cos y by f(y)

clearly wants to obtain

(3) f(0)*f(4 z)*f(x)

But what of the user who, in the face of the same CEUC, says

(4) replace sin x+cos x by f(x)

In this case, the system assumes that the "x" mentioned here is a proper noun: the "x" of the CEUC (1). That is, this rule would go in as if the user had straightaway said

(5) replace sin 'x+cos 'x by f(x)

and (1) will replaced by

(6) (sin 0+cos 0)*(sin 4 z+cos 4 z)*f(x)

Still another complication arises when the user says

(7) replace sin x+cos x by f(x) when numberp x

Here, the system assumes (as it did with (2)), that x is a pronoun, bound within the rule. No strange changes are made to the rule, and (1) will be replaced by

(8) f(0)*(sin 4 z+cos 4 z)*(sin x+cos x)

Before formulating a general description of this algorithm for undercover quotation, it will be useful to define the atomsof function. Atomsof is applied to an expression e:

(a) If e is a constant or the quotation of another expression, then atomsof(e) is the null set.

(b) If e is a single name, then atomsof(e) is the unit set of e.

(c) If e is $f(e_1, \ldots, e_n)$, then atomsof(e) is the union over i of atomsof(e_i).

For example, atomsof ([the CEUC (1)]) is the set whose members are z and x.

The general rule is then as follows: If the CEUC is e and the user presents a rule with tfe t and suggested form f, then the final form is derived from f by replacing certain members of atomsof(f) by quotations of themselves. The affected names are exactly those which are members of atomsof(e) but not members of atomsof(t).

In the cases (2), (4), (5), and (7), this works out as follows:

offered rule	atomsof(f)	atomsof(t)	affected names
(2)	{y}	null set	none
(4)	{x}	null set	x
(5)	null set	null set	none
(7)	{x}	{x}	none

2.4 FUNCTION DEFINITIONS

Appendix C contains a list of the built-in functions which may be useful in construction of tfe's and in expressions which are to be evaluated. Many of these functions, in addition to being available to the user, are used internally by the system.

The user may use the assert facility (§A.3.1) to define his own functions, or to redefine the built-in ones. In the latter case, he must accept the possibility that his assertions will violently affect the behavior and perhaps the soundness of the system itself.

Each assertion affects a function by indicating the value to be returned when the arguments have certain properties. When several assertions apply to the same function, the most recent ones take precedence. This strategy is exactly that of the rule-searcher (§2.3.3) and it is justified by the same argument.

One important built-in function which the user may redefine is defined. Given a name as argument, the built-in version of defined returns "$TRUE$" or "$FALSE$" as the name is or is not the name of a defined function. As noted in §2.3.2.1 and §2.3.3, defined functions must be recognized for special handling by the form-matching and ruletype routines.

In order to affect the behavior of these routines, the user may wish to redefine the defined routine, so that certain functions are falsely considered to be defined. Examples of this strategem appear in Chapters III, IV, and V.

Similarly, the user may wish to alter the constantp function so that such names as "e" and "pi" are given special handling.

CHAPTER III

POWER OF THE SYSTEM

3.1 ABSTRACT AUTOMATA

It superficially seems useful to determine which abstract automaton (e.g., Turing machine, push-down machine) is most nearly akin to FAMOUS. I believe that such an investigation is doomed to sterility. To dispose of the issue, I present the following theorem.

3.1.1 Theorem: Any Turing machine may be encoded in a set of rules for FAMOUS

Consider a finite state automaton M, equipped with two semi-infinite counters MC1 and MC2, and the instruction set

(1) (a) I1(J), or "Increment the contents of MC1 $\left(c(MC1)\right)$ and go to instruction J"

(b) I2(J), or "Increment c(MC2) and go to instruction J"

(2) (a) D1(J, L), or

1. If c(MC1) ≠ 0, decrement c(MC1); then
2. If c(MC1) ≠ 0, do not execute subinstruction 3, but go directly to instruction J; otherwise
3. Go to instruction L."

(b) D2(J, L), or

1. If c(MC2) ≠ 0, decrement c(MC2); then
2. If c(MC2) ≠ 0, do not execute subinstruction 3, but go directly to instruction J; otherwise,
3. Go to instruction L."

(3) HALT

3.1.1.1 Lemma (Minsky, [15]):

Given the transitions of an arbitrary Turing machine T, there is an effective procedure for programming M so that if

(1) T is started scanning the xth square of a tape with the binary number k written on it, and

(2) M is started with c(MC1) = $2^k 3^x$ and c(MC2) =0,

then T will halt at the yth square of a tape with the binary number N written on it if and only if M halts with

$$c(MC1)=2^N 3^y$$

Now given the program and initial conditions of a Minsky machine M, I construct the set R(M) of rules for FAMOUS as follows:

(1) (a) If the nth instruction of M is I1(J), then I include the rule

replace state(n, c1, c2) by state (J, count(c1), c2)

(b) A similar rule is included if the nth instruction of M is I2(J)

(2) (a) If the nth instruction of M is D1(J, K), then I include the rules

replace state(n, 0, c2) by state (K, 0, c2)
replace state(n, count(0), c2) by state (K, 0, c2)
replace state(n, count(count(c1)), c2) by state(J, count(c1), c2)

(b) Similar rules are included if the nth instruction of M is D2(J, K).

(3) If the nth instruction of M is HALT, then I ignore it.

Statement #1: M, started in state i with c(MC1)=c1 and c(MC2)=c2, halts in state k with c(MC1)=N.

Statement #2: FAMOUS, told to continue with R(M) as rules, state and count defined, and

state(i, $\underbrace{\text{count(count(...count(0)...))}}_{c1}$, $\underbrace{\text{count(...count(0))}}_{c2}$) as EUC, will return to the user with

state(k, $\underbrace{\text{count(count(...count(0)...))}}_{N}$, y) as EUC.

3.1.1.2 Lemma: Statements #1 and #2 are formally equivalent.

Proof of the Lemma: Obvious.

Proof of the Theorem: Immediate from the Lemmas.

3.1.2 Example

To test the construction concretely and to provide an example of a complete FAMOUS encoding, I consider the following Minsky machine:

1. I1(2)
2. I1(3)
3. D1(4, 5)
4. I2(3)
5. D2(6, 8)
6. I1(7)
7. I1(5)
8. HALT

If MC2 is initially zero, this machine will double the initial contents of MC1. Given the assertions

```
assert defined ('state)
assert defined ('count)
```

and the rules

```
replace state(1, c1, c2) by state(2, count(c1), c2)
replace state(2, c1, c2) by state(3, count(c1), c2)
replace state(3, 0, c2) by state(5, 0, c2)
replace state(3, count(0), c2) by state (5, 0, c2)
replace state(3, count(count(c1)), c2) by state(4, count(c1), c2)
replace state(4, c1, c2) by state(3, c1, count(c2))
replace state(5, c1, 0) by state(8, c1, 0)
replace state(5, c1, count(0)) by state(8, c1, 0)
replace state(5, c1, count(count(c2))) by state(6, c1, count(c2))
replace state(6, c1, c2) by state(7, count(c1), c2)
replace state(7, c1, c2) by state(5, count(c1), c2)
```

experiment shows that FAMOUS will indeed take

$$\text{state}(1, \underbrace{\text{count}(\text{count}(\ldots\text{count}}_{n}(0), 0)$$

into

$$\text{state}(8, \underbrace{\text{count}(\text{count}(\ldots\text{count}}_{2n}(0), 0)$$

in time. Similarly, other finite-looking devices (e.g., commercial digital computers) can easily be encoded in sets of FAMOUS rules. Even a push-down stack running a, b, . . . , z from top to bottom can be encoded by a strategem like

$$\text{stack}(a, \text{stack}(b, \ldots \text{stack}(z, 0)\ldots)))$$

In all of these cases, only coding tricks are being demonstrated. The most significant of these tricks is that which allows deterministic sequential processes to be encoded at all.

3.2 MODELING AND THE WANG ALGORITHM

The real issue, however, is not one of encoding inputs and outputs. Since FAMOUS is a Turing machine, discussion of FAMOUS' abstract hierarchical power is fruitless.

Instead, the discussion must be cast in the informal terms of modeling power. There is nothing to be learned from a FAMOUS construction which is only the same black box as some external entity. When the external entity is coherently and compactly altered, I demand that the FAMOUS construction be coherently and compactly patchable.

The relevant questions, therefore, are such as these:

(a) What sort of data-structures can FAMOUS handle? What operations can FAMOUS perform on this data?
(b) In particular, what entities external to FAMOUS (e.g., algorithms) can be modeled by FAMOUS?

To help answer these questions, it will be useful to consider another example of a FAMOUS model. The external entity being modeled will be a version of the Wang algorithm for proofs in the propositional calculus [29] . My version differs from Wang's only in that mine does not explicitly produce proofs; instead, only "VALID" or "INVALID" is produced for each input schema.

The Wang algorithm consists of a complete set of rules-of-inference for the propositional calculus. These cut-free rules have the useful property that each may be run backward. That is, any schema may be effectively backed up through at least one rule to one or two other schemata. These other schemata are "simpler" in the sense of being one step closer to the finite goal of assertion ("VALID") or denial ("INVALID") of the one implicit axiom.

The rules are as follows:

If "VALID" then $\lambda \rightarrow \alpha$, where λ and α are sets of formulas and some atomic formula is a member of both λ and α.

If "INVALID" then $\lambda \rightarrow \alpha$, where λ and α are disjoint sets of atomic formulas.

If Φ, $\alpha \rightarrow \lambda$, ρ then $\alpha \rightarrow \lambda, \sim\Phi, \rho$

If λ, $\rho \rightarrow \pi$, Φ then $\lambda, \sim \Phi, \rho \rightarrow \pi$

If $\alpha \rightarrow \lambda, \Phi, \rho$ and $\alpha \rightarrow \lambda, \Psi, \rho$ then $\alpha \rightarrow \lambda, \Phi \wedge \Psi, \rho$

If $\lambda, \Phi, \Psi, \rho \rightarrow \pi$ then $\lambda, \Phi \wedge \Psi, \rho \rightarrow \pi$

If $\alpha \rightarrow \lambda, \Phi, \Psi, \rho$ then $\alpha \rightarrow \lambda, \Phi \vee \Psi, \rho$

If $\lambda, \Phi, \rho \rightarrow \pi$ and $\lambda, \Psi, \rho \rightarrow \pi$ then $\lambda, \Phi \vee \Psi, \rho \rightarrow \pi$

If $\alpha, \Phi \rightarrow \lambda, \Psi, \rho$ then $\alpha \rightarrow \lambda, \Phi \supset \Psi, \rho$

If $\lambda, \Psi, \rho \rightarrow \pi$ and $\lambda, \rho \rightarrow \pi, \Phi$ then $\lambda, \Phi \supset \Psi, \rho \rightarrow \pi$

If $\Phi, \alpha \rightarrow \lambda, \Psi, \rho$ and $\Psi, \alpha \rightarrow \lambda, \Phi, \rho$ then $\alpha \rightarrow \lambda, \Phi \equiv \Psi, \rho$

If $\Phi, \Psi, \lambda, \rho \rightarrow \pi$ and $\lambda, \rho \rightarrow \pi, \Phi, \Psi$ then $\lambda, \Phi \equiv \Psi, \rho \rightarrow \pi$

The strategy, then, is to remove the logical connectives, gradually reducing all of the component formulas to atomic ones.* Modeling this process in FAMOUS will present two problems:

(1) The formula-sets on either side of the arrow are not nestings of functions of fixed numbers of arguments.

(2) These sets cannot be encoded with the simple scheme suggested above for push-down stacks. Even after a formula has been broken into its atomic components (i.e., they have presumably been popped from the stack) these components must be remembered for the validity test.

* For a better-motivated discussion of the algorithm, the original description is still best. A realization of the algorithm via non-sequenced transformations was first proposed by McCarthy [14,§4].

One is thus naturally driven to having two stacks* on each side of the arrow. One stack on each side will be the list of formulas to be analyzed; the other stack will be a repository for atomic formulas.

I start by giving the formula as an argument to the function test. The rules will take this expression into "VALID" or "INVALID". When two premises arise from one formula, I join them into a larger formula whose connector is "*". For purposes of the match algorithm, I advise FAMOUS as follows:

```
assert defined ('test)
assert defined ('arrow)
assert defined ('equiv)
assert defined ('implies)
assert constantp ('valid)
assert constantp ('invalid)
assert constantp ('endl)
assert constantp ('endr)
```

Now I introduce two rules for the compression of our final results:

```
replace valid*p by p
replace invalid*p by invalid
```

I also need a start-up rule

```
replace test(a) by arrow(endla, endl, endra, list(a, endr))
```

two rules for collecting atomic formulas

```
replace arrow(la, l, ra, list(x, r)) by
    arrow(la, l, list(x, ra), r) when atom x
replace arrow(la, list(x, l), ra, r) by
    arrow(list(x, la), l, ra, r) when atom x
```

* Here is another Turing-machine model.

and finally the rules of Wang

```
replace arrow(la, l, ra, r) by valid when
    joint(atomsof la, atomsof ra)
replace arrow(la, endl, ra, endr) by invalid when
    null joint(atomsof la, atomsof ra)
replace arrow(la, l, ra, list(not p, r)) by
    arrow(la, list(p, l), ra, r)
replace arrow(la, list(not p, l), ra, r) by
    arrow(la, l, ra, list(p, r))
replace arrow(la, l, ra, list(a and b, r)) by
    arrow(la, l, ra, list(a, r))*arrow(la, l, ra, list(b, r))
replace arrow(la, list(a and b, l), ra, r) by
    arrow(la, list(a, list(b, l)), ra, r)
replace arrow(la, l, ra, list(a or b, r)) by
    arrow(la, l, ra, list(a, list(b, r)))
replace arrow(la, list(a or b, l), ra, r) by
    arrow(la, list(a, l), ra, r)*arrow(la, list(b, l), ra, r)
replace arrow(la, l, ra, list(implies(a, b), r)) by
    arrow(la, list(a, l), ra, list(b, r))
replace arrow(la, list(implies(a, b), l), ra, r) by
    arrow(la, list(b, l), ra, r)*arrow(la, l, ra, list(a, r))
replace arrow(la, l, ra, list(equiv(a, b), r)) by
    arrow(la, list(a, l), ra, list(b, r))*
    arrow(la, list(b, l), ra, list(a, r))
replace arrow(la, list(equiv(a, b), l), ra, r) by
    arrow(la, list(a, list(b, l)), ra, r)*
    arrow(la, l, ra, list(a, list(b, r)))
```

Returning to the modeling questions which I raised above, I consider them in reverse order.

(b) What external entities can be modeled by FAMOUS?

It might, at first glance, appear that the Wang algorithm falls well within the borders of FAMOUS' capabilities. Actually, there are some grounds for saying that whatever black-box encoding may be done, no structural model of of the Wang algorithm can be constructed within FAMOUS.

To be sure, there are some "coherent compact alterations" which can be made to the algorithm without requiring more than an extra rule or two in the model. For example, one might add

If $\alpha, \rho \rightarrow \lambda, \Phi, \Psi, \pi$ then $\alpha, \Phi|\Psi, \rho \rightarrow \lambda, \pi$
If $\alpha, \Phi, \rho \rightarrow \lambda, \pi$ and $\alpha, \Psi, \rho \rightarrow \lambda, \pi$ then $\alpha, \rho \rightarrow \lambda, \Phi|\Psi, \pi$

to the algorithm, where "a| b" might be read as "neither a nor b". The model would need to be altered only to the extent of adding

```
assert defined ('nor)
replace arrow(la, l, ra, list(nor(a, b), r)) by
    arrow(la, list(a, l), ra, r)*arrow(la, list(b, l), ra, r)
replace arrow(la, list(nor(a, b), l), ra, r) by
    arrow(la, l, ra, list(a, list(b, r)))
```

But what if the algorithm were altered so that ordering of expressions were of importance? The propositional calculus being what it is, such an alteration would admittedly be meaningless. But surely one can imagine a research environment in which a union operation had to be replaced by some sort of concatenation. Suppose, for example, that when a complex subformula were reduced, its components had to go to the end of the "to-be-analyzed" queue. The new FAMOUS encoding would certainly be more than a local modification of the old one. In brief, it cannot be said that FAMOUS allows ordered sets to be modeled cleanly.

There is another way of leading to this conclusion. That is, to observe only that the Wang rules are not obvious enough. While the Minsky model was virtually canonical, the Wang representation required tricky ad-hoc encoding. This trickiness was forced by the difficulty noted: Sets don't really fit into FAMOUS.

Entities which do fit in deal with rigid tree-structures: Algebraic formulas and fixed-length lists (e.g., state-descriptions of some automata) are the best examples of such structures.

(a) What operations does FAMOUS perform on its data?

FAMOUS has only four different capabilities: replacement, abbreviation, function definition, and function evaluation.

Abbreviation is replacement with an attached genealogy. It is no more than a human-factors trick, albeit a particularly useful one.

FAMOUS' function-definition (assert) facility is not closely tied to the rest of the system. It is almost an independent system sharing a limited portion of the FAMOUS data-base.

Function evaluation and definition might seem, moreover, to be inessential primitives. In principle, it seems reasonable to express the definition of sine as set of replacements, so that

evaluate sin x when numberp x

is expressed as a simple triggering replacement, say

replace sin x by do('sin, x) when numberp x

And at worst, the sine of a number x could be obtained from x by a set of rules which encoded a "sine" program for a digital computer.

But the fact that sine can be pragmatically approximated with a digital computer program is an accident. The meaning of sine is a differential equation or a hypergeometric series. No sequence of syntactic replacements is at all implicit in either of these formulations.

Even if a function's arguments and values may be syntactically specified, the function itself may be defined in terms of scintillation counters or human choices. No such function can be properly represented by a set of syntactic replacements, and the oracular function-evaluation facilities of FAMOUS are available for just this reason.

3.3 SEMANTICS AND Ω-THEORY

This matter of the meaning of programming notions (such as sine) is one of some contention. In conversation, Bar-Hillel [1] has asserted that the

meaning of a compiler-language program is the code into which it compiles. Feldman has gone so far as to write that the meaning of such a program is not defined until the compiler algorithms are specified [6, p. 33].

These points of view are so restrictive as to be almost meaningless. The author of a program (or the designer of a language) is often unaware of the implementation techniques which have been or could be used. Even worse, the Bar-Hillel/Feldman views entail the belief that program synonymy is irreflexive under conditions of changing implementation.

Such a notion of synonymy is appropriate under certain circumstances. FORTRAN, for example, really cannot be defined without reference to the implementation. The FORTRAN programmer is intimately concerned with the digital computer, and it matters very much which computer that is.

On the other hand, many modern programming systems derive their semantics from pre-computer notions of mathematics and logic. Computer code, although it is perhaps the first referent which comes to hand, provides no more than a slurring explanation of these systems.

Perhaps the best semantic framework in which to consider FAMOUS is that of ℧ -theory (mem-theory). In its present stage of development, this theory is more of an attitude than a quantitative formalism, and it is notoriously distorted or evaporated by synopsis.

I consequently shall not attempt to provide a general introduction to ℧ -theory. In the following notes, my remarks on ℧ -theory have more mnemonic than expository intent.

(a) Mem-theory attempts to model all computational (and other) processes as sequences of local syntactic changes. The alternate view, which ℧ -theory explicitly rejects [30, p. 2], is that of changes in total state, as perceived by an omniscient observer.

In the case of algebraic manipulation, the observer would require powers exceeding those of a Turing machine (see Chapter IV). Like ℧ -theory, therefore, FAMOUS eschews any hypostatization of a (changing or otherwise) global state.

(b) In both FAMOUS and ↄ -theory, these local changes are the result of applying transformational rules. The rules of ↄ -theory are not explicitly sequenced, and they consequently compete for roles in the process being modeled. Like the rules of FAMOUS, they do not drag the data about the ground; rather, they define the gradients and let the data roll [3, p. 8].

(c) Two laws restrain the competition among the rules; the first law is that relating to conflict [3, p. 5].

A joint application of two rules would contain conflict if it would purport to move the same portion of the accessible universe in two different directions at once. For example, the FAMOUS rules

replace 'a by 0
replace 'a by 1

would give rise to conflict if they were simultaneously applied to the expression

a

FAMOUS, of course, avoids conflict by the simple strategem of applying only one rule at a time.

(d) But in so doing, FAMOUS flaunts ↄ -theory's second law of rule-competition: that related to <u>loss</u> [3, p. 5].

Loss is approximately the far extreme from conflict. While the conflict-law warns against grouping rule-applications which are not simultaneously meaningful, the loss-law warns that certain groups of rule-applications must never be disassembled.

Consider, for example, the non-FAMOUS rules

replace "a" by 0 when the "a" appears in the context "f(a, x)"
for some non-number x
replace "b" by 0 when the "b" appears in the context "f(x, b)"
for some non-number x

These rules should obviously take

f(a, b)

into

f(0, 0)

<u>But they will do that only if they are applied simultaneously</u>. If they are modeled with the FAMOUS rules

replace f('a, x) by f(0, x) when not numberp x
replace f(x, 'b) by f(x, 0) when not numberp x

then

f(a, b)

will be taken into

f(a, 0)

or

f(0, b)

If either of the suggested rules is applied first, the other cannot be applied at all. Hence the loss law: Subject to the bound of the conflict law, maximal sets of rule-applications should be used simultaneously.

(e) Except for the conflict- and loss-laws, ↻ -theory gives no guidance for the selection of sets of rule-applications. Indeed, ↻ -theory explicitly suggests that random choices be made at this point, so that an underspecified system will predictably have unpredictable behavior [10, p. 4].

At first glance, this randomness seems to be quite contrary to the spirit of FAMOUS. Implementing true randomness in a digital computer is not at all easy, but it seems from §§ 2.3.3 and 2.3.4(e) that FAMOUS did not even try for randomness.

The non-randomness of FAMOUS is actually the result of two isolated decisions:

(1) FAMOUS is biased toward evaluation and consequently toward referentially-transparent constructions. This bias leads FAMOUS to the restriction of § 2.3.4(e).

(2) FAMOUS gratuitously assumes that the rule-provider knows what he is doing. That is, rephrasing the argument of § 2.3.3, FAMOUS believes that any apparent case of conflict is really illusory. By using the last-in-first-found algorithm described in § 2.3.3, FAMOUS effectively sees only conflict-free sets of rules.

(f) Almost any ↻ -theoretic process may be viewed (from outside the theory) as the union of two or more coupled subprocesses, each with

its own structure and rules[23]. In the regions of intersection, all of the subprocesses concerned may contribute rules; elsewhere, the subprocesses are independent.

To the extent that a subprocess is only of behavioral interest, it may be left with its oracular innards uncharted. The Σ-theoretic notion of coupling, although still only slightly developed, does seem to provide an evocative model for the function-evaluation of FAMOUS.

This empty page was substituted for a blank page in the original document.

CHAPTER IV

ALGEBRAIC SIMPLIFICATION

4.1 AN ESSAY ON SIMPLIFICATION

No one can engage in mechanical algebraic manipulation without running up against the problem of simplification. After a general-purpose algebraic algorithm has been applied to any particular data, those data will usually be in dissonant or unrecognizable forms. "Simplification" must be applied.

4.1.1 The Measure of Simplicity

The following dogmas are consequently implicit in nearly all existing systems:*

(Dogma 1) There is a unique computable partial ordering R on the set of all expressions. Each nonextendable set of equivalent expressions has a unique least element with respect to R.

(Dogma 2) There is a computable procedure f for finding, from an expression e, the equivalent expression f(e) which is minimal (simplest) under R. The procedure f is distributable, so that if e is

$$\Phi(e_1, \ldots, e_n)$$

then

$$\Phi(f(e_1), \ldots, f(e_n))$$

is at least as simple as e.

For example, R is commonly held to suggest the following:

(3) If e is a sum or product of single names, then the simplest expression equivalent to e shows those names in alphabetic order.

* These systems are numerous and approximately interchangeable. For a list, see [22].

(4) If e is wholly composed of numbers and arithmetic signs, then the simplest equivalent to e is that number which is the result of the computation expressed by e.

(5) If e is a product, one of whose factors is zero, then the simplest equivalent to e is zero.

From the behavioral standpoint, in other words, R is a set of transformations. Given a large number of such transformations, implementation of an efficient system can be a challenging puzzle.

But it is easy to become so involved in this puzzle that the real issues are lost. Note, for example, that (3) and (4) cannot possibly be a fair sampling of R. For surely

```
(x+u)*(x+v)*(x+w)*(x+y)*(x+z)
```

and

```
f(1)+f(2)+f(3)+f(4), where f(i)=2**2**i
```

seem simpler than

```
(u+x)*(v+x)*(w+x)*(x+y)*(x+z)
```

and

```
66066
```

Even (5) would be suspect, if a limit problem were being examined for applicability of L'Hôpital's rule.

But suppose that patches could correct (3), (4), and (5). What epicycles of R could possibly cope with such expressions as

(6) `0*sin(x)+1*cos(x)+2*tan(x)+3*cot(x)+4*sec(x)+5*csc(x)`

This mathematically unruly expression has a ridiculous, complicated graph. Expressed in sines and cosines, it is an undistinctive jumble. But

in the exact form shown, it is an indexed list of the trigonometric functions, in the order in which we met them in high school. The expression (6) would not be simplified by alphabetically ordering the terms, nor even by eliminating any term.

And dogma (1) collapses altogether when confronted with the equivalent forms

(7) 1/(1+cos x)

and

(8) (csc x)**2-cot(x)*csc x

Is (7) simpler than (8)? Certainly (7) is handier if one would sketch a graph. But integration of (7) requires the ugly z=tan(x/2) transformation, while (8) integrates immediately.

Dogma (2) fares even worse than dogma (1). Such trivial examples as

(9) (x+y)**2-2*x*y

suggest that the hill-climbing (theorem-proving) problems of dogma (2)'s procedure f may be formidable.

But dogma (2) is not suggested to death; dogma (2) is proved to death:

Theorem (Richardson, [20]):

Let N be the set of one-place real functions generated by composition from the following primitives:

rational numbers
pi
log(2)
addition
multiplication
sin
exp

Now let f be a member of M. The assertion

$$(\exists x)\ (f(x) < 0)$$

is recursively undecidable.

Corollary: Let N be the set of one-place real functions generated by composition from the primitives of M and the notion of absolute value. Now let f be a member of N. The assertion

$$(\exists x)\ (f(x) \neq 0)$$

is recursively undecidable.

Most algebraic manipulation systems* are at least as rich as N. In all of these systems, the rule

(10) replace x+0 by x

is present. But Richardson's corollary shows that there can be no exhaustive way of searching for the "0" of (10).

In even richer systems, of course, results like Richardson's are quite easy to achieve. Once the trigonometric and arctrigonometric functions are at hand, for example, it is a few minutes' exercise to express the floor (greatest-integer-less-than-or-equal-to) function.** Given floor and (10), it is trivial to express such gems as the Fermat conjecture as simplification problems.

4.1.2 The Context of Simplicity

Even though (3), (4), and (5) are obviously not of universal value, they might be said to describe simplification in certain limited contexts. In one such context, one might add such rules as

* But not all. For example, see ALPAK [2].

** E.g., as x-(2/pi)*atan((f(x)+f(x+1))/2), where f(y)=g(y)+abs g(y), g(y) = tan (pi*y/2)

(11) replace sin x by x

if all of sin's arguments are known to be near zero,

(12) replace x**n by 0 when numberp n and
greaterp (n, 3)

if everything is near zero,

(13) abbreviate sin x + cos x as f(x)

if expressions of the form sin x + cos x are confusingly numerous,

(14) replace x by y when numberp x and floatp x and equal (x,
fix(x+0.5)) and leave ('y, fix(x+ 0.5))

if non-integral numbers (e.g., 355.0000001) very close to integers are necessarily the results of errors (i.e., from 355),

(15) replace x by y*pi when numberp x and
integer(x/3.1415927) and
leave('y, fix(x/3.1415927+0.5))

If numbers which are nearly exact multiples of pi (e.g., 355) should be represented as such (i.e., 113*pi), and

(16) replace x by 2

if an attempt at parameterization were discovered to have been unilluminating.

But now it is not clear how simplification differs from the rest of algebraic manipulation. Certainly (16) could not conceivably be part of any fixed "simplifier". The reader may well suspect that he has taken a wrong turn.

I believe that the fuzziness of "simplification" is inherent. The algebraic notion of "simplification" disappears under scrutiny like the linguistic notion

of "referent".* There appears to be no real loss to the surrounding subject in either case.

Indeed, the vanishing of "simplification" is of positive value. In systems in which "simplification" is the work of a well-defined subsystem, it is possible to ask "During such-and-such a complexity-producing manipulation, when should the simplifier be used?" The false hypostatization of "simplification" has fostered the development of a considerable literature (e.g., [27]) devoted to such pseudo-questions.

4.2 THREE ESSAYS AT SIMPLIFICATION

One interesting use to which a general-purpose system like FAMOUS may be put is imitation of special-purpose systems. In this way, these systems may be described in a compact common language.

4.2.1 Wooldridge-Russell Simplify (WRS)

All modern "simplifiers" can trace their ancestry to Wooldridge-Russell Simplify [32], which evolved at Stanford in the years up to 1963. Despite its age, WRS includes many features which most modern "simplifiers" omit.

One of these features is a well-integrated polynomial facility, which I have not tried to model in FAMOUS. The WRS treatment of non-polynomial division is complexly tied to the polynomial facility; WRS performs synthetic division to find non-explicit factors. This division feature was also not modeled. In every other respect, WRS is completely described in the remainder of this section.

WRS labels certain results as undefined; it expects the undef label to propagate as follows:

```
assert defined ('undef)
assert opaque ('undef)
```

* The analogy is not accidental. In both cases, a totality (utterances or expressions) is partitioned (by synonymy or algebraic equivalence), and the equivalence classes are then hypostatized into a life of their own. Quine [19, Chapter II] is particularly eloquent concerning these matters; see also Craik [4, p. 102].

```
replace f(undef x) by undef f(undef x)
replace f(undef x, y) by undef f(undef x, y)
replace f(x, undef y) by undef f(x, undef y)
```

WRS has three rules for unary minus:

```
evaluate -n when numberp n
replace --x by x
replace -(a-b) by b-a
```

Four rules for reciprocals:

```
replace recip 0 by undef recip 0
evaluate recip n when numberp n and not zerop n
replace recip recip x by x
replace recip (-x) by -recip x
```

Three rules for division, one of which is only suggested here:

```
replace x/y by quotient when equal (0, remainder(x, y))
replace x/0 by undef (x/0)
replace x/y by x*recip y when not equal(0, y)
```

Six rules for the power operator

```
replace 0**a by 0
evaluate a**n when numberp a and numberp n
replace a**0 by 1
replace 0**0 by undef (0**0)
replace 1**a by 1
replace a**1 by a
```

The rules for addition are simple, but lingering behind the first is the assumption of a canonical ordering for products, with numbers first. This is as good a place as any to set down a fair definition of expless (§§ 2.3.4(f) and 2.3.4(g)). To that end, I start with an auxiliary function

```
assert complexity(x) = 2
assert if atom x then complexity x=1
assert if numberp x then complexity x=0
```

and now the definition of expless itself

```
assert expless(a, b, $FALSE$) = lessp(complexity a, complexity b)
assert expless(a, b, $TRUE$) = not expless(a, b, $FALSE$)
assert if equal (complexity a, complexity b) then expless(a, b, $FALSE$)=
      lessp(canonical a, canonical b)
assert if onep complexity a and onep complexity(b) then expless(a, b, c)=
     not(alphaorder(a, b) )
```

Now the awaited rules of addition:

```
replace n*a+m*a by (n+m)*a when numberp n and numberp m
evaluate m+n when numberp m and numberp n
replace a+m*a by (m+1)* when numberp m
replace a+a by 2*a
replace b+(-a) by b+(-1)*a
replace a+0 by a
```

Multiplication is similar, but non-numeric exponents are combined as well as numeric ones:

```
evaluate a*b when numberp a and numberp b
replace a*a by a**2
replace a*a**b by a**(b+1)
replace a**c*a**b by a**(b+c)
replace a*recip a by 1
replace a*recip(a**b) by a**(1-b)
replace a**b*recip(a**c) by a**(b-c)
replace a**b*recip(a) by a**(b-1)
replace a*(-b) by (-1)*a*b
replace 1*a by a
replace 0*a by 0
replace recip(a)*recip b by recip(a*b)
```

This completes the list of ordinary rules of WRS. Three other features of WRS should be mentioned in this description. First of all, WRS allows users to provide a factor list of names. When the name "x" is placed on this list, the rule

```
replace a+b by x*(al+otherfactor) when
    factor ('x, a) and leave('al, otherfactor) and
    factor ('x, b)
```

is added to the system.

Secondly, users of WRS are able to specify a setting of the recipmode switch. In FAMOUS terms, recipmode is a function to define like meter. The system includes the following rules:

```
replace a**(-b) by recip (a**b) when recipmode( )
replace recip(a**b) by a**(-b) when not recipmode( )
replace a**n by recip(a**m) when recipmode( ) and
    numberp n and minusp n and leave ('m, -n)
replace recip(a)**b by a**(-b) when not recipmode( )
```

Finally, WRS accepts an expand list roughly opposite in function to the factor list. If a name "x" is on the expand list, then the following rules are added.

```
replace (a+b)*c by a*c + b*c when
    member ('x, union(atomsof a, atomsof b))
replace (a*b)**c by a**c*b**c when
    member ('x, union(atomsof a, atomsof b))
replace (a+b)**n by (a**2+2*a*b+b**2)*(a+b)**m when
    member ('x, union(atomsof a, atomsof b)) and
    afixp n and greaterp (n, 1) and leave ('m, n-2)
replace a+recip(b) by (1+a*b)*recip b when
    member ('x, atomsof b)
replace a+n*recip(b) by (n+a*b)*recip b when
    member ('x, atomsof b) and numberp n
```

4.2.2 AUTSIM

This section is devoted to a partial model of the AUTSIM "simplifier" of FORMAC. The incompleteness of the model is not due to any difficulties of modeling per se, but rather to the fact that AUTSIM is only partially documented.*

For the most part, of course, the rules of AUTSIM are the same as those of WRS or any other conventional "simplifier". Only the distinctive features of AUTSIM will be described here:

(a) Nothing in FORMAC corresponds in any way to the factor-list, recipmode, and expand-list features of WRS.

(b) One type of acceptable FORMAC constant is the rational number, and it is important to note here that AUTSIM demands that all sums, differences, products and quotients of constants be evaluated.

```
assert r(a, b) = a/b
```

The rules first arrange for reduction to lowest terms:

```
evaluate r(0, a)
evaluate r(a, a)
evaluate r(a, 1)
replace r(a, b) by r(numerator, denominator) when
    not reduced (a, b)
assert reduced(a, b)
assert if leave('g, gcd(a, b)) and not onep g and
    leave('numerator, a/g) and leave('denominator,
    b/g) then not reduced(a, b)
assert gcd(a, b) = gcd(rem(b, a), abs(a))
assert if zerop rem(b, a) then gcd(a, b)=abs(a)
assert if greaterp(a, b) then gcd(a, b) = gcd(b, a)
assert rem(a, b) = abs(a) - abs(b*fix(a/b))
assert abs(a) = a
assert if minusp a then abs a = -a
```

* My primary source was [28], but I found useful bits and pieces throughout the FORMAC literature. For a list of that literature, see [22].

The rules should also make an effort to preserve rational form:

```
assert quasinumber(n)=numberp n or
    equal(mainof n, 'r)
evaluate f(a, b) when infixop f and
    not equal(f, '**) and
    quasinumber a and quasinumber b
assert integer(x) = numberp x and
    (equal(x, fix(x+0.5)) or equal(x, fix(x-0.5)))
replace n+r(a, b) by r(a+b*n, b) when
    integer n
replace n-r(a, b) by r(b*n-a, b) when
    integer n
replace r(a, b) -n by r(a-b*n, b) when
    integer n
replace n*r(a, b) by r(a*n, b) when
    integer n
replace n/r(a, b) by r(b*n, a) when
    integer n
replace r(a, b)/n by r(a, b*n) when
    integer n
replace r(a, b)+r(c, d) by r(a*d+b*c, b*d)
replace r(a, b)-r(c, d) by r(a*d-b*c, b*d)
replace r(a, b)*r(c, d) by r(a*c, b*d)
replace r(a, b)/r(c, d) by r(a*d, b*c)
```

Finally, the rules must arrange for sign management:

```
evaluate -n when numberp n
replace -r(a, b) by r(-a, b)
replace r(a, b) by r(-a, -b) when minusp b
```

(c) AUTSIM effectively factors away the minus sign of expressions which are raised to integral powers.

```
replace (-a)**n by a**n when even(n)
replace (-a)**n by -(a**n) when odd(n)
assert even(n) = integer n and zerop rem(n, 2)
assert odd(n) = integer n and not even n
```

(d) In conversation, R.G. Tobey has informed me of the following undocumented feature of AUTSIM. If e is a sum, one of whose terms is "log z" for some z, then AUTSIM will replace

exp(e)

by

z*exp(e')

where e' is the result of removing the "log z" term from e.

To model this facility in FAMOUS, it is necessary to have a means of testing whether a sum contains a logarithmic term.

```
assert not logsum(x)
assert if equal(mainof x, 'log) then logsum x
assert if sum x then logsum x=
    logsum arg(x, 1) or logsum arg(x, 2)
```

Given the logsum test to prevent endless searching, it is simple to send a log-seeking syntactic device into a sum.

```
assert defined('findlog)
assert defined('foundlog)

replace exp x by exp findlog x when logsum x
replace findlog a by a
replace findlog(a+b) by findlog a + findlog b
replace findlog(a-b) by findlog a-findlog b
replace findlog log z by foundlog(z, 0)
```

Having found a logarithmic term, the rules bring this term out of the sum.

```
replace a+foundlog(z, b) by foundlog(z, a+b)
replace a-foundlog(z, b) by foundlog(1/z, a-b)
replace foundlog(z, a)-b by foundlog(z, a-b)
replace exp foundlog(z, a) by z*exp a
```

(e) The functions of FORMAC are divided into three classes:

(1) The four arithmetic operators
(2) Integer-valued functions, like factorial
(3) Transcendental functions

Two special switches control evaluation of expressions consisting of integer-valued or transcendental functions applied to constant arguments.

```
evaluate f(x) when quasinumber x and
    (intfcn(f) and evalintfcn( ) or
    transfcn(f) and evaltransfcn( ))
evaluate f(x, y) when quasinumber x and
    quasinumber y and (intfcn(f) and
    evalintfcn( ) or transfcn(f) and
    evaltransfcn( ))
```

The intfcn and transfcn predicates, of course, can be defined by enumeration:

```
assert not transfcn(f)
assert transfcn('**)
assert transfcn('exp)
      etc.
```

(f) Three undefined forms are recognized by FORMAC: zero to a negative power, log(0), and 0**0. In the first two cases, FORMAC substitutes 0 and prints a diagnostic; the last case is ignored.

```
replace 0**n by 0 when numberp n and
    minusp n and typeout('zero.to.negative.power)
replace 0**r(a, b) by 0 when minusp a and
    typeout('zero.to.negative.power)
replace log 0 by 0 when typeout('log(0))
leave 0**0
```

(g) Finally, FORMAC is intent upon cancelling terms, and several apparently dilatory rules are directed toward this end.

```
replace a+log(b*c) by a+log b+log c
replace a-log(b*c) by a-log b-log c
replace log(b*c)-a by log b+log c-a
replace (a*b)**c by a**c*b**c
replace (a/b)**c by a**c/b**c
```

4.2.3 An Exercise

As a final experiment along the lines of "simplification", a FAMOUS "simplifier" was incrementally constructed in response to the demands of a relatively coherent series of mathematical problems.

These problems were the differentiation exercises in an elementary text [26] . The experiment covered all of the problems involving rational functions, trigonometric functions, inverse trigonometric functions, and natural logarithms. The experiment was stopped at this point by a rising tedium/machine-time ratio, but it could conceivably have been carried through the remainder of the text.

The differentiation itself naturally gave FAMOUS no trouble at all. In addition, differentiation proved to be a copious source of expressions worth "simplifying". This was quite fortunate, since the cognitive dissonance of "simplifyable" expressions makes them very difficult to produce by hand.

The "simplifier" developed here came to include a number of really odd-looking rules. These rules reflect recurrences of such specialized expressions as

```
x*(1-(a/x)**2)**(1/2)
```

On the other hand, a number of rather prosaic rules added early in the game turned out to be undesirable in the long run. This was hardly surprising, in view of the arguments of §§4.1.1 and 4.1.2. Inasmuch as there was no hope for an asymptotic set of rules, the rules in question were left in.

A number of expressions could only have been unraveled by the most bizarre ad hoc rules. These expressions were generally in one or another factored form, and the rules which were needed to "simplify" these expressions would have had to look ahead to the possible merits of expansion. In §6.1, I

discuss an improved matching procedure for FAMOUS which might be useful in easing situations of this kind.

Here are the rules and assertions which were developed, in the order in which they appeared; a definition of expless, not shown, preceded everything. The first few rules are taken from the text:

```
assert defined('d)
replace d(x, x) by 1
replace d(x**n, x) by n*x**(n-1) when numberp n
replace d(u*v, x) by u*d(v, x)+v*d(u, x)
replace d(u+v, x) by d(u, x)+d(v, x)
replace d(u-v, x) by d(u, x)-d(v, x)
```

The expression

```
d(t**2-4*t+3, t)
```

first to be tested, became

```
2*t**(2-1)+0-(4*1+0*t)
```

and the following rules were added:

```
evaluate f(x, y) when numberp x and numberp y and infixop f
replace x**1 by x
replace x+0 by x
replace x*1 by x
replace x*0 by 0
```

The expression

```
2*t**3-5*t**2+4*t-3
```

became

```
6*t**2-10*t+4-0
```

and the following rule was added:

```
replace x-0 by x
```

Similarly, the remaining rules were added in response to the various demands of circumstance. Only spot-check cases will be remarked upon.

```
replace x-x by 0
evaluate -n when numberp n
replace x*x**n by x**(n+1)
replace x*z-x*w by x*(z-w)
replace x*(z*x+w) by z*x**2+w*x
replace x-z*x by x*(1-z)
replace (-1)*x by -x
replace -x-y by -(x+y)
replace x*(-z) by -(x*z)
replace x*(-z) by (-x)*z when numberp x
replace m*(x-n) by m*x-m*n when
    numberp m and numberp n
replace m*(x+n) by m*x+m*n when
    numberp m and numberp n
replace d(u/v, x) by (v*d(u, x)-u*d(v, x) )/v**2
```

The expression

```
d(t/(t**2+1), t)
```

became

```
(t**2+1-2*t*t)/(t**2+1)**2
```

When the rule

```
replace x*x by x**2
```

was added, the expression advanced to

```
(-(t**2)+1)/(t**2+1)**2
```

and the unary minus was finally cleared up with

```
replace -a+b by b-a
replace a-n by a+(-n) when
    numberp n and minusp n
replace n*y+a by a-(-n)*y when numberp n and
    minusp n
```

```
replace a*b**n by a /b**(-n) when numberp n and minusp n
replace a**n/b**n by (a/b)**n
replace a+n/b by a-(-n)/b when numberp n and minusp n
replace y**(-1) by 1/y
```

The expression

```
d(x/(x**2-4)**0.5,x)
```

became

```
( (x**2-4)**0.5-x**2/ (x**2-4)**0.5 )/ ((x**2-4)**0.5)**2
```

The rule

```
replace ( a**b)**c by a** (b*c)
```

was obvious enough, but the ungainly

```
replace (a-b/c**y) /c by (a*c**y-b)/c**(y+1)
```

was also necessary. The expression

```
d( (x+1)**2*(x**2+2*x)**-2,x)
```

was treated at this time, and an impasse was reached at

```
(2*x+2)*(2*x+x**2-(2*x+2)*(x+1) )/(2*x+x**2)**3
```

Only an ad hoc rule could have profitably been added; the rule

```
now replace (a+b)*(c+e) by a*c+a*e+ b*c+ b*e
```

was successfully used instead.

```
replace 0-y by -y
replace y by z when numberp y and floatp y and
    equal (y,fix (y+ 0.5) ) and leave ('z, fix (y+ 0.5) )
```

```
replace a--b by a+b
replace y*z+y*w by y* (z+w)
replace n-y by -(-n+y) when numberp n and minusp n
replace (m*y+n) /p by (m/p) *y+n/p when numberp m and
     numberp n and numberp p
replace y**n+ y*w by y* (y**(n-1) +w) when numberp n
replace a**n/a**m by a** (n-m)
replace (-a) /b by -(a/b)
replace (a/b)**n*b**m by a**n*b**(m-n)

assert cos(u) = sin(u+1.5707963)
replace d(sin u, x) by d(u, x)*cos u
replace d(cos, u, x) by -d(u, x)*sin u
replace -(a*b) by (-a)*b when numberp a
replace m*y-n*y by (m-n)*y
replace (sin u) **2+(cos u)**2 by 1
assert tan (u) = sin u/cos u
assert cot (u) = cos u/sin u
assert sec (u) = recip cos u
assert csc (u) = recip sin u
replace d(tan u, x) by d(u, x)*(sec u)**2
replace d(cot u, x) by -d(u, x) * (csc u)**2
replace d(sec u, x) by (sec u)*(tan u)*d(u, x)
replace d(csc u, x) by -(csc u)*(cot u)*d(u, x)
```

The expression

```
d(sec(x)**4-tan(x)**4, x)
```

became

```
4*(tan(x)*sec(x)**4-tan(x)**3*sec(x)**2)
```

Only the ugliest adhocity, viz.

```
now replace t*s**4-t**3*s**2 by t*s**2*(s**2-t**2)
```

would do the trick, after which

```
replace (sec u)**2-(tan u)**2 by 1
```

produced the desired result.

```
replace (-a)**n by a**n when even (n)
assert even (x) = integer x and zerop rem(x, 2)
assert rem (a, b) = a-abs(b*fix (a/b) )
assert abs(a) = a
assert if minusp a then abs a= -a
replace cos(u)*sin(u) by (1/2)*sin 2 u
assert acos(u) = abs(asin u -1.5707693)
assert acot (u) = atan recip u
assert asec(u) = acos recip u
assert acsc(u) = asin recip u
replace d(asin u, x) by d(u, x)*(1-u**2)** -0.5
replace d(acos u, x) by -d(u, x)*(1-u**2)** -0.5
replace d(atan u, x) by d(u, x) / (1+u**2)
replace d(acot u, x) by -d(u, x)/(1+u**2)
replace d(asec u, x) by d(u, x) / (abs(u) * (u** 2-1)** 0.5
replace d(acsc u, x) by -d(u, x)/(abs(u)*(u**2-1)** 0.5
replace abs(n*x) by n*abs x when numberp n and greaterp (n, 0)
replace a/(-b) by (-a)/b when numberp a
```

The expression

```
d(acot(2/x) + atan (x/2) , x)
```

became

```
0.5/( (x/2) **2+1) -- 2/(x**2*( (2/x)**2+1))
```

The "--2" was easy enough to get rid of:

```
replace a-b/c by a+(-b)/c when numberp b and minusp b
```

But the confusion in the fraction took some effort to root out:

```
replace u**2* ( (a/u)**2+b) by a**2+b*u**2
```

```
replace a/( (b/c)**2+y) by a*c**2/(b**2+y*c**2) when numberp c or
    numberp y
replace a/b+c/b by (a+c)/b
replace a**0 by 1
replace 0/a by 0
replace (n*x)**m by n**m*x**m when integer n and integer m
replace d(log u,x) by d(u,x)/u
```

The expression

```
d(x*log x-x,x)
```

only got to

```
log x+x/x-1
```

without

```
replace a/a by 1
```

It is somewhat surprising that this rule should wait so long to appear.

```
replace (a/b+b)/c by (a+b**2)/(b*c)
replace a+a by 2*a
replace n*u+u by (n+1) *u when numberp n
replace a**n/a by a** (n-1)
replace a/a**n by a** (1-n)
replace a**b*a**c by a**(b+c)
replace a*(b/a-c) by b-a*c
```

CHAPTER V
LIMIT PROBLEMS

5.1 GENERAL DISCUSSION

This chapter is devoted to discussion of FAMOUS - based systems for the solution of "limit problems". These problems are typically represented as

(1) $\lim_{x \to k} \epsilon$

or

(2) $\lim_{x \to k+} \epsilon$

or

(3) $\lim_{x \to k-} \epsilon$

where ϵ is a finite expression compounded of piecewise-analytic functions; k is a real number, $-\infty$, ∞, or $+\infty$; and the desired answer is "indeterminate" or like k.

Freshmen are occasionally given "limit problems" involving the characteristic function of the set of rational numbers. Sometimes other unruly functions are used, depending upon the imagination of the instructor. Despite the existence and persistence of these non-Borel anomalies, my restriction to piecewise analyticity excludes few problems found in elementary texts.

The most striking thing about limit problems is the fact that only one effectively computable procedure for solving them — L'Hôpital's Rule — seems to exist. Unfortunately, this is not to say that L'Hôpital's rule solves all limit problems.

Consider, for example,

(4) $\lim_{x \to +\infty} \sin x / \exp x$

The answer is plainly zero, but L'Hôpital's rule is of no help.

A more malignant example starts with

(5) $\lim_{x \to +\infty}$ exp x/exp x

Here again, the answer is plain. But the answer is reached through simplification, not through L'Hôpital's rule. And if (5) had been written as

(6) $\lim_{x \to +\infty}$ exp x/(exp x + f(x)*exp(x**2))

where f(x) is a Richardson function (§ 4.1.1), non-obviously identical to zero, the answer would not have been plain at all.

The example keeps the following disappointing theorem from being very surprising.

Theorem: Let N be the set of one-place real functions defined for the Richardson corollary (§ 4.1.1). Then there is no recursive decision procedure for statements of the form

$$\lim_{x \to +\infty} f(x) = 0$$

for f in N.

Proof: Suppose such a decision procedure existed, and let g be a function in N. Then the following are equivalent:

$$(\exists\, x)(g(x) \neq 0)$$

and

$$\lim_{x \to +\infty} g(x \sin x) \neq 0$$

But it is known to be undecidable whether g is identically zero. QED.

Despite the theorem, textbook problems are handled so mechanically by competent students that it seemed worthwhile to prepare a set of FAMOUS rules which would attack these problems.

The "two-sided" limit problems of type (1) turn out to be quite a bit cleaner than the "one-sided" problems of types (2) and (3). In the case of limits of type (1), piecewise analyticity of the function f allows us to replace

(7) $\lim_{x \to k} f(\epsilon)$

by

(8) $f(\lim_{x \to k} \epsilon)$

But the corresponding transformations for one-sided limits are not legitimate. At their points of singularity, piecewise analytic functions comply with the (7)-to-(8) rule by letting one or both of (7) and (8) be indeterminate. The corresponding expressions for one-sided limits, however, are often both defined — and different in value. In an effort to return to the ordinary, inside-out evaluation rule, the misleading notations k+ and k- have been used to represent values of real variables.

Theorem: Let P be the set of one-place real functions generated by composition from the primitive notions of

the rationals
pi
log 2

addition
multiplication
division
sin
exp
abs

Suppose f(x) is in P and $\lim_{x \to k+} f(x) = a$. Then the statement

As x descends to k, f(x) descends to a

is recursively undecidable.

Proof: Suppose the decision procedure denied by this theorem existed, and let g be a function in P. Observe that

(a) If a total function h(x) ever has the value v, then h(sin(1/x)/x) has the value v — infinitely often, in fact — in every open interval (0, a)

(b) x/(abs x+1) is always defined, and its magnitude is always less than 1. It is negative when and only when x is.

Thus g is always greater than or equal to zero if and only if

x*g(sin(1/x)/x)/(abs g(sin(1/x)/x) +1)

descends to zero as x does. But it is recursively undecidable whether g is always greater than or equal to zero. QED.

This theorem describes one of the reasons why the technique which takes

$\lim_{x \to 0+}$ exp recip sin x

into

exp recip (0+)

and consequently into

+ ∞

cannot be generalized very far. To be sure, one can specify the treatment of any number of fixed one-place functions. For example, one chooses between

(value of sin k) +

and

(value of sin k) -

as a replacement for

$$\lim_{x \to k+} \sin x$$

by simply finding the quadrant in which k lies. But the situation is beyond saving when the (two-place) arithmetic operators are introduced. The solution of

$$\lim_{x \to 0+} \exp \text{ recip } (\tan x - \sin x)$$

is not approached by the transformation into

exp recip ((0+) - (0+))

5.2 A FAMOUS SYSTEM FOR TWO-SIDED LIMIT PROBLEMS

The arguments of the preceding section show that one-sided and two-sided limit problems are (to a first approximation, ignoring possible degrees of unsolvability) equally and impossibly difficult. Experience, of course, suggests that textbook problems of these types are about equally and trivially easy.

In any event, there can be no question that one-sided limit problems require for their solution a large and disorderly miscellany of correlative information.

For this reason, no machine system was developed for one-sided limit problems. This section describes a FAMOUS system for the solution of two-sided limit problems. The system was developed in exactly the order in which it is presented.

The reader will notice any number of implausible rules which, he should also notice, are largely eclipsed by rules which <u>follow</u> them. The last-in-first-found rule-search of § 2.3.3 should be recalled.

To represent

$$\lim_{x \to k} \epsilon$$

I use the notation

limit (x, k, ϵ*)

where ϵ* is the FAMOUS representation of ϵ. The basis and induction step for one-place functions are simple:

(1) assert defined ('limit)
(2) replace limit (x, k, y) by y when atom y
(3) replace limit (x, k, x) by k
(4) replace limit (x, k, f(y)) by f(limit (x, k, y))

For the case of the (two-place) arithmetic functions, the rule

(NOT 5) replace limit (x, k, f(y, z)) by f (limit (x, k, y), limit (x, k, z))

must <u>not</u> be included. Taking the limits of the arguments y and z may lead to one of the indeterminate forms. To aid in describing these forms, I introduce three constants:

(5) assert constantp ('infinity)
(6) assert constantp ('plusinfinity)
(7) assert constantp ('minusinfinity)
(8) assert infinite (x) = equal (x, 'infinity) or equal (x, 'plusinfinity) or equal (x, 'minusinfinity)

Now the indeterminate forms can be explicitly listed:

(9) assert constantp ('indeterminate)
(10) assert indet (a) = equal (a, 'indeterminate)

(11) assert nastysum (a, b)= indet a or indet b
(12) assert if infinite a then nastysum (a, infinity)
(13) assert if infinite b then nastysum (infinity, b)
(14) assert nastysum (plusinfinity, minusinfinity)
(15) assert nastysum (minusinfinity, plusinfinity)

(16) assert nastydifference (a, b)=indet a or indet b
(17) assert if infinite a then nastydifference (a, infinity)
(18) assert if infinite b then nastydifference (infinity, b)
(19) assert if infinite a then nastydifference (a, a)

(20) assert nastyproduct (a, b)=indet a or indet b
(21) assert if infinite a then nastyproduct (a, 0)

(22) assert if infinite b then nastyproduct (0, b)

(23) assert nastyquotient (a, b)=indet a or indet b
(24) assert if infinite a and infinite b then nastyquotient (a, b)
(25) assert nastyquotient (0, 0)

(26) assert nastypower (a, b)=indet a or indet b
(27) assert if infinite a or equal (0, a) then nastypower (a, 0)
(28) assert if infinite b then nastypower (1, b)
(29) assert nastypower (a, infinity)

And in the cases of the arithmetic operators, I simply

(a) Determine the limits of the arguments;
(b) If they do not lead to indeterminacy, apply rules which serve the purpose of (NOT 5);
(c) Transform the expression into quotient form and apply L'Hôpital's rule, if the rules equivalent to (NOT 5) have not been applied.

Step (c) is not without its difficulties. Given that

tan (2 t) *csc 4 t

becomes 0 * ∞ as t goes to zero, it is not obvious that the useful equivalent quotient is

(30) tan 2 t/sin 4 t

rather than

(31) csc 4 t/cot 2 t

An algorithm for choices of this kind is not forthcoming, and it is consequently necessary to try both alternatives simultaneously until one is fruitful. This stategem must even be applied to expressions which are already quotients; one might be given (31), and one would hope for a path through (30).

With this discussion in mind, I tentatively separate the arguments of the arithmetic operators:

(32) assert defined ('sumlimit)
(33) assert defined ('differencelimit)
(34) assert defined ('productlimit)
(35) assert defined ('quotientlimit)
(36) assert defined ('powerlimit)

(37) replace limit (x, k, el+e2) by sumlimit (el, e2, x, k)
(38) replace limit (x, k, el-e2) by differencelimit (el, e2, x, k)
(39) replace limit (x, k, el*e2) by productlimit (el, e2, x, k)
(40) replace limit (x, k, el/ e2) by quotientlimit (el, e2, x, k)
(41) replace limit (x, k, el**e2) by powerlimit (el, e2, x, k)

The sumlimit, . . ., powerlimit functions will ultimately be a handle on the limits of the arguments:

(42) assert defined ('limitsum)
(43) assert defined ('limitdifference)
(44) assert defined ('limitproduct)
(45) assert defined ('limitquotient)
(46) assert defined ('limitpower)
(47) replace sumlimit (el, e2, x, k) by limitsum (el, e2, x, k, limit (x, k, el), limit (x, k, e2))
(48) replace differencelimit (el, e2, x, k) by limitdifference (el, e2, x, k, limit (x, k, el), limit (x, k, e2))
(49) replace productlimit (el, e2, x, k) by limitproduct (el, e2, x, k, limit (x, k, el), limit (x, k, e2))
(50) replace quotientlimit (el, e2, x, k) by limitquotient (el, e2, x, k, limit (x, k, el), limit (x, k, e2))
(51) replace powerlimit (el, e2, x, k) by limitpower (el, e2, x, k, limit (x, k, el), limit (x, k, e2))

Before that, however, sumlimit, ... , powerlimit provide a good point at which to perform, in accordance with the arguments of § 5. 1, all the "simplification" one cares to describe.

Most of this manipulation, of course, will conform to the patterns of Chapter IV:

(52) replace differencelimit (el, el, x, k) by 0
(53) replace sumlimit (el, el, x, k) by 2* limit (x, k, el)
etc.

It is natural to take this opportunity to shortcut (2):

(54) replace sumlimit (el, c, x, k) by limit (x, k, el) + c when constantp c
(55) replace sumlimit (c, e2, x, k) by limit (x, k, e2) + c when constantp c
(56) replace differencelimit (c, e2, x, k) by c - limit (x, k, e2) when constantp c
(57) replace differencelimit (el, c, x, k) by limit (x, k, el) - c when constantp c
(58) replace productlimit (el, c, x, k) by c* limit (x, k, el) when constantp c
(59) replace productlimit (c, e2, x, k) by c* limit (x, k, e2) when constantp c
(60) replace quotientlimit (c, e2, x, k) by c* recip limit (x, k, e2) when constantp c
(61) replace quotientlimit (el, c, x, k) by limit (x, k, el)/c when constantp c
(62) replace powerlimit (el, c, x, k) by limit (x, k, el)**c when constantp c
(63) replace powerlimit (c, e2, x, k) by c** limit (x, k, e2) when constantp c

But one must be sure to catch the forms which are already indeterminate:

(64) replace sumlimit (el, e2, x, k) by indeterminate when nastysum (el, e2)
(65) replace differencelimit (el, e2, x, k) by indeterminate when nastydifference (el, e2)
(66) replace productlimit (el, e2, x, k) by indeterminate when nastyproduct (el, e2)
(67) replace quotientlimit (el, e2, x, k) by indeterminate when nastyquotient (el, e2)
(68) replace powerlimit (el, e2, x, k) by indeterminate when nastypower (el, e2)

Finally, this is the place to introduce miscellaneous outside information about special cases for which solutions are known. A large, important, non-recursive set of these cases is associated with the "order" considerations described by Hardy [9] .

For example, suppose that the problem at hand is

limit (x, plusinfinity, el/e2)

In general, it is not known that this should become zero.

(69) assert not knownzero (e1, e2, var)

It is known, of course, that

limit (x, plusinfinity, x/exp x)

should become zero:

(70) assert if equal (mainof e2, 'exp) and equal (arg (e2, 1), var) then knownzero (var, e2, var)

as should

limit (x, plusinfinity, log x/x)

(71) assert if equal (mainof e1, 'log) and equal (arg(e1, 1), var) then knownzero (e1, var, var)

Given that

limit (x, plusinfinity, a/b)

should become zero, so should

limit (x, plusinfinity, log a/b)

and

limit (x, plusinfinity, a/exp b)

(72) assert if equal (mainof e1, 'log) and knownzero (arg(e1, 1), e2, var) then knownzero (e1, e2, var)

(73) assert if equal (mainof e2, 'exp) and knownzero (e1, arg(e2, 1), var) then knownzero (e1, e2, var)

Now the following rule puts all these definitions into operation:

(74) replace quotientlimit (e1, e2, x, plusinfinity) by 0 when knownzero (e1, e2, var)

In the best of times, the result of limitsum, . . ., limitpower is that one may find the desired solution by applying the appropriate operator to the limits of the operator's former arguments.

(75) replace limitsum (el, e2, x, k, a, b) by a+b when atom a and atom b
(76) replace limitdifference (el, e2, x, k, a, b) by a-b when atom a and atom b
(77) replace limitproduct (el, e2, x, k, a, b) by a*b when atom a and atom b
(78) replace limitquotient (el, e2, x, k, a, b) by a/b when atom a and atom b
(79) replace limitpower (el, e2, x, k, a, b) by a**b when atom a and atom b

In all the interesting cases, of course, (75) - (79) are illegitimate. Here the remarks attending (30) and (31) must be kept in mind, and the choose function is the means by which I follow two branches at once. Choose, defined below, simply determines which alternative quotient form has reached a state which is expressible without limit notation.

Notice that the choose mechanism does not keep the EUC from being infinitely subject to change. In fact, choose increases the probability that the patience limit (§ 2.3.4(c)) will be struck. But once patience has stopped the infinite recursion along one branch, the system may be able to realize that the other branch has borne fruit.

(80) assert defined ('choose)
(81) assert limitfree(x) = null joint (funcsof x, list ('limit, 'sumlimit, 'limitsum, 'differencelimit, 'limitdifference, 'productlimit, 'limitproduct, 'quotientlimit, 'limitquotient, 'powerlimit, 'limitpower)
(82) replace choose (a, b) by a when limitfree a
(83) replace choose (a, b) by b when limitfree b

Here, finally, is l'Hôpital's rule. The next five rules take expressions with arithmetic main operators, turn these expressions into quotients, and differentiate.

(84) replace limitsum (el, e2, x, k, a, b) by log (-choose (
quotientlimit (exp(el)*d(el, x), exp(-e2)*d(e2, x), x, k),
quotientlimit (exp(e2)*d(e2, x), exp(-el)*d(el, x), x, k))) when
nastysum (a, b)
(85) replace limitdifference (el, e2, x, k, a, b) by log choose (
quotientlimit (exp(el)*d(el, x), exp(e2)*d(e2, x), x, k),
quotientlimit (exp(-e2)*d(e2, x), exp(-el)*d(el, x), x, k)) when
nastydifference (a, b)

(86) replace limitproduct(e1, e2, x, k, a, b) by -choose(
quotientlimit(d(e1, x)*e2**2, d(e2, x), x, k),
quotientlimit(d(e2, x)*e1**2, d(e1, x), x, k)) when
nastyproduct(a, b)

(87) replace limitquotient(e1, e2, x, k, a, b) by choose(
quotientlimit(d(e1, x), d(e2, x), x, k),
quotientlimit(d(e2, x)*e1**2, d(e1, x)*e2**2, x, k)) when
nastyquotient(a, b)

(88) replace limitpower(e1, e2, x, k, a, b) by exp(-choose
(quotientlimit(e1*d(e2, x)*(log e1)**2, d(e1, x), x, k),
quotientlimit(d(e1, x)*e2**2, e1*d(e2, x), x, k))) when
nastypower(a, b)

In (84)-(88), d is just the differentiation operator, defined as in § 4.2.3. These rules (and (64)-(68)) make an important assumption about the constants and functions liberated by (2)-(4), some of the "simplification" rules, (54)-(68), the "order" considerations, and (75)-(79). That is, that these expression-parts will be continually combined. The infinite predicate, after all, will not recognize

infinity + 1

or the like. The following rules might also be considered part of the shortcuts (54)-(63).

One could make some of the crucial functions opaque, and then have the following rules examine "NEED" (§ 2.3.4(h)) so as to take effect only when absolutely necessary. I have not bothered with such a refinement.

(89) evaluate f(x, y) when numberp x and numberp y and infixop f

(90) evaluate -n when numberp n

(91) replace -infinity by infinity

(92) replace -plusinfinity by minusinfinity

(93) replace -minusinfinity by plusinfinity

(94) assert nx( ) = infinite x and numberp n

(95) replace n+x by x when nx()

(96) replace x+x by x when infinite x

(97) replace x+y by indeterminate when nastysum(x, y)

(98) replace x-y by x+(-y) when infinite x or infinite y

(99) replace plusinfinity*n by plusinfinity when numberp n or equal(n, 'plusinfinity)
(100) replace plusinfinity*n by minusinfinity when numberp n and minusp n
(101) replace minusinfinity*x by -(plusinfinity*x)
(102) replace infinity*x by infinity
(103) replace x*y by indeterminate when nastyproduct (x, y)
(104) replace x/y by x*recip y when infinite x or infinite y

(105) replace n**plusinfinity by plusinfinity when numberp n
(106) replace n**plusinfinity by 0 when numberp n and lessp(n, 1)
(107) replace n**plusinfinity by infinity when numberp n and lessp(n, -1) or infinite n
(108) replace x**n by infinity when nx( )
(109) replace plusinfinity**x by plusinfinity
(110) replace minusinfinity**n by plusinfinity when even n
(111) replace infinity**n by plusinfinity when even n
(112) replace minusinfinity**n by minusinfinity when odd n
(113) replace x**n by recip(x**(-n))when nx( ) and minusp n or equal (n, 'minusinfinity)
(114) replace x**y by indeterminate when nastypower(x, y)

(115) replace f(indeterminate) by indeterminate
(116) evaluate recip x when numberp x
(117) replace recip 0 by infinity
(118) replace recip x by 0 when infinite x

(119) evaluate log x when numberp x
(120) replace log 0 by minusinfinity
(121) replace log x by plusinfinity when infinite x

(122) evaluate exp x when numberp x
(123) replace exp minusinfinity by 0
(124) replace exp infinity by indeterminate
(125) replace exp plusinfinity by plusinfinity

Additional rules similar to (116)-(125) are needed as new functions are brought into play.

The system described in this section was applied to a small but representative sample of problems from elementary and advanced texts [26, 31] . All of these problems were successfully solved.

This empty page was substituted for a blank page in the original document.

CHAPTER VI
LOOKING AHEAD

This chapter is devoted to (rather speculative) discussion of possible improvements on the extensions to the work described earlier.

6.1 MATCHING

The FAMOUS matching algorithm described in § 2.3.2.1 provides a comprehensive means of checking the tree-stucture of an algebraic expression. The elegance of the algorithm does not alter the fact that tree-structure is hopelessly inadequate to the problem at hand.

The trouble is that an expression Φ may "contain" an expression Ψ in a sense other than that of syntax (tree-structure). FAMOUS makes an inadequate detour toward this fact when it is handling rules whose forms have Abelian-group operators as ruletypes. FAMOUS will, for example, succeed in applying the rule

(1) replace x-x by 0

to the expression

(2) (a+ b) - (b+ c)

even though "b-b" is not a subexpression of (2). Similarly, Joel Moses' remarkable SCHATCHEN program [16, pp. 11-13] will find

(3) a*x**2+n when numberp n

in the expression

(4) z**2-4

FAMOUS will not, on the other hand, find

(5) exp(log z+ a)

in

(6) exp (x+ log y+ 2)

The trick which worked in the case of (1) - (2) is not universal, and one is led to adhocities like that of logsum (§4.2.2(d)).

From one point of view, of course, seeking a "universal" matching algorithm is futile. One might, after all, hope that such an algorithm would match 0 successfully with any expression which is identically zero.

But a lesson of Chapter V is surely that recursive undecidability can be a remote and unthreatening form of hopelessness. Reconsidering examples (1) - (2) and (3) - (4), I see that a rule was looking at an expression Φ for a portion satisfying certain conditions. That portion Ψ was to be changed, say to θ. By examining Φ and the rule, the system discovered the expression

(7) $\Omega\ (\Psi, w)$

equivalent to Φ; the result of the whole operation is

(8) $\Omega\ (\theta, w)$

These examples suggest a new notion of matching, perhaps a supermatch function, which uses the present (shape-testing and a-list-building) match as a subroutine. As before, a rule either succeeds in transforming an expression, or it leaves that expression unchanged.

Even though the expression is ultimately unchanged, however, the rule may have made any number of tentative changes (like the Φ-to-(7) change) before withdrawing. All changes, both these scratchwork ones and those performed by successful rules, are specified in terms of skeletons (like the present forms and substitutes) and a-lists.

A rule's applicability to a given expression is tested, then, by fitting that expression to an initial skeleton. A given rule may have a number of different initial skeletons; from the expression's point of view, these are alternatives.

Associated with each initial skeleton is a tfe which

(a) tests the a-list arising from the match;
(b) adds to or alters the a-list; and
(c) chooses which of the rule's next-level skeletons is, in conjunction with the a-list, to be thought of as describing the expression.

As might be inferred from my use of "next" in (c), the process may then be iterated. If the match is not found to be unsuccessful at one or another point, the expression is changed to that expression which is described by the last skeleton and a-list.

For example, a rule might have the initial skeletons

(9) 'x
(10) n*('x)
(11) 'x+m
(12) 'x-m
(13) n*('x)+m
(14) n*('x)-m
(15) m-n('x)
(16) m-'x

where the associated tfe's made sure that the n's and m's were numbers. The second stage might allow the rule to view any expression of any one of the forms (9) - (16) as being of the form

(17) a*('x)+b

The tfe associated with (17) might then ask about the expression's y-intercept or whatever.

There is any amount of floss which, it seems at first, can easily be added to this general scheme. For example, one might add a facility whereby identifiers (e.g., "quadratic") could be singled out and given fixed, special matching properties. These identifiers might even be given arguments (e.g., "quadraticin (x, a2, a1, a0)"). The possible mechanisms are many, and any is probably easily implemented in programming languages like FLIP [25] and

CONVERT [8].

The real problems of supermatch are not implementation proplems. The detailed design will interact strongly with the design of an associated control language; the user who has to advise supermatch must not feel that he is being forced to program. Quite possibly, much of the "floss" indicated above will necessarily be left behind.

6.2 ASSERTIONS AND RULES

Despite the arguments of § 3.2, the reader may feel that my opposition to "compiling rules into function definitions" is founded on a distinction without a difference. More subtly, the reader may feel that the problems discussed in Chapter III do not arise "in practical cases".

It is interesting to examine a simple example. Consider the assertions

(1) assert factorial (x) = x* factorial (x-1)
(2) assert factorial 0=1

Following these assertions and the rule

(3) evaluate factorial x when numberp x

the expression

(4) factorial (3)

will become

(5) 6

as it should. What rules, when "compiled" by some unspecified compiler, would have the same effect as (1) - (3) ? The obvious choices are

(6) replace factorial (x) by x*factorial (x-1) when numberp x

(7) replace factorial (0) by 1

but these are plainly wrong. From (4), these rules produce a greater or lesser portion of the infinite expression

(8) 3*(3-1)*(3-(1+1))*(3-(1+1+1))* . . .

The "(x-1)" of (6), unfortunately for (6), is a construction, not a difference. I try again:

(9) replace factorial (x) by x*factorial (y) when numberp x and leave ('y, x-1)
(7) replace factorial (0) by 1

Now, (9) and (7) are somewhat better: they take (4) into

(10) 3*2*1*1

leaving me with several alternatives. The rule

(11) evaluate x*y when numberp x and numberp y

will do the trick, of course, but it does much more than what is required. I can do no better than to replace (9) and (7) by

(12) replace factorial (x) by fact (x, factorial (y)) when numberp x and leave ('y, x-1)
(7) replace factorial (0) by 1
(13) replace fact (x, y) by z when numberp y and leave ('z, x*y)

6.3 EFFICIENCY

The current FAMOUS implementation presents a mean lethal dose of inefficiency. That is, approximately half of the problems given to FAMOUS have been solved so slowly that the users have lost interest in waiting for their solutions.

The figure "one-half", moreover, is probably charitable. The system's major user is overmotivated; all of the system's users have been more interested in seeing what the system can do than in seeing what the system can do

with some predetermined problem. The system has, of course, strongly reinforced an increasingly trivial problem mix.

Where is the time all going? At first, elegant inefficiences of implementation were thought to be the sink. A number of these inefficiencies were rooted out, but the resulting improvements in performance were disappointing.

This is not to say that implementation flaws could not be the time-waster. A number of important implementation strategies have not been attempted. Often, these implementation-oriented notions would have had repercussions as far away as the external appearance of the system.

For example, the present ruletype mechanism is easy to criticize. A FAMOUS differentiator has all its rules in a single pile which must be searched for each application. One might think in terms of a more fine-grained ruletype function, or one might give up ruletype altogether in favor of a system in which the EUC and the rules were parts of a multiply-connected plex-structure [21]. This last suggestion is most intriguing, but it is not practical within the present host system (CTSS LISP).

In other algebraic manipulators, several strategems are used to improve efficiency. Without drastic modification, these strategems are generally irrelevant to FAMOUS.

In FORMAC and elsewhere, for example, each expression carries an "already-simplified" bit. This bit serves to lock the "simplifier" out of certain subexpressions when the expression is reshuffled.

In FAMOUS, of course, the rules are continually changing. An expression which has been "simplified" today many still require further "simplification" tomorrow. An "already simplified" bit just cannot be set.

Even worse, the significance of the patience bound (§2.3.4(c)) must not be forgotten. Even though an expression has been processed by a given set of rules, that expression may still be subject to processing by those same rules.

Finally, consider the case of a rule which builds an expression e' out of subexpressions $e_1, \ldots, e_n$ taken from an old expression e. On occasion,

e' will surely deserve retreatment by the rules. If, for example, the rules are

(1) replace a/a by 1

and

(2) replace a+a by 2*a

then the processing of

(3) (sin 4 z) /2+(sin 4 z) /2

must not stop at

(4) 2*(sin 4 z) /2

On the other hand, the

(5) sin 4 z

of (4) need not be examined after (4) is reached. (5) was, after all, a subexpression of (3), and rule (2) was not applied until the subexpressions of (3) became stable.*

But this account of (5) is not perfectly general. Suppose, for example, that the EUC is an expression in x whose derivative at x=2 is wanted. It is reasonable to run the compressive evaluation and the expansive differentiation at once; I would use the command sequence

```
assert opaque ('d)
hold
replace x by 2
consider d($,x)
continue
```

* Here I am pushing the patience issue under the rug.

This scheme assumes, of course, that although

```
d(x**2, x)
```

will not turn into the senseless form

```
d(2**2, 2)
```

The "2*x" which does appear will have its "x" replaced by 2. But in the notation of the example (1) - (5) , this means that subexpressions of the old expression are being resubmitted to the rules.

To take another case of a strategem common in other algebraic manipulators, consider the class of rules including

(6) replace x-x by 0
(7) replace 0*x by 0
(8) replace x/x by 1

Rules in this class make superfluous entries on the a-list. In cases (6) - (8), the name "x" is not used in construction of the tfe or substitute.

Because they cause whole pieces of expressions to be discarded, and because they are so easy to implement, rules like (6) - (8) are extremely popular with existing "simplifiers". These rules are frequently and easily implemented in no distinguishable form: they are woven into the fabric of the system in innumberable places.

In FAMOUS applications, it is true that (6) - (8) are common rules. But

(9) replace x-y by indeterminate when nastydifference (x, y)
(10) replace x*y by indeterminate when nastyproduct (x, y)
(11) replace x/y by indeterminate when nastyquotient (x, y)

are also useful at times, and even

(12) replace x-x by pi
(13) replace 0*x by e

(14) replace x/x by pi**e

are legitmate, although a little unusual.

If FAMOUS knew, in some sense, that (6) - (8) were present and untrammeled by any of (9) - (14), it could do very nicely. But the present FAMOUS never makes any global observations.

With even the most limited inter-rule considerations, the prospects for improving efficiency seem very good indeed. A human being, given the rules

(15) replace d(c, x) by 0 when numberp c
(16) replace d(x, x) by 1
(17) replace d(u*v, x) by u*d(v, x) + v*d(u, x)
(18) replace 0*x by 0
(19) replace 1*x by x
(20) replace 0 + x by x

will, after reflection or experience, come to know the rule

(21) replace d(c*x, x) by c when numberp c

which, where it is applicable, is an efficient summary of all of (15) - (20). The argument which condemns (21) as implicit in — and superfluous to — (15) - (20) does not argue for speed of operation. One is reminded of Quine's discussion [16, p. 26] of economy of axiomatization versus economy of length of proof.

One is also reminded of the tradeoffs in compiler construction, where discrimination of special cases (for loops with DO-loop logic, procedure blocks called 0 times or one time, etc.) proves to be feasible and rewarding.

The compiler analogy is particularly encouraging. Just a few years ago, certain compiler techniques were thought to be so inherently inefficient that they would never rise above the status of academic curiosities. Now that those techniques are better understood, they are commonly as fast as the less elegant alternatives.

The present FAMOUS does not carry even the first suggestion of an attack upon this problem of inter-rule considerations. But the present FAMOUS has quite fulfilled its purpose if it has suggested only what the rewards of such an attack might be.

APPENDIX A

COMMAND DESCRIPTIONS

The commands of FAMOUS may be divided into four main groups:

(a) Four commands are concerned with expressions.
(b) Eleven commands are concerned with rules.
(c) Three commands are concerned with function definitions.
(d) Five commands have miscellaneous metafunctions.

In the syntactic descriptions below, the following conventions are used:

(a) e's always represent expressions, in the input form described in § D. 1.
(b) id's represent single names, in the input form described in § D. 2(a).
(c) n's represent integers, in the input form described in § D. 2(b).
(d) Capitalized strings are literally intended, as are punctuation marks other than brackets and braces.
(e) Optional phrases are shown in brackets.
(f) Alternatives are listed vertically between braces.

Summary of Commands

ABBREVIATE e AS e'

ASSERT [IF e THEN] $\left\{\begin{matrix} e' \\ \text{NOT } e' \\ e' = e'' \end{matrix}\right\}$

CONSIDER e $\left[\text{, WHERE } e_1 = e_1' \left[, e_2 = e_2' \left[, \ldots \left[, e_n = e_n'\right] \ \ldots\right]\right]\right]$

CONTINUE

DESK

EVALUATE e [WHEN e']

EXPAND id

HOLD

LEAVE e [WHEN e']

LISTEN

NOW { ABBREVIATE e AS e' | EVALUATE e [WHEN e'] | EXPAND id | REPLACE e BY e' [WHEN e''] }

QUIT

REASSERT id [(SCAN)]

RECONSIDER (n)

REPLACE e BY e' [WHEN e'']

RESET { EXPRESSION | EXPRESSIONS | RULES }

RETRIEVE id [(id_1 [, id_2 [.... [, id_n] ...]])]

RULE

SAVE { ASSERTIONS | EXPRESSION | RULES } AS id

SCAN RULES [(id_1 [, id_2 [, ... [, id_n] ...]])]

SUPPRESS (id_1 [, id_2 [, ... [, id_n] ...]])

A.1 COMMANDS CONCERNED WITH EXPRESSIONS

A.1.1 Consider

CONSIDER e $\left[\text{, WHERE } e_1=e'_1 \left[,e_2=e'_2 \left[,\ldots \left[, e_n=e'_n \right] \ldots\right]\right]\right]$

In the usual case, the expression e becomes the CEUC and the new wherelist is specified by the remainder of the command. If no wherelist is specified, the wherelist is made null.

If the peculiar name "$" is a member of

atomsof(e)

then

(1) the new CEUC is formed by substituting the old CEUC for every appearance of "$" in e, and

(2) the wherelist is made null.

In either case, the new EUC is finally printed out with the next sequential expression-number.

Examples:

```
consider x - f(sin 2 g), where f(z) = z**2, g=2*a
consider x**x**4 x
consider $+$
```

A.1.2 Save

SAVE EXPRESSION AS id

The EUC is stored in a file named

EXPS id'

where id' is a short name printed out in response to the save command. Any expression previously associated with this name (id) and this file (EXPS id') is lost.

The name id may not contain periods.

The utility of the save command is best understood in connection with the continue command (§A.4.1).

Examples:

save expression as fred
save expression as schroedingerequation

A.1.3 Reconsider

RECONSIDER (n)

The expression and wherelist which were numbered (n) become the EUC, which is printed out.

If no expression so numbered is known to the system, the system complains.

An elusive bug in the host system (CTSS LISP) occasionally causes FAMOUS to lose expressions numbered "(1)".

Examples:

reconsider (12)
reconsider (3)

A.1.4 Reset

RESET { EXPRESSION / EXPRESSIONS }

(a) The CEUC disappears.
(b) The expression number is set to 0.
(c) The numbered expressions leading up to the just-vanished one are forgotten.
(d) If the command is typed with the final "s", all expressions named via save (§A.1.2) are forgotten.

Examples:

reset expression
reset expressions

A.2 COMMANDS DEALING WITH RULES

A.2.1 Reset

RESET RULES

All the current rules disappear.

Example:

reset rules

A.2.2 Save

SAVE RULES AS id

All the current rules are stored on the disk in a file named

RULES id'

where id' is a short name printed out in response to the save command. Any set of rules previously associated with this name (id), and with this file (RULES id'), is lost.

The name id may not contain periods.

The utility of the save command is best understood in connection with the retrieve command (§A.2.3).

Examples:

save rules as george
save rules as almostwooldridge

A.2.3 Retrieve

$$\text{RETRIEVE id}\left[\ (\text{id}_1\left[,\ \text{id}_2\ \left[,\ldots\left[,\ \text{id}_n\right]\ldots\right]\right])\right]$$

The optional portion of this command is a descriptor list.

Certain rules from the saved package named id are added to the current stock. A rule r is taken if either

(a) the command's descriptor list is null, or
(b) the command's descriptor list contains some descriptor of r.

If no rule package named id is known to the system, the system complains.

Examples:

retrieve george
retrieve sam (*)
retrieve almostwooldridge (dx, abbrev)

A.2.4 Suppress

SUPPRESS (id_1 [, id_2 [, ... [, id_n] ...]])

The parameters of this command are descriptors.

Certain of the current rules are lost. A rule r is lost if the command's descriptor list contains some descriptor of r.

Examples:

suppress (*)
suppress (dx, abbrev, expand)

A.2.5 Scan

SCAN RULES [(id_1 [, id_2 [, ... [, id_n] ...]])]

The optional portion of this command is a descriptor list.

Certain of the current rules are displayed for the user's comment. The other rules are unaffected.

A rule r is among those scanned if either

(a) The command's descriptor list is null, or
(b) The command's descriptor list contains some descriptor of r.

The user may make any one of five comments after a rule is presented.

(a) "OK" will cause the rule to be retained.
(b) "NG" will cause the rule to be lost.
(c) "OKOK" will cause the rule to be retained and the scan to be discontinued.
(d) "NGNG" will cause the rule to be lost. In addition, printing of scanned rules is discontinued, and the "NG" response is assumed to have been given for each scanned but unprinted rule.
(e) "LABEL" id will cause the descriptor id to be added to the descriptors of the rule. The rule is printed out again for further comment.
(f) "UNLABEL" id will cause the descriptor id to be removed from among the descriptors of the rule. The rule is printed out again for further comment.

If the scan ends normally (i.e., not via (c) or (d)), the system prints 'DONE'.

Examples:

scan rules
scan rules (*)
scan rules (dx, abbrev)

A.2.6 Abbreviate

ABBREVIATE e AS e'

A new rule is added to the current stock (§A.2.12).

(a) The raw form is e
(b) The raw tfe is "$TRUE$"
(c) The raw substitute is e'

(d) The raw descriptor-list is the unit set of "ABBREV"

If the hold switch (§§ A.2.11, A.4.1, and A.4.3) is not on, the continue command (§A.4.1) is initiated after the rule has been added.

Examples:

```
abbreviate 2.71828 as e
abbreviate a*b**y as f(y)
```

A.2.7 Evaluate

EVALUATE e [WHEN e']

A new rule is added to the current stock (§A.2.12).

(a) The raw form is e
(b) The raw tfe is e', if e' is given, otherwise "$TRUE$"
(c) The raw substitute is "EVALUATE"
(d) The raw descriptor-list is the unit set of "EVALUATE"

If the hold switch (§§ A.2.11, A.4.1, and A.4.3) is not on, the continue command (§A.4.1) is automatically initiated after the rule has been added.

Examples:

```
evaluate log 2
evaluate sin x when numberp x
evaluate p(x, y) when numberp x and numberp y and
                                    infixop p
```

A.2.8 Expand

EXPAND id

The parameter of this command is the name of an abbreviation. That is, it is the "e" of

```
abbreviate 2.71828 as e
```

or the "f" of

abbreviate a*b**y as f(y)

The wherelist is searched for an abbreviation with this name. If no such abbreviation is found, the system complains and returns to command level.

If the abbreviation is found on the wherelist, in the form e = e', then

(a) It is deleted from the wherelist.
(b) The rule that created it is deleted from the current stock.
(c) A new rule is added to the current stock (§A.2.12).

    (1) The raw form is e
    (2) The raw tfe is "equal (id, 'id)"
    (3) The raw substitute is e'
    (4) The raw descriptor-list is the unit set of "EXPAND"

(d) If the hold switch(§§A.2.11, A.4.1, and A.4.3) is not on, the continue command (§A.4.1) is automatically initiated.

Examples:

expand e
expand f

A.2.9 Leave

LEAVE e [WHEN e']

A new rule is added to the current stock (§A.2.12).

(a) The raw form is e.
(b) The raw tfe is e', if e' is given, otherwise "$TRUE$"
(c) The raw substitute is "LEAVE"
(d) The raw descriptor-list is the unit set of "LEAVE"

If the hold switch(§§A.2.11, A.4.1, and A.4.3) is not on, the continue command (§A.4.1) is automatically initiated after the rule has been added.

Examples:

```
leave log 0
leave sin x when integer x
```

A.2.10 Replace

REPLACE e BY e' [WHEN e'']

A new rule is added to the current stock (§A.2.12).

(a) The raw form is e
(b) The raw tfe is e'', if e'' is given, otherwise "$TRUE$"
(c) The raw substitute is e'
(d) The raw descriptor-list is null

If the hold switch (§§ A.2.11, A.4.1, and A.4.3) is not on, the continue command (§A.4.1) is automatically initiated after the rule has been added.

Examples:

```
replace 1*y by y
replace x + y by factr * (other1 + other2) when commonfactor (x, y)
```

A.2.11 Now

NOW { ABBREVIATE e AS e' | EVALUATE e [WHEN e'] | EXPAND id | REPLACE e BY e' [WHEN e''] }

(a) The current stock of rules is hidden safely from the rest of the system.
(b) The hold switch is turned off.
(c) The command indicated by the parameters is initiated.
(d) The rules hidden in (a) are restored; the rule generated by (c) is lost.

Examples:

```
now abbreviate 2.71828 as e
now evaluate sin x when numberp x
now expand e
now replace 1*y by y
```

A.2.12 Addition of Rules

The rule which finally enters the current stock will generally be somewhat different from the rule first specified by the user. The following transformations are performed:

(a) The ruletype of the form is added to the rule's descriptor list.

(b) If

```
opaque(ruletype(getexp()))
```

has the value "$TRUE$", then ruletype(getexp()) is added to the descriptor list.

(c) The form is altered as described in § 2.3.5.

(d) If the peculiar name "$" appears in the form, it is replaced by the CEUC.

(e) For every function name f in the form which is

(1) Not a member of atomsof(the tfe) (§ 2.3.5), and

(2) not the name of a defined function, but

(3) a function name in the CEUC, a clause

```
equal(f,'f)
```

is conjoined to the tfe.

A.3 COMMANDS CONCERNED WITH FUNCTION DEFINITION

A.3.1 Assert

$$\text{ASSERT} \quad \left[\text{IF e THEN}\right] \left\{ \begin{array}{l} \text{e'} \\ \text{NOT e'} \\ \text{e' = e''} \end{array} \right\}$$

The intention is to define or redefine the outermost function of e'. In particular, if this function f is ever applied to arguments which satisfy the tfe e, then it is intended by this assertion to guarantee that f has the value "$TRUE$", "$FALSE$", or the value of e'', depending upon which form is used.

(a) If the optional part of this command is not present, e is taken to be "$TRUE$".

(b) An effective e'' is chosen ("$TRUE$", "$FALSE$", or the given one).

(c) Say that e' was given as $f(e_1, e_2, \ldots, e_n)$. If any e_i is a number; identical to some e_j, $j > i$; an APVAL; or not a single variable name; then e_i is replaced in e' by a new variable-name e_i', and the clause

$$\text{equal}\,(e_i, e_i')$$

is conjoined to the tfe e.

(d) Say that e' is now $f(e_1', e_2', \ldots, e_n')$. If any e_i' is

(1) not a member of atomsof(e), and

(2) either

(a) an atom-name or function-name in the CEUC, or

(b) the name of a constant, or

(c) the name of a defined function

then e_i' is replaced by its quotation in the value e'', and the clause

$$\text{equal}\,(e_i', '(e_i'))$$

is conjoined to the tfe e.

(e) If any function-names in e or e'' are not names of defined functions, warnings are printed.

(f) The system prints its interpretation of the polished assertion, and the user is expected to express his approval or disapproval:

(1) If the user types "OK" then the assertion is passed on to the inner assert mechanism (§A. 3. 2).
(2) If the user types "NG", the assertion is discarded.

Examples:

```
assert not meter()
assert maxtime() = 30
assert patience() = 5
assert opaque(z) = equal (x, '(')
assert factorial(z) = z*factorial(z-1)
assert factorial(0) = 1
assert expless(a, b, c)
assert if atom x and atom y and alphaorder(x, y) then expless(y, x, z)
assert if numberp x or numberp y then expless(x, y, z) = not numberp x
```

A. 3. 2 The Inner Assert Facility

The inner assert facility accepts one assertion (IF tfe THEN $f(id_1, \ldots, id_n) = e$) at a time.

(a) The assertion is placed on the list of all assertions made in this copy of FAMOUS.
(b) If the tfe is "$TRUE$", then the system is given the definition

$$f(id_1, \ldots, id_n) = e$$

Any previous definition of f is lost.
(c) If the tfe is not "$TRUE$" but the function f has a previous definition, say

$$f(id'_1, \ldots, id'_n) = \Phi$$

Then a new function g is introduced with the definition

$$g(id'_1, \ldots, id'_n) = \Phi$$

and the system is given the definition

$$f(id_1, \ldots, id_n) = \underline{\text{if}}\ tfe\ \underline{\text{then}}\ e\ \underline{\text{otherwise}}\ g(id_1, \ldots, id_n)$$

(d) If the tfe is not "$TRUE$" and the function f has no previous definition, then the system prints a warning and accepts the definition

$$f(id_1, \ldots, id_n) = \underline{\text{if}}\ tfe\ \underline{\text{then}}\ e\ \underline{\text{otherwise}}\ \text{"UNDEFINED"}$$

A.3.3 Save

SAVE ASSERTIONS AS id

The assertions on the present list are stored in a file named

ASSERT id'

where id' is a short name printed out in response to the save command. Any set of assertions previously associated with this name (id) and this file (ASSERT id') is lost.

The name id may not contain periods.

The utility of the save command is best understood in connection with the reassert command (§A.3.4).

Examples:

save assertions as basics
save assertions as transcendentals

A.3.4 Reassert

REASSERT id[(SCAN)]

The intention is to add the assertions saved in the package id to the present list, using the inner assert facility (§A.3.2).

If the "(SCAN)" parameter is present, each assertion is displayed for the user's comment. The user may make any one of four comments after an assertion is displayed.

(a) "OK" will cause the assertion to be handed on to the inner assert facility.
(b) "NG" will cause the assertion to be ignored.
(c) "OKOK" will cause the assertion to be handed on to the inner assert facility. In addition, the printing of incoming assertions is discontinued for the remainder of the command, and the response "OK" is considered to have been given for each unprinted assertion.
(d) "NGNG" will cause the command to be immediately discontinued.

Examples:

```
reassert basics
reassert transcendentals (scan)
```

A.4 MISCELLANEOUS COMMANDS

A.4.1 Continue

CONTINUE

(a) If the CEUC is null (as it is after reset, § A.1.4), then the system complains and returns to command level.
(b) If any name of a saved expression (§ A.1.2) is found among atomsof(getexp()), then that expression is substituted for that name. The wherelist of the saved expression is added, where consistent, to the wherelist of the EUC.
(c) The hold switch is turned off.
(d) The EUC is thrown to the rules.
(e) The expression number is incremented.
(f) The new EUC is printed.

Example:

```
continue
```

A.4.2 Desk

DESK

The system is thrown into a desk-calculator mode of operation. Successive expressions are read from the typewriter, passed to the rules, and generally discarded. Precisely,

(a) An expression is read from the console.
(b) If it is "STOP" the system returns to command level.
(c) The CEUC is substituted for each appearance of "$" in the expression.
(d) The expression becomes the CEUC.
(e) The wherelist becomes null.
(f) The continue command is initiated.
(g) Control returns to (a).

Example:

desk

A.4.3 Hold

HOLD

The hold switch is turned on. The utility of the hold command is best understood in connection with the abbreviate, evaluate, expand, leave, and replace commands (§§ A.2.6–A.2.10).

Example:

hold

A.4.4 Listen

LISTEN

See § E.3.

Example:

listen

A.4.5 Quit

QUIT

See § E.3.

Example:

quit

This empty page was substituted for a blank page in the original document.

APPENDIX B

THE FAMOUS EVALUATION ROUTINE

The value of an expression is computed as follows:

(a) The host system (CTSS LISP) forces the following names to be distinguished as "APVALS":

blank	eor	oblist
breaks	eqsign	period
cleanout	floydftab	pluss
colon	floydgtab	prime
comma	fnflags	rpar
cr	fsleft	singles
dash	fwleft	slash
dollar	inlist	star
eof	lpar	tytab

If e is an APVAL, then e has a predetermined value which is generally not useful to FAMOUS. The APVALs should consequently be avoided.

(b) If e is "$TRUE$", "$FALSE$", a number, a constant, or the name of a defined function, then the value of e is e.

(c) If e is "NEED", then the value of e is a set of descriptors, as described in § 2.3.4(i).

(d) If e is a name which appears as the first part of a pair on the a-list, then the value of e is the second part of that pair.

(e) If e is the quotation of an expression e', then the value of e is e'.

(f) If e is

$$e_1 \text{ AND } e_2 \text{ AND } \ldots \text{ AND } e_n$$

then the successive e_i's are evaluated until e_K is found to have the value "$FALSE$". When this happens, the value of e is said to be "$FALSE$" and $e_{K+1}, \ldots, e_n$ are not evaluated. If all of $e_1, \ldots, e_n$ have value "$TRUE$", then e has value "$TRUE$".

(g) If e is

$$e_1 \text{ OR } e_2 \text{ OR } \ldots \text{ OR } e_n$$

then the successive e_i's are evaluated until e_K is found to have value "$TRUE$". When this happens, the value of e is said to be "$TRUE$" and $e_{K+1}, \ldots, e_n$ are not evaluated. If all of $e_1, \ldots, e_n$ have value "$FALSE$", then e has value "$FALSE$".

(h) If e is $f(e_1, \ldots, e_n)$, and if f is a defined function, then the value of e is the result of applying the function named by f to the values of $e_1 \ldots e_n$.

(i) If e is $f(e_1, \ldots, e_n)$, and if f appears as the first part of an entry on the a-list whose second part is g, then the value of e is the value of $g(e_1, \ldots, e_n)$.

Several parts of the evaluation routine have not been described. In the first place, I have not discussed cases of conflict: names on the a-list which are also APVAL's, etc. There are rules governing such conflict, but I find it improbable that a user of FAMOUS could make anything useful of his knowledge of these rules.

Nor have I discussed some additional ways in which it is possible for an expression to acquire a value. These ways, like the conflict-rules of the previous paragraph, are accidental bequests of the implementation. They seem, once again, to have no proper interest for the user of FAMOUS.

APPENDIX C

FUNCTIONS

Throughout this appendix, e's are expressions and b's are truth-values.

C.1 FUNCTIONS WHICH MUST BE DEFINED BY THE USER

Use	Purpose
expless(e_1, e_2, b)	Expless is the system's source of advice about the ordering of products and sums. If the user prefers $e_1 + e_2$ to $e_2 + e_1$ he should be sure that expless(e_1, e_2, $FALSE$) is $FALSE$. Similarly, expless(e_1, e_2, $TRUE$) should be $FALSE$ if e_1*e_2 is preferred to e_2*e_1 For example, suppose the user favors the order numbers/names/(complex factors) in products, and (complex terms)/names/numbers in sums. Then he might define expless by the assertions

```
assert expless(a, b, c)
assert complexity (x) = 2
assert if atom x then complexity x = 1
assert if numberp x then complexity x = 0
assert expless (a, b, $FALSE$) = lessp (complexity
       a, complexity b)
assert expless (a, b, $TRUE$) = not expless (a, b,
       $FALSE$)
```

Use	Purpose
	To these assertions, alphabetical ordering of names and other refinements might now be added. Of course, the trickiness of expless would not have been thrown to the user if only prosaic orderings could be specified.
maxtime()	See § 2.3.4(d); typical values run between 30 and 100.
meter()	If meter() does not have the value "$FALSE$", then the system will clock its excursions from command level, and the running time (in tenths of seconds) will be printed upon each return.
opaque(f)	See § 2.3.4(h)
patience()	See § 2.3.4(c)

C.2 NUMERICAL PREDICATES

Use	$TRUE$ Iff
fixp(x)	x is in integral internal representation
floatp(x)	x is in non-integral internal representation
greaterp(x, y)	x > y
lessp(x, y)	x < y
minusp(x)	x < 0
onep(x)	zerop(x-1)
zerop(x)	$\|x\| \leq 3*10^{-6}$

C.3 NUMERICAL FUNCTIONS

The five arithmetic operators are available. Exponentiation, which is written with FORTRAN's double asterisk, has the peculiar definition

x**y = if minusp y and floatp y then $x^{|y|}$ otherwise x^{y}

Other functions available include the following:

Use	Comment
addl(x)	x+1
asin(x)	$Sin^{-1}(x)$

Use	Comment
atan(x)	$Tan^{-1}(x)$
cosh(x)	
exp(x)	e^x
fix(x)	(sign of x) (greatest integer $\leq \|x\|$), in integral representation
float(x)	x, in non-integral representation
log(x)	natural log
max(x, y, ... , z)	largest of x, y, ... , z
min(x, y, ... , z)	smallest of x, y, ... , z
minus(x)	-x
recip(x)	1/x
sin(x)	
sinh(x)	
subl(x)	x-1
tanh(x)	

C.4 EXPRESSION-HANDLING PREDICATES

Use	$TRUE$ Iff
afixp(e)	numberp (e) and fixp (e)
anti(e)	e is - or /
associativep(e)	e is +, *, AND, or OR
atom(e)	e is a single number or name
commonfactor(e_1, e_2)	factor(e_1, e_2) or factor(e_2, e_1) or e_1 and e_2 have an explicit common factor. If commonfactor(e_1, e_2) has the value "$TRUE$", then commonfactor has used leave to set "FACTR", "OTHER1", and "OTHER2" on the a-list, so that e_1 = FACTR*OTHER1 e_2 = FACTR*OTHER2
constantp(e)	numberp(e)

Use	$TRUE$ Iff
defined(e)	e is the name of a defined function
equal(e_1, e_2)	(numberp(e_1) and numberp(e_2) and zerop(e_1 - e_2)) or (e_1 and e_2 are identical)
factor (e_1, e_2)	e_1 is an explicit factor of e_2. If factor(e_1, e_2) has the value "$TRUE$", then factor has used leave to set "OTHERFACTOR" on the a-list, so that $e_2 = e_1$*OTHERFACTOR
infixop(e)	e is +, -, *, /, **, AND, or OR
numberp(e)	e is a number
sum(e)	e is a sum or difference
typeout(e)	always "$TRUE$", types e on the console

C.5 EXPRESSION-HANDLING FUNCTIONS

Use	Value
arg(e, n)	nth argument of outermost function of e
atomsof(e)	See § 2.3.5
canonical(e)	A number, invariant with e. If e is a number, then e.
funcsof(e)	Set of function-names appearing in e.
leave(id, e)	See § 2.3.2.2
listif(e)	If atom(e) then unit set of e otherwise e.
mainof(e)	If atom(e) then e otherwise outermost function-name of e.
ruletype(e)	See § 2.3.3

C.6 MISCELLANEOUS PREDICATES

Use	$TRUE$ Iff
alphaorder(id_1, id_2)	id_1 is lexicographically $\leq id_2$
member(x, y)	$x \in y$
null(x)	x is empty
subset(x, y)	$x \subset y$

C.7 MISCELLANEOUS FUNCTIONS

Use	Value
getexp()	CEUC
getn()	Expression number
gettype()	Ruletype(getexp())
joint(x, y)	If null ($x \cap y$) then the null set otherwise some member of $x \cap y$
list(x, y, ... , z)	Set whose members are x, y, ... , z
setdifference(x, y)	$\hat{z}(z \in y \text{ and } z \notin x)$
union(x, y)	$x \cup y$

This empty page was substituted for a blank page in the original document.

APPENDIX D

CONSOLE INPUT-OUTPUT

The console I/O facilities are physically and logically a separate part of FAMOUS.

D.1 INPUT SIDE

The input side of FAMOUS consists of four separate programs.

(a) The rdline routine collects characters into names and determines when a logical line has been completed.
(b) The clean routine performs miscellaneous functions to simplify the syntactic structure of the input stream.
(c) The floydpolish routine is a table-driven precedence-grammar phrase finder.
(d) The rephrase routine rearranges the phrases discovered by floydpolish into the standard internal form used by FAMOUS.

D.1.1 Rdline

D.1.1.1 Dividing the Input Stream Into Useful Units

The primary task of rdline is division of the input stream into meaningful groups of characters. These groups are known as elements.

(a) A string beginning with one of

( )
' (apostrophe)
, (comma)
=
+
- (minus)
/

is an element which ends with that character.

(b) A string beginning with an asterisk is an element which ends with

(1) that character, if the next character is not also an asterisk, or
(2) the next character, if it is also an asterisk.

(c) A string beginning with a letter is an element which ends just before the first following one of

(blank or carriage return)
(
)
*
' (apostrophe)
, (comma)
/
=
+
- (minus)

(d) A string beginning with a number or decimal point is an element which ends just before the first following one of

(blank or carriage return)
(plus sign or minus sign not preceded by letter "E")
(
)
*
' (apostrophe)
, (comma)
=
/

D.1.1.2 Determining the End of a Logical Line

At the end of each physical typed line, or whenever a superfluous right parenthesis is detected, rdline must decide whether the logical line has come to an end. Rdline will consider the logical line complete when

(a) All left parentheses have been matched, and
(b) The last element is not an infix operator, an equal sign, a comma, "WHEN", "THEN", "WHERE", "AS", or "BY".

Physical input lines may not be longer than 72 characters. Input is in 6-bit mode, so the CTSS kill (?) and delete (") conventions [5,§AC.2.02] are available.

D.1.2 Clean

The primary purpose of clean is highly implementation-dependent. In addition, clean serves a special function with respect to unary plus and minus signs.

Floyd [7, pp. 322-323] and others have observed that unary plus and minus signs may be distinguished by the elements which precede them. Clean does this and then either

(a) Causes a following number to absorb the sign, or
(b) Changes the sign to an unambiguous function-name.

D.1.3 Floydpolish

Floydpolish is exactly the precedence-grammar phrase-marker described by Floyd [7]. The grammar now used is

<exp>:: = <term>|<exp><aop><term>

<term>:: = <factor>|<term><mop><factor>

<factor>:: = <primary>|<primary>**<factor>

<primary :: = <idn>|<idn><args>|(<exp>)

<args>:: = <primary>|( )|(<arglist>)

<arglist>:: = <exp>|<exp>,<arglist>

<aop>:: = +|-|OR

<mop :: = *|/|AND

<idn>:: = Any element not (,), comma, **, a mop, or an aop.

which has the precedence table

element	f	g
⊢	1	-
⊣	-	1
(	2	10
,	3	4
aop	5	4
mop	7	6
**	7	8
idn	9	10
)	9	2

Some sample phrases generated by this grammar are

```
4 sin 3 log x
f(b( ) + 3 x)**b(  )**3.6
lessp(x, y) and lessp(y, z)
```

D.1.4 Rephrase

Rephrase is a rather specialized routine which is almost wholly determined by vagaries of the current internal representation of expressions. The one interesting feature of rephrase is provoked by phrases of the form

< idn > < args >

For example, the expression

4 sin f(x)

contains three such phrases.

The innermost of these, "f(x)", is easy to handle. The parentheses give it away as a case of functional application.

In the next case, that of "sin f(x)", rephrase asks if sin is a defined function. Since it is, this phrase is also considered to express functional application.

In the final case, that of the whole expression, neither of the previous arguments is applicable. The expression is treated, therefore, as if it had been written

```
4* sin(f(x))
```

D.2 EXTERNAL REPRESENTATIONS

(a) A string of 30 or fewer letters, periods, dollar signs, and digits, the first character of which is a letter, represents the name of a function or variable.

(b) A string of decimal digits represents an integer. A preceding sign is optional.

(c) A string consisting of

(1) Decimal digits
(2) A decimal point
(3) Decimal digits

represents a real number. A preceding sign is optional. For example, "3.14159" represents an approximation to pi.

(d) A string consisting of (1) - (3) of (c) and

(4) The letter 'E'
(5) Optionally, a sign
(6) One or two decimal digits

represents a real number. Let the numbers represented by (1) - (3) and (5) - (6) be x and y, respectively. Then the number represented by (1) - (6) is x*10**y.

A preceding sign is optional. For example, "31.4159E-1", "+ 3.14159E0", and "0.314159E1" all represent the same approximation to pi.

(e) The apostrophe represents the quotation function. For example,

```
'x
```

and

'(x)

equivalently represent the quotation of x.

(f) Extra blanks are harmless; they may be freely used to improve the readability of input.

(g) Similarly, extra right parentheses at the end of a logical line are harmless; counting them is a waste of time.

D.3 OUTPUT SIDE

FAMOUS' output routines are much simpler than the input ones. They utilize only the most conservative precedence relations, and they seem to have no external interest.

Every expression printed out by FAMOUS is in a permissible input form.

APPENDIX E

OPERATING CONSIDERATIONS

E.1 ERROR PROCEDURES

E.1.1 Mistyped Commands or Responses

Whenever FAMOUS expects a stereotyped input, a non-standard line is handled in the following standard way:

(a) The message "Eh?" is printed,*
(b) Any waiting typed input is discarded, and
(c) FAMOUS tries to read a new input line to replace the erroneous one.

E.1.2 Host System Errors

The host system (CTSS LISP) will complain under various circumstances. For example, the user may ask that a valueless expression be evaluated, or he may ask the system to deal with numbers of unworkable size.

After a complaint from the host system, FAMOUS is back at its command level.

E.1.3 Interrupt Signals

A console interrupt signal [5, § AC.2.02] causes an immediate host system error (§ E.1.2). These signals may be useful if faulty function-definitions cause looping in the system.

For brief periods during the execution of certain commands, the data-base is in an inconsistent or meaningless state. Console interrupt signals are ignored during these periods.

---

*"Thus, the user is forced to read his input line to find the error, rather than possibly being misled by a message unrelated to the actual error." [24, p. 462]

E.2 SCANNING

The user may occasionally cause FAMOUS to scan some list of rules or assertions. He is then given an opportunity to comment on each of the scanned items.

The user may indicate in one of these comments that he wishes the scan to stop. In general, however, the scan will continue until the list being scanned is exhausted. When a list is exhausted, the system prints the message "DONE".

E.3 COMMAND LEVELS

FAMOUS is a looping LISP program. To get out of this loop, the quit command may be used.

To move in the other direction, the listen command has been provided. This command simply executes the CTSS LISP function listen[ ] [12, p. 2]; return from listen puts the system back at FAMOUS command level.

BIBLIOGRAPHY

1. Bar-Hillel, Y., Luncheon conversation, November 30, 1965

2. Brown, W.S., "A Language and System for Symbolic Algebra on a Digital Computer", Proceedings of the 1965 IBM Scientific Computing Symposium on Computer - Aided Experimentation (to appear)

3. Chagnon, Spencer O., "The Simulation Rule and Admissible Events in Simulation," in (11)

4. Craik, K.J.W., The Nature of Explanation, Cambridge, England, The University Press, 1943

5. Crisman, P.A. (ed.), The Compatible Time-Sharing System: A Programmer's Guide (second edition), Cambridge, M.I.T. Press, 1965

6. Feldman, Jerome A., "A Formal Semantics for Computer-Oriented Languages", Ph. D. Thesis, Carnegie Institute of Technology, 1964

7. Floyd, Robert W., "Syntactic Analysis and Operator Precedence", Journal of the Association for Computing Machinery, X (July 1963), pp. 316-333

8. Guzmán-Arenas, Adolfo, and Harold v. McIntosh, "CONVERT", presented at the Symposium on Symbolic and Algebraic Manipulation, March 29 - 31, 1966

9. Hardy, G. H., Orders of Infinity, London, Cambridge University Press, 1924

10. Holt, Anatol W., "The Semantics of the Simulation Rule", in (11)

11. Holt, Anatol W., et al, ⊃ -Theory: Technical Documentary Report # 1, Princeton, New Jersey, Applied Data Research, 1965 (articles paginated separately)

Bibliography (continued)

12. Martin, William A., and Timothy P. Hart, "Time-Sharing LISP", M.I.T. Artificial Inteligence Project Memorandum #67, 1964 (unpublished)

13. McCarthy, John, "Programs with Common Sense", Mechanisation of Thought Processes, London, H.M. Stationery Office, 1959 pp. 77-84

14. _____, "The Wang Algorithm for the Propositional Calculus Programmed in LISP", M.I.T. Artificial Intelligence Project Memorandum #14, 1960 (unpublished)

15. Minsky, Marvin L., "Recursive Unsolvability of Post's Problem of 'Tag' and Other Topics in Theory of Turing Machines", Annals of Mathematics, LXXIV (1961), pp. 437-455

16. Moses, Joel, "Symbolic Integration", M.I.T. Artificial Intelligence Project Memorandum #97, 1966 (unpublished)

17. Quine, Willard Van Orman, From a Logical Point of View, Cambridge, Harvard University Press, 1961

18. _____, Mathematical Logic, Cambridge, Harvard University Press, 1958

19. _____, Word and Object. Cambridge, Technology Press, 1960

20. Richardson, Daniel, PhD Thesis, University of Bristol (England), 1966

21. Ross, Douglas T., and Jorge E. Rodriguez, " Theoretical Foundations for the Computer-Aided Design System", Proceedings of the 1963 Spring Joint Computer Conference, Baltimore, Spartan Books, 1963, pp. 305-322

Bibliography (continued)

22. Sammet, Jean E., "An Annotated, Descriptor-Based Bibliography on the Use of Computers for Non-Numerical Mathematics", I.B.M. Technical Report TR 00.1427, 1966

23. Shapiro, Robert M., "System Coordination and a Formal Definition of Coupling ", in (11)

24. Shaw, J.C., "JOSS: A Designer's View of an Experimental On-Line Computing System", Proceedings of the 1964 Fall Joint Computer Conference, Baltimore, Spartan Books, 1964, pp. 455-464

25. Teitelman, Warren, "FLIP: A Format List Processor". M.I.T. Artificial Intelligence Project Memorandum #87, 1965 (unpublished)

26. Thomas, George B., Jr., Calculus and Analytic Geometry (third edition). Reading, Massachusetts, Addison-Wesley, 1960

27. Tobey, R.G., "Experience with FORMAC Algorithm Design ". I.B.M. Technical Report TR 00.1413, 1966

28. Tobey, R.G., Bobrow, R. J., and S.N. Zilles, "Automatic Simplification in FORMAC ", I.B.M. Technical Report TR 00.1343, 1965

29. Wang, Hao, "Toward Mechanical Mathematics", I.B.M. Journal of Research and Development, IV (January 1960), pp. 2-22

30. Warshall, Stephen, "An Informal Description of ℒ -Theory", (unpublished), 1966

31. Widder, David V., Advanced Calculus. Englewood Cliffs, New Jersey, Prentice - Hall, 1947

32. Wooldridge, Dean, Jr., "An Algebraic Simplify Program in LISP", Stanford Artificial Intelligence Project Memorandum #11, 1963

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-35

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 118 (125-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): FOLLOWS TITLE PAGE & PAGES 29, 63, 91, 99, 111

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-118) UNNUMBERED TITLE & BLANK PAGES, iii-vi, 1-29, UNNUMBERED BLK, 31-63, UNNUMBERED BLK, 65-91, UNNUMBERED BLK, 93-99, UNNUMBERED BLK, 101-111, UNNUMBERED BLK
(119-125) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGTS (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 12/28/95 Date Returned: 12/28/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
An On-Line System for Algebraic Manipulation

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Doctoral Thesis, Applied Mathematics, Harvard University, July 1966

5. AUTHOR(S) *(Last name, first name, initial)*
Fenichel, Robert R.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1966	120	32

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Office of Naval Research, Nonr-4102(1)	MAC-TR-35 (THESIS)
b. PROJECT NO. NR 048-189	
c. RR 003-09-01	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT FAMOUS is an on-line system for the manipulation of linguistic forms. Although these forms can have quite arbitrary interpretations, the standard interpretation is that they are algebraic expressions. FAMOUS allows its "algebraic expressions" to include arbitrary functions which may or may not be defined. In this way, regular non-algebraic constructions may be concealed as arguments of ad hoc functions. Rules of local change are the heart of FAMOUS, and supplied by the user. Using these rules, FAMOUS looks at an algebraic manipulation as a series of local changes. The centrality of proximity in FAMOUS was orginally prompted by ∇-theory, which might be called the study of proximity.

The presentation in Chapter II is complete, but it has rather a cookbook tone. Chapter III is a more reflective attempt to define the power and nature of the system. Algebraic "simplification" has been a benchmark of algebraic manipulators, and it is discussed in Chapter IV. A more novel application, that of limit problems, is discussed in Chapter V. Finally, Chapter VI consists of miscellaneous remarks about possible and impossible lines of further work.

14. KEY WORDS
Algebraic manipulation
Machine-aided cognition
Multiple-access computers
On-line computer systems
On-line manipulators
Real-time computer systems
Time-sharing
Time-shared computer systems

DD FORM 1 JAN 64 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/28/1995

M.I.T. Libraries
Document Services